Two Sides of the Same Coin: Counting and Enumerating Keys Post Side-Channel Attacks Revisited.

Daniel P. Martin[1], Luke Mather[2], and Elisabeth Oswald[3]

[1] School of Mathematics, University of Bristol, Bristol, BS8 1TW, UK, and the Heilbronn Institute for Mathematical Research, Bristol, UK.
dan.martin@bristol.ac.uk

[2] Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom.
luke.mather@bristol.ac.uk

[3] Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom.
elisabeth.oswald@bristol.ac.uk

Abstract. Motivated by the need to assess the concrete security of a device after a side channel attack, there has been a flurry of recent work designing both key rank and key enumeration algorithms. Two main competitors for key ranking can be found in the literature: a convolution based algorithm put forward by Glowacz *et al.* (FSE 2015), and a path counting based algorithm proposed by Martin *et al.* (Asiacrypt 2015). Both key ranking algorithms can be extended to key enumeration algorithms (Poussier *et al.* (CHES 2016) and Martin *et al.* (Asiacrypt 2015)). The two approaches were proposed independently, and have so far been treated as uniquely different techniques, with different levels of accuracy. However, we show that both approaches (for ranking) are mathematically equivalent for a suitable choice of their respective discretisation parameter. This settles questions about which one returns more accurate rankings. We then turn our attention to their related enumeration algorithms and determine why and how these algorithms differ in their practical performance.

Keywords: Key Rank, Key Enumeration, Side Channel Attacks

1 Introduction

Side-channel analysis (SCA) is a powerful tool for extracting cryptographic keys from secure devices. For instance, if an adversary can measure the power consumption of a device performing cryptographic operations, then the resulting power traces may subsequently lead to the recovery of the secret key [6]. SCA attacks typically utilise a *divide-and-conquer* strategy: they target small portions of a key individually, obtaining information on the distribution of the likelihood of each portion, before combining these results to recover a full key.

Until recently, SCA attacks have been considered to be "all-or-nothing" attacks: if the attack did not perfectly identify the correct value for each portion of the key as the most likely, then the attack would be considered a failure. However, beginning with the work of Veyrat-Charvillon *et al.* [16] in 2012, it is now possible for an adversary to make use of the information produced by an *imperfect* attack. In an imperfect attack, the adversary finds some, but not sufficient side-channel information pertaining to the key. Consequently, they must then *enumerate* and test the most likely candidate keys (in order from the most to the least likely using known plaintext and ciphertext pairs) to determine whether a candidate is the correct key. This scenario is significant for evaluation bodies and certification authorities—the potential implication of this recent research has prompted JHAS (JIL Hardware-related Attacks Subgroup; an industry led group that essentially defines Common Criteria security evaluation practice) to set up a specific working group to address the issue.

Informally, the number of candidate keys an adversary must enumerate (and test) after an imperfect side-channel attack before arriving at the correct key is termed the *rank* of the key. Recent efforts [1,3,5,12,18,17] considered determining the rank of the correct (known) key after the side-channel phase of an attack.

Although the rank is an extremely informative measure of security, it does not completely capture the strength of an adversary. If after an attack the rank of a key is 2^{40}, then the adversary (who does not know this) must generate and eliminate the $2^{40} - 1$ candidate keys that were (incorrectly) rated to be more likely by the attack. The generation and testing of candidate keys is thus more costly than just computing the rank of a key, and it is important to know how challenging this task is in practice (especially if it does not scale linearly). Hence, it is important to characterise the existing key enumeration algorithms in terms of their run-time, as well as whether the adversary can parallelise their effort. The most recent works of Poussier *et al.* [15] and Martin *et al.* [12] go some way towards this goal, but come to somewhat different conclusions. They treat each others approaches as uniquely different, argue about differences in accuracy and report differences in performance numbers (albeit measured on different platforms).

1.1 Our Contributions

We look "under the hood" of the mathematical representation of path counting, used by Martin *et al.* [12], by utilising an elegant representation recently given in [11]. The intuition from this representation is that some aspects of the counting can be expressed as "binning" items for a specific weight, as used by Poussier *et al.* [15], and vice versa, the binning of scores seems to relate to counting the number of integer scores. Thus the two approaches could be mathematically equivalent.

Our first contribution is hence to make this intuition formal. We thus show how to express the histogram method as a (recent) version of the path counting approach, and thus show mathematical equivalence between the two ranking methods. Our proof is based on the fact that the convolution based approach

assumes equally spaced bins, and this implies an equivalence between the "precision" parameter of the path counting approach and the "number of bins" parameter of the convolution based approach. Using this we rewrite the equations that underly the convolution based approach, such that they are equivalent to the equations of the path counting approach. By showing mathematical correspondence between "precision" and "number of bins" we also settle any open questions about the accuracy of those methods (both methods are equally accurate).

Whilst both methods arrive at the same result mathematically (assuming use of the same discretisation parameter), there is a clear difference in how they are expressed algorithmically, which implies that their practical performance will be different. Whilst a rigorous complexity analysis of the path-counting based algorithms is available, we argue that a similar analysis for the convolution based approach must depend on assumptions about the distribution of values in the (intermediate) histogram bins. Thus rather than making artificial assumptions, we suggest relying on practical experiments to compare its performance with the best variation of the path-counting based approach. To achieve a like-for-like comparison we run both on the discretisation parameter for which their underlying mathematical representations are equivalent. Our comparison shows that up to 12 bits of precision (which is equivalent to 2^{12} bins) the convolution based method is faster than path-counting. From 12 bits of precision onwards path-counting wins.

Precision is crucial for the ability to parallelise large enumeration efforts across many cores. Thus we conclude that for small to medium size search efforts, convolution is the better choice, whilst for large scale search efforts a path-counting implementation is preferable.

1.2 Outline

Section 2 outlines the notation, gives a useful example, and provides some basic definitions. Section 3 explains the two approaches to ranking as well as their related enumeration algorithms that we study in this work. Section 4 proves the mathematical equivalence of path counting based and convolution based ranking (as defined in Section 3). Section 5 examines the real-world performance of the enumeration algorithms when implemented in the same language, executed on the same machine, and using corresponding levels of precision. Section 6 discusses considerations for parallel implementations, in particular with respect to the role that precision plays. We conclude in Section 7. To maintain a good flow throughout this paper, we place all algorithms in Appendix A.

2 Preliminaries

In this section we begin by introducing the notation that will be used for the remainder of the paper. We also recall the key rank and enumeration definitions.

2.1 Notation and Setup

We use a bold typeface to denote multi-dimensional variables. A key $\mathbf{k}$ can be partitioned into m independent subkeys, each of which can take one of n possible values (for ease of notation, we assume that all subkeys are of the same size). We denote this as $\mathbf{k} = (k^1, \dots, k^m)$ and mark the true secret key as $\mathbf{s} = (s^1, \dots, s^m)$.

We focus on side-channel attacks on symmetric encryption schemes, which typically return a score vector per subkey as a result. A side-channel attack takes in a set of leakages (of size N, where N might be as low as one) corresponding to known plaintexts $x_i \in \mathcal{X}$, $i = 1, \dots, N$, and by making some guesses about a small part (the subkey) of the unknown key, returns the output of a function that is termed a distinguisher in the side-channel literature. There are many techniques for side channel analysis, using different types of distinguishers, which result in different types of scores (see [9] for an overview). We assume that we deal with distinguishers that produce additive scores that indicate the likelihood of subkey values. Thus each element in the distinguishing vector $\mathbf{D}^i$ (for subkey k^i) contains a *score* associated with how likely the associated subkey value is to the be correct key. The score $D_{j,i}$ corresponds to the likelihood of subkey i taking value j.

The subkey distinguishing vectors all have the same size and thus can be arranged into a distinguisher matrix $\mathbf{D}$ (each column vector corresponds to a subkey $\mathbf{D}^i$). The result of a side-channel attack is hence a set of *distinguishing vectors*, which hold the information about subkeys (when studied individually), and the entire key (when studied jointly).

2.2 Running Example

We introduce a running example which will be used throughout the paper to help explain all of the algorithms detailed. We will consider a secret key $\mathbf{s} = (3, 1)$ (consisting of two subkeys, each of which can take one of three possible values $1, 2, 3$). After a side channel attack a (hypothetical) distinguisher outputs the (additive) score matrix representing (log)likelihoods, such that the largest value corresponds to the most likely key[4]:

$$\mathbf{D} = \begin{pmatrix} 6/11 & 2/11 \\ 3/11 & 6/11 \\ 2/11 & 3/11 \end{pmatrix}$$

The path counting algorithm by Martin *et al.* explicitly converts distinguishing scores to integers, and requires that the most likely distinguishing score corresponds to the smallest integer. While in principle any arbitrary method can be used to convert scores to integers with the desired properties (as in this

[4] For the ease of explanation we omit in the reminder the (log) and just use the term likelihoods. Previous works such as [2,14,17] showed that it is possible to 'convert' various side channel attack outputs to probabilities. Other papers [12,10,4] examine converting probabilities to integers.

example), for the remainder of this work we consider the mapping proposed by Martin *et al.*: a distinguishing value $D_{j,i}$ is mapped to a weight $W_{j,i}$ via $W_{j,i} = \lfloor 2^p \cdot D_{j,i} \rfloor$, for a chosen precision parameter p. This is called the “map to weight” conversion.[5]

This results in matrix of integer weights $\mathbf{W}$, which for our running example is as follows:

$$\mathbf{W} = \begin{pmatrix} 1 & 3 \\ 2 & 1 \\ 3 & 2 \end{pmatrix}$$

In our example the likelihood of the target key is $2/11 + 2/11 = 4/11$. The weight of the target key is $3 + 3 = 6$. All other combinations have higher likelihoods (or equivalently smaller weights). Thus as there are 9 keys overall, 8 keys are more likely than our target secret key.

2.3 Definitions

Given the weights (or scores), it is possible to order (full) keys based on their overall weights (likelihoods) as the scores are additive. Thus the definition of the rank of a (target) key can be given in a natural way (either using weights or likelihoods). For simplicity, we now only give the definition based on weights. Using weights, the rank of a target key is informally defined as the number of keys that are more likely (have smaller weight) than the given (target) key.

Definition 1 (Key Rank (weight based)). *Given an $n \times m$ matrix $\mathbf{W}$ and target key $\mathbf{s}$, the rank of the key $\mathbf{s}$ is defined as the number of keys $\mathbf{k}$ with a weight smaller than the weight of $\mathbf{s}$. Formally:*

$$\mathsf{rank}_{\mathbf{s}}(\mathbf{w}) = |\{\mathbf{k} = (k^1, \ldots, k^m) : \sum_{i=1}^{m} W_{k^i,i} < \sum_{i=1}^{m} W_{s^i,i}\}|$$

In the context of an attack, where an adversary has access to a weight matrix but does not know the target key $\mathbf{s}$, the adversary will want to enumerate (and test) keys with respect to their likelihood as given by the weight matrix and some set budget B. We hence define key enumeration with respect to a weight matrix and a budget.

Definition 2 (Key Enumeration (weight based)). *Given an $n \times m$ weight matrix $\mathbf{W}$ and $B \in \mathbb{Z}$, output the B keys with the lowest weights (breaking ties arbitrarily).*

This definition pays no attention to the order in which the B most likely keys are returned. Optimal key enumeration would output the B most likely keys $\mathbf{k}_1, \ldots, \mathbf{k}_B$ in the order of their weights.

[5] If the initial scores have the largest value most likely, the map to weight function will have to account for this first.

3 Ranking and Enumeration Approaches

Our work touches both on mathematical, as well as algorithmic, aspects of two competing approaches to key ranking and enumeration. To aid readability we now recap their working principle.

3.1 Path Counting Algorithm of Martin *et al.* [12]

The Rank Algorithm Based on Path Counting. Intuitively the algorithm works by constructing a graph with $m \cdot W_2 + 2$ nodes, where W_2 is the weight of the key $\mathbf{s}$ to be ranked. Each of the m rows in the graph corresponds to a subkey and the columns W_2 correspond to the weight of a partially constructed key. If there is a path from the initial node to the accept node, this corresponds to a valid key with a weight less than the secret key $\mathbf{s}$. The algorithm then calculates the rank of the keys by counting the number of paths between the start node and the accept node.

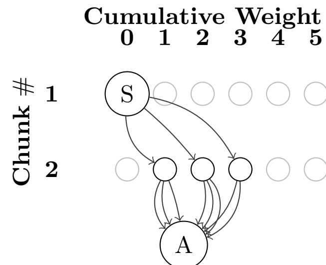

Fig. 1. The graph for our running example. Paths not contributing to the rank are excluded for clarity.

The graph for the running example can be seen in Fig. 1. The number of paths from the initial node S to the accept node A is exactly the rank of our secret key. The formal description of the algorithm is given in Alg. 1, which for the sake of flow is placed in App. A.

Recently an elegant mathematical description of the algorithms was provided in [11], which we give below. The matrix elements $b_{i,w}$ contain the number of paths from the corresponding vertex (in the graph) to the accept node. Consequently, the element $b_{1,0}$ then corresponds to the number of all paths to the accept node in the graph, which in turn gives the rank of the target key.

$$b_{i,w} := \sum_{j=1}^{n} b_{i+1,w+W_{j,i}} \text{ for } i < m \tag{1}$$

$$b_{m,w} := \sum_{j=1}^{n} \mathbf{1}\{W_{j,i} < W - w\} \tag{2}$$

where $\mathbf{1}(\cdot)$ returns 1 if the expression evaluates to true and 0 otherwise. This expression can be adapted to account for lower and upper weight bounds (W_1 and W_2) as detailed in [11].

Path Counting Based Enumeration Algorithms. Several variations of key enumeration algorithms based on the path based ranking idea can be found in the existing literature. In the original paper [12] the algorithm constructs partial keys, and passes them through the graph. At the end of the algorithm, $\mathbf{O}[0]$ contains the set of keys with weight between W_1 and W_2. Intuitively; for the final subkey, if the weight is within the correct range then the subkey value is added to the set of partially constructed keys. For the remaining subkeys the correct weight is looked up in $\mathbf{O}$ and the subkey value is appended to every partial key in the set.

The time complexity of this algorithm is $\mathcal{O}(m^2 \cdot n \cdot W_2 \cdot B \cdot \log n)$, where B is the number of keys with weight at between W_1 and W_2.

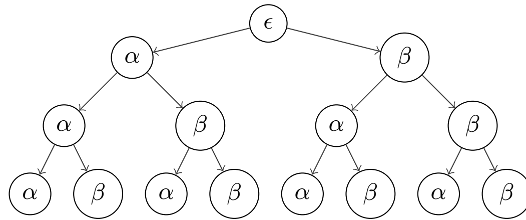

Fig. 2. The key tree for all possible three character keys containing 'α' or 'β' [12].

*Forest Enumeration (*FOREST*) [12,10].* In the same paper [12] the authors commented on the fact that if many keys are being enumerated, then there will be a lot of redundancy. For example if all keys with $k^1 = \alpha$ are enumerated, then the same initial key byte ($k^1 = \alpha$) would be stored 2^{120} times. Consequently one can improve memory complexity by storing the keys in a tree structure (with each level corresponding a subkey), instead; see Fig. 2 for an example and Alg. 2 for the formal description.

Another advantage comes with a reduction of the time complexity. This is because a subkey does not need to be "added" to all possible partial keys seen

so far (this "adding" would be linear in the number of partial keys) but just has to be added as the root of forest (turning it into a tree), which takes constant time.

The FOREST algorithm has thus a time complexity of $\mathcal{O}(m \cdot n \cdot W_2 \cdot \log n + m \cdot B \cdot \log n)$.

*Single Key Enumeration (*SINGLEKEY*) [11].* Recently a variation of the enumeration algorithm was given that allows a quantum speed up [11]. Unlike the previous algorithms, the memory complexity of this new version does not depend on the number of keys to be enumerated. The algorithm first computes the key rank, however it keeps the entire matrix $\mathbf{b}$ in memory (instead of just keeping the most recent two rows, as per Alg. 1). Hence the algorithm takes as input a 'key number' and uses it to "walk down" the graph to find that particular key. For instance, consider the graph in Fig. 2: a key is output by starting at the initial node S and following a path to the accept node A. Since each edge in the graph corresponds to an assignment to a subkey, the walk corresponds to a valid key assignment. Using the information stored in the rank graph, and the implicit ordering of subkey values, a path can be chosen in consistent manner so that no keys get missed and no keys get repeated. This process is repeated to enumerate multiple keys. The formal description is given in Alg. 3.

This algorithm has a time complexity of $\mathcal{O}(m^2 \cdot n \cdot W_2 \cdot \log n + B \cdot m^2 \cdot n \cdot \log n)$. This is asymptotically slightly worse than FOREST, however, it offers better parallelisation because it can parallelise over the number of keys B, instead of the total weight W_2. We will return to this aspect in Section 6. Another advantage is that its memory does not depend on the number of keys being enumerated, unlike all other algorithms discussed in this work.

3.2 Convolution Based Algorithm of Glowacz *et al.* [5]

The Rank Algorithm Based on Convolution. This rank algorithm begins by creating a histogram $\mathbf{H}_i$ per subkey i using $\mathbf{D}^i$. The number of bins β is a user controlled parameter. These histograms can be used to calculate the subkey rank. For example, if for subkey i the value is in bin y, then the subkey rank is given by $\sum_{l=y}^{\beta} H_{i,l}$. The algorithm then uses the following fact. If $\mathbf{H}_1, \mathbf{H}_2$ are the histograms for sets S_1, S_2 respectively then $\mathbf{H} = \mathsf{conv}(\mathbf{H}_1, \mathbf{H}_2)$ is the histogram for $S = \{s_1 + s_2 : s_1 \in S_1, s_2 \in S_2\}$. Thus repeatedly convolving in the subkey histograms, gives a histogram on the entire key space and summing the counts up to the bin containing $\mathbf{s}$ will give the rank of $\mathbf{s}$. Note that, given the bin numbers for each of the subkeys in $\mathbf{s}$, it is easy to compute the bin containing $\mathbf{s}$.

Mathematically this results in a recursion that can be formalised as given below. The element $\mathbf{c}_1$ then corresponds to the "final" histogram, from which the rank can be derived as $r \leftarrow \sum_{l=\mathsf{bin}(\mathbf{s})}^{m\cdot(\beta-1)+1} c_{1,l}$ (where $c_{1,l}$ refers to the elements of the l-th bin in the histogram $\mathbf{c}_1$.

$$\mathbf{c}_i := \mathsf{conv}(\mathbf{c}_{i+1}, \mathbf{H}_i) \text{ for } 1 \leq i < m \tag{3}$$

$$\mathbf{c}_m := \mathbf{H}_m \tag{4}$$

The full algorithm is given in Alg 4.

To continue with our example, Fig. 3 shows the two initial histograms for $\mathbf{D}^1$ and $\mathbf{D}^2$ respectively. The secret key $(3, 1)$ would be located in the bin with label 2, thus summing over the bins from label 2 gives correct rank $2 + 3 + 2 + 1 = 8$.

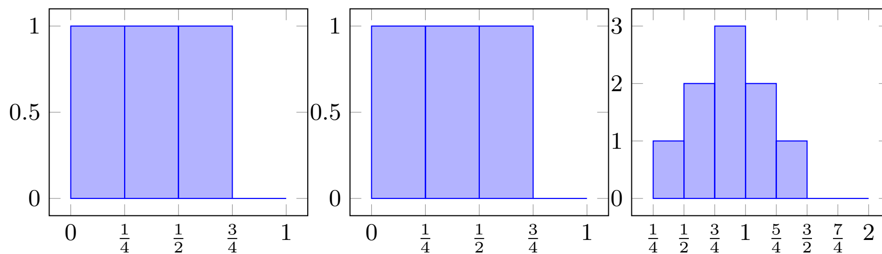

Fig. 3. Histograms for the running examples. From left to right: the histogram of the first subkey, the histogram for the second subkey, the histogram for the convolution.

Enumeration (Histogram) [15]. The algorithm first calculates the $\mathbf{c}_i$'s used by the histogram ranking algorithm. To enumerate keys the algorithm computes the keys in a recursive manner. Given a key of length m and $\mathbf{c}_1$, to enumerate keys of weight w, for each possible weight of subkey x, combine all subkeys of weight x from $\mathbf{H}_1$ (this information is easily tracked), with the partial keys output from the recursive call using length $m-1$, weight $w-x$ and histogram $\mathbf{c}_2$.

The time complexity of this algorithm depends on the distribution of keys within the histograms. As such a distribution is not known, and there are no obvious assumptions that can be made about it, the only way to assess the performance of this algorithm is via experiments.

4 Mathematical Equivalence of Ranking Approaches

In this section we show that the path counting approach and the histogram convolution approach are mathematically equivalent: this means that rank $r = b_{1,0} = \sum_{l=\mathsf{bin}(\mathbf{s})}^{m\cdot(\beta-1)+1} c_{1,l}$.

We start with equations for the histogram convolution. Recall that $\mathbf{H}_i$ denotes the histogram of the distinguishing vector $\mathbf{D}^i$, and we refer to the w-th bin of a histogram $\mathbf{H}$ via the notation $\mathbf{H}_w$ or $\mathbf{H}_{i,w}$ (if we index into the i-th

histogram as well). The convolution of two histograms $\mathbf{H} = \mathsf{conv}(\mathbf{H}_1, \mathbf{H}_2)$ is defined as $H_w = \sum_{i=0}^{w} H_{1,i} \cdot H_{2,w-i}$, which implies that all considered histograms have linearly spaced bins.

$$\mathbf{c}_i := \mathsf{conv}(\mathbf{c}_{i+1}, \mathbf{H}_i) \text{ for } 1 \leq i < m \tag{5}$$

$$\mathbf{c}_m := \mathbf{H}_m \tag{6}$$

4.1 Binning Equals Integer Conversion

Let α be the spacing of the bins. In a histogram, the value $D_{j,i}$ will hence be located in the bin $\lfloor \frac{D_{j,i}}{\alpha} \rfloor$.The value of α is determined by the number of bins β, which is a user supplied parameter, i.e. $\alpha = \frac{1}{\beta}$ (since the bins are equally sized). We set $\beta = 2^p$, where p is the precision parameter that is used in the "map to weight" float-to-integer conversion used prior to the path counting algorithm. Consequently, we get that the value $D_{j,i}$ is located in bin $\lfloor \beta \cdot D_{j,i} \rfloor$.

Evidently this results in precisely the "map to weight" conversion that is utilised to map values $D_{j,i}$ to integer weight values $W_{j,i}$ as given by [12].[6]

4.2 Base Case

To show that $b_{1,0} = \sum_{l=\mathsf{bin}(\mathbf{s})}^{m\cdot(\beta-1)+1} c_{1,l}$ we first consider the base case, which is $\mathbf{c}_m = \mathbf{H}_m$. We expand this expression by considering it for the w-th bin in the histogram:

$$\begin{aligned} c_{m,w} &= H_{m,w} \\ &= \sum_{j=1}^{n} \mathbf{1}\{D_{j,m} \text{ falls in bin } w\} \end{aligned}$$

We simply plug in the definition of a histogram for $H_{m,w}$, which is to count all elements that are located in bin w: $\sum_{j=1}^{n} \mathbf{1}\{D_{j,m} \text{ is located in bin } w\}$. Now using the fact that converting the distinguishing scores to integer values is equivalent to binning, this looks now like the base case of Martin *et al.*.

However, the array does $\mathbf{c}_m$ does not contain the same values as $\mathbf{b}_m$. The histogram $\mathbf{c}_m$ contains values which fall into a range, while $\mathbf{b}_m$ contains values which are less than a certain boundary. As an effect of this $\mathbf{b}_m$ is the cumulative sum of $\mathbf{c}_m$. This is why the histogram rank must return a sum over $\mathbf{c}_1$ when it completes, while path count rank can just return $b_{1,0}$.

[6] The conversion between largest being most likely and smallest being most likely, will simply result in a "flip" of the arrays that are stored.

4.3 Recurrence Relation

Next we consider the recurrence relation $\mathbf{c}_i = \mathsf{conv}(\mathbf{c}_{i+1}, \mathbf{H}_i)$. Like before, we consider the w-th bin:

$$\begin{aligned}
c_{i,w} &= \mathsf{conv}(c_{i+1}, \mathbf{H}_i)_w \\
&= \sum_{l=0}^{w} H_{i,l} \cdot c_{i+1,w-l} \\
&= \sum_{l=0}^{w} (\sum_{j=1}^{n} \mathbf{1}\{D_{j,i} \text{ falls in bin } l\}) \cdot c_{i+1,w-l} \\
&= \sum_{j=1}^{n} c_{i+1,w-x_{j,i}} \\
&= \sum_{j=1}^{n} c_{i+1,w-W_{j,i}}
\end{aligned}$$

We expand the convolution function using its definition. We then plug in the definition of a histogram, and finally rearrange the terms. In the second but last step we denote by $x_{j,i}$ the histogram bin $D_{j,i}$ is located in after the convolution. We have shown previously that the bin $x_{j,i}$ corresponds to the $W_{j,i}$, thus, the final step follows. What remains to consider is that we subtract $W_{j,i}$ here rather than adding it as in Martin *et al.*'s recurrence relation. Recall that in the "map to weight" conversion larger scores are mapped to smaller weights (and hence the weight based definition of key rank counts keys with weight smaller than the target weight whereas the likelihood based definition counts keys with scores larger than the target weight). Thus here we subtract $W_{j,i}$, whereas in the recurrence relation in $b_{i,w}$ we add $W_{j,i}$. Finally then we have indeed that $r = b_{1,0} = \sum_{l=\mathsf{bin}(\mathbf{s})}^{m\cdot(\beta-1)+1} c_{1,l}$.

We have shown that the two sets of equations for path counting and histogram convolution counting are equivalent and the algorithms computing over them, for both rank and enumeration are equivalent for suitable input parameters. In particular the number of bins and the spacing of the bins in the histogram algorithm, are in direct correspondence with each other. Therefore, due to the correctness of each algorithm, they both compute the same metric to the same accuracy. Thus, the decision of which algorithm to favour over another, comes down to its particular use case, and the differences in the algorithmic representations. We spend the remainder of the paper exploring this space.

5 Experimental Analysis

Although convolution based ranking and path counting based ranking are mathematically equivalent assuming their discretisation parameter is chosen correspondingly, the algorithms that they result in are different. Thus their related

key enumeration algorithms are different as well, and come with different algorithmic complexities. Among the different variations of the path counting enumeration algorithms, the Forest algorithm is the most desirable both in terms of time and space complexity when it comes to "realistic" search efforts. Only in the case of an extremely deep key search, the single key enumeration algorithm would potentially be a better choice because of its its capability to parallelise based on the number of keys to enumerate rather than the precision parameter.

As we argued before it is impossible to give sound bounds for the convolution based algorithm because its performance depends on the distribution of items in bins. We hence now set up a concrete experiment, based on the best available implementations of two respective approaches. Our comparison is comprised of two parts. First we provide some concrete experiments on a single core across different values of their discretisation parameter in this section. These experiments enable us to conclude on their sequential performance depending on that parameter. Thereafter, in the next section, we consider the impact of this parameter on larger enumeration efforts, which will require the use of many cores in parallel.

5.1 Sequential Performance

As described in Section 2, both algorithms effectively discretise distinguishing scores: Forest uses a score-to-integer-values "weight conversion" method prior to execution and Histogram uses convolution, after which the subkeys assigned to a particular bin are considered equally likely to be correct candidates. The level of precision retained in the score conversion process and the quantity of histogram bins used directly impact algorithm run-time and memory usage. Both algorithms are also impacted by the number of distinguishing vectors and the number of subkey candidates per distinguishing vector.

Experimental Setup. The experiments outlined in this section were timed using a workstation equipped with a Intel Xeon E5-1650v2 CPU and 32 GiB of 1600MHz PC3-12800 DDR3 RAM. All code was compiled using version 4.8.4 of GCC with level 3 optimisations enabled.

The experimental strategy consisted of simulating DPA attacks on a 128-bit AES key, using 16 independent attacks on the 8-bit SubBytes output for each repeated experiment. Each set of synthetic trace data was simulated under the standard DPA model as described in [8], using fresh randomness to generate simulated leakage measurements. We chose a low signal to noise ratio which ensured that the correct key was ranked between 2^{40} and 2^{70}. This ensures the creation of realistic distinguishing vectors, which are important to realistically assess the performance of the enumeration alogrithms. For each experiment, we recorded the time taken to generate the first $2^{11}, 2^{12}, \ldots, 2^{39}$ most likely key candidates (producing 29 measurements in total).

We performed this process for $p = 11$ through $p = 16$ bits of precision (in the case of Histogram this equates to using 2^p bins for each initial histogram

at a precision level p bits). This range of precision covers a degree of parallelism most suited to a well-resourced adversary such as a nation state or an individual organisation with access to a super-computer or a botnet, and who desires an enumeration capability that can be used search for very deep keys.

Configurations of Algorithms. For both algorithms we timed the enumeration of keys but not the verification. Verification typically consists of the encryption or decryption of one or more known pairs of plaintext and ciphertext using a key candidate, and thus is a fixed cost.

FOREST *Configuration.* We used the open-source implementation of FOREST provided by the authors of [7]. Execution time was recorded from the moment the distinguishing vectors were converted into integer weights, up until every key targeted was fully generated. The range of weights provided to the algorithm was taken to be the minimum key weight observed up to and including the first weight at which at least the targeted number of keys would be enumerated.

HISTOGRAM *Configuration.* We used the open-source implementation of HISTOGRAM provided by the authors of [15]. Execution time was recorded from the moment distinguishing vectors were converted into histograms up until every key within the relevant bins was generated. The bin indexes selected for enumeration were calculated using the selection method provided by the open-source implementation.

Pre-processing. The authors of HISTOGRAM note that it is possible to pre-process distinguishing scores by multiplying through pairs of distinguishing vectors [15]. This method is essentially the approach described in the 2014 work of [13]. For example, given 16 distinguishing vectors each associated with 256 subkey candidates, one can multiply each consecutive pair together, producing 8 distinguishing vectors each containing 65536 subkey candidates. We will define this as a single "step" of pre-processing.

In [15], it is demonstrated that is pre-processing provides a significant performance increase to the HISTOGRAM algorithm. In practice, the pre-processing method can be applied to any enumeration algorithm. The pre-processing can be repeatedly applied at a significant memory cost: taking the previous example, the 8 distinguishing vectors could again be pairwise multiplied at a cost of having to store the scores corresponding to 2^{32} subkey candidates in memory at a time.

In this work, we compare the FOREST algorithm with no pre-processing applied against an implementation of HISTOGRAM with both no pre-processing and single level of pre-processing applied.

5.2 Results

Figures 4 and 5 illustrate the results of our experiments. All time measurements are taken to the logarithm base 10. Figure 4 shows the performance of FOREST

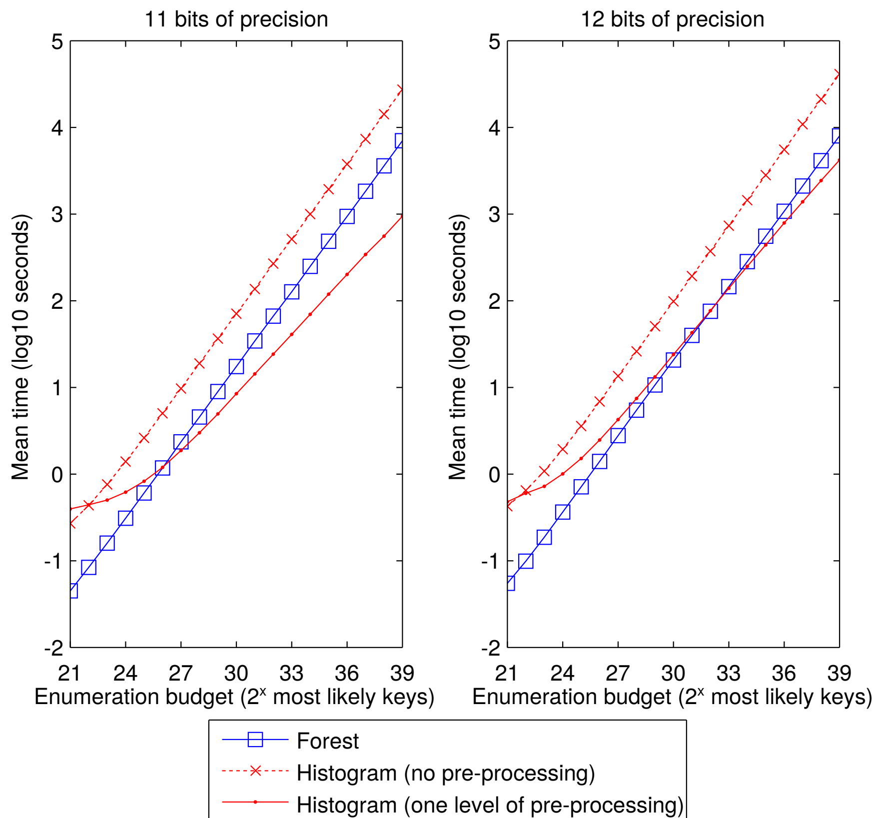

Fig. 4. The mean running time of the FOREST algorithm and the HISTOGRAM algorithm configured with and without pre-processing, for a variety of enumeration budgets and at 11 and 12 bits of precision.

and HISTOGRAM when the level of precision is at 11 and 12 bits (2048 and 4096 histogram bins), and include measurements when HISTOGRAM uses distinguishing vectors that have no pre-processing applied (16 distinguishing vectors, each 8-bits in size) and a single pre-processing step applied (8 distinguishing vectors, each 16-bits in size).

Above 12 bits of precision the performance of the no pre-processing variant of HISTOGRAM was such that it was impractical to continue running experiments using it. Figure 5 therefore contains measurements for HISTOGRAM using a single pre-processing step only. It covers experiments run at 13, 14, 15 and 16 bits of precision (8192, 16384, 32768 and 65536 histogram bins).

Impact of Pre-processing. Figure 4 confirms the results of the HISTOGRAM authors, finding that pre-processing is very impactful to the run-time of the HIS-

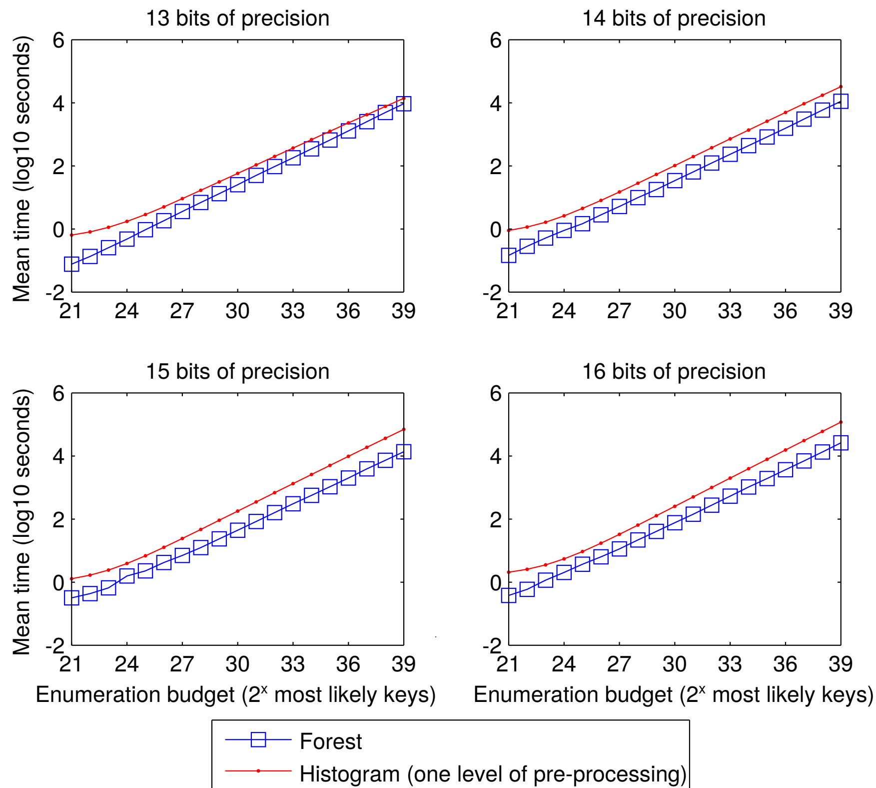

Fig. 5. The mean running time of the Forest algorithm and the Histogram algorithm configured with a single step of pre-processing, for a variety of enumeration budgets and at 13 to 16 bits of precision.

togram algorithm. This allows it to be significantly faster than Forest at our lowest level of precision, and eventually demonstrated an advantage at the second-lowest level of precision. Whilst the log-scale graphs are the most practical way to visualise the algorithm performance, they do not give an intuitive idea of scale: when the enumeration budget was 2^{39}, at 11 bits of precision Histogram with pre-processing was on average approximately 7 times faster than Forest. Whether Forest would benefit equally as well to the pre-processing step is an interesting question for future research.

An additional consideration could be whether two steps of pre-processing provide equivalent performance gains. Assuming a 128-bit AES key and distinguishing scores stored as double-precision values, moving to two steps of pre-processing would require the adversary to have at least 256 GiB of RAM ($4 \cdot 64$ GiB) available to each parallel execution unit.

Impact of Precision. However, as can be observed in Figure 5, above 12 bits of precision the run-time of HISTOGRAM degraded to the point that FOREST was significantly faster even when the pre-processing was applied. The performance gap widens as precision increases, indicating that if a precision of greater than 12 bits is required, FOREST is highly likely to be the most suitable choice. For smaller-scale efforts, such as those performed by individuals constrained by resources and time, HISTOGRAM configured for low levels of precision may be the most expedient method to test the first 2^{40} keys.

Minimum Precision Requirements. The previous works by [5,10,12] consistently found that a precision of at least 12 bits was required for to ensure that "stable" results were observed over repeat experiments. We can guess as to the underlying cause: the distinguishing vectors in their experiments were produced by attacks targeting 8-bit subkeys. An 8-bit vector may hold 2^8 distinct values and so, at the very minimum, 8 bits of precision are required to assign each subkey candidate a unique value. The attacks used in the experiments aimed to recover a target key that consists of 16 subkeys. Given that the final score for a candidate is the *sum* of its respective subkey scores, and that each addition of two values implies the need for one extra bit to represent the result, at least $2^4 * 2^8 = 2^{12}$ bits of precision are required to maintain the ability to assign a unique value for each element resulting from a cumulative sum.

General Observations. The performance of both algorithms seem to behave consistently. This may be useful for an adversary when attempting to calibrate their effort: it may be possible to derive parameters that allow a reasonably accurate prediction of the run-time of a workload. This would allow the adversary to fine-tune the choice of precision and number of compute resources to enumerate to a pre-defined depth in a pre-defined period of time.

One interesting future research questions is to understand whether the behavior observed in Figures 4 and 5 continues when enumerating extremely deep keys (for instance, below a depth of 2^{50} or 2^{60}).

6 Considering Parallelism

Before looking in more detail at the respective algorithms we briefly reflect on the need to balance effort in case of any parallel enumeration effort. In this respect we note that it is possible for the adversary to evenly distribute the workload across multiple hardware resources for both algorithms. This can be done using an (inexpensive) key ranking algorithm: in the case of HISTOGRAM the adversary would ascertain how many keys are assigned to each bin in the final convolved histogram, and in the case of FOREST the adversary would ascertain how many keys are assigned to each unique weight value.

Both FOREST and HISTOGRAM are most intuitively parallelised along their "discretisation" parameter. HISTOGRAM can be parallelised along the number of bins in the final convolved histogram (corresponding to parallel invocations of

the "Decompose_bin" algorithm described in [15]). FOREST can be parallelised along each unique weight value: the adversary can choose to sequentially process the keys associated with a unique weight or within a continuous range of weights.

6.1 Exact Parallelisation Potential

For a given attack configuration, the number of parallel execution units that can simultaneously execute the Decompose_bin algorithm is upper-bounded by $\beta \cdot m - m + 1$. The number of parallel execution units in the FOREST is bounded by the maximum observed integer weight associated with a key. If the maximum weight is W, then the adversary can execute at most W parallel enumeration instances.

The number of histogram bins used and the level of precision retained in the integer weight conversion process thus effectively act as tunable *precision* and parallelism parameter: the lower the number of bins or conversion precision, the less resolution available in the final ordering of keys and the fewer parallel invocations of the respective algorithm can be made. Recall that we can consider precision in terms of a number of bits: p-bits of precision is equivalent to using 2^p histogram bins or converting scores to integer weights such that the maximum value associated with any subkey candidate is 2^p. Given a fixed level of precision, the theoretical parallelism potential of each algorithm is almost identical: given p-bits of precision, FOREST can be run with, at most, $m \cdot 2^p$ parallel invocations. HISTOGRAM can be run with, at most, $m \cdot 2^p - m + 1$ parallel invocations.

The algorithm SINGLEKEY is not limited by the precision parameter and can parallelise up to the number of keys that it wishes to enumerate (one key per core). We leave it as an interesting research question, as to when SINGLEKEY becomes more desirable.

6.2 Trading Off Sequential Performance For Parallelism

The sequential performance of both algorithms deteriorates as the precision parameter increases. The natural assumption in brute-force cryptanalysis problems is that the more computational resources the adversary can deploy in parallel, the faster they are likely to achieve a breakthrough. Whether this remains true in all instantiations of an enumeration problem is unclear. The work of Poussier *et al.* proposes that an adversary who is willing to enumerate a very large amount of keys (for instance, beyond 2^{64}), might be better served by *reducing* the number of bins in each histogram – the argument being that it may be more efficient to maximise the occupancy of *smaller* quantities of hardware by providing each unit *larger* quantities of factorised keys.[7] Whether this is indeed the case requires a careful analysis, including the efficiency of the bin decomposition algorithm (in

[7] A set of factorised keys can be converted into a set of keys by taking the cross product between all subkey sets. For example given the key factorisation $([1, 2], [3, 4])$ which will all have the same weight in each subkey, this represents the four keys $(1, 3), (1, 4), (2, 3), (2, 4)$.

the case of HISTOGRAM), the efficiency of the forest tree traversal (in the case of FOREST), whether specialised hardware is available, and memory requirements.

A complication arises as to how close an adversary wishes to be to a pre-selected number of keys enumerated. Taking the proposal of Poussier *et al.*, let us consider an adversary attempting to recover a 128-bit AES key by enumerating the output of a side-channel attack targeting each of the 16 8-bit SubBytes outputs, with HISTOGRAM configured to use 256 bins per histogram. Using these parameters, the final histogram will contain 4081 ($256 \cdot 16 - 15$), or just under 2^{12} bins. Therefore, the average number of key candidates associated with each bin is approximately 2^{116}.

At first glance, this seems to be a disaster for the adversary. Fortunately, assuming a 'good' side-channel attack, the bins associated with the most likely key candidates will contain far fewer candidates than the bins associated with the less likely candidates. However, some informal reasoning demonstrates how the probability of the adversary getting 'unlucky' has increased: it is reasonable to assume the expected position of the correct key amongst its equally-likely candidates is in the middle of its bin, and so if the size of a bin is extremely large, the chance of the adversary having to enumerate a significant number of unnecessary keys increases. Experiments in other works indicate that the expected number of keys per bin increases *exponentially* as the rank of the correct key increases, and so this consideration becomes more important as the computational budget of an adversary increases [7].

7 Conclusions and Future Research

Over the past few years two approaches for rank computation and key enumeration have been proposed and researched. These were believed to be distinct from each other. We show in this contribution that they are mathematically equivalent, i.e. they both compute the exact same rank when choosing their discretisation parameter correspondingly. Thus they can both be equally accurate (which matters for key ranking). Knowing that they are mathematically equivalent, we then turn our focus on their algorithmic representations, which are different. We compare their enumeration versions fairly (using the same platform, the same language and compiler) via their performance on different levels of the discretisation parameter.

Our practical experiments indicate that HISTOGRAM performs best for low discretisation, and FOREST wins for higher parameters. We explain that a minimum of 12 bits should be allowed for accurate rankings, and any more bits are desirable for large scale enumeration efforts. Thus the FOREST algorithm should be the preferred choice if large quantities of parallelism are required.

An important direction for future research is to identify, given a fixed amount of computational resources and time, how best to distribute the enumeration workload. A solution to this will help identify the ideal level of precision used in an enumeration algorithm. Furthermore, a particularly useful research direction would be to consider how an evaluator could take the estimated rank of a side-

channel attack, a definition of a class of adversary – for instance, a group with access to a botnet or a compute cloud – and to be able to derive a reasonable estimate for the the total duration and cost of enumerating that key, *without* doing the complete enumeration task. Whilst in our paper we observe consistent results for relatively small search efforts, which could be seen as a stepping stone in this direction, the best parallelisation strategy to tackle large scale search efforts remains an open question.

Acknowledgements and Disclaimer This work was in part supported by EPSRC via grant EP/N011635/1 (LADA). No research data was created for this paper. The final publication will be available at link.springer.com in the proceedings of CT-RSA 2018.

References

1. Bernstein, D.J., Lange, T., van Vredendaal, C.: Tighter, faster, simpler side-channel security evaluations beyond computing power. IACR Cryptology ePrint Archive 2015, 221 (2015), http://eprint.iacr.org/2015/221
2. Bernstein, D.J., Lange, T., van Vredendaal, C.: Tighter, faster, simpler side-channel security evaluations beyond computing power. Cryptology ePrint Archive, Report 2015/221 (2015), http://eprint.iacr.org/2015/221
3. Bogdanov, A., Kizhvatov, I., Manzoor, K., Tischhauser, E., Witteman, M.: Fast and memory-efficient key recovery in side-channel attacks. IACR Cryptology ePrint Archive 2015, 795 (2015)
4. Bogdanov, A., Kizhvatov, I., Manzoor, K., Tischhauser, E., Witteman, M.: Fast and memory-efficient key recovery in side-channel attacks. In: Dunkelman, O., Keliher, L. (eds.) SAC 2015. LNCS, vol. 9566, pp. 310–327. Springer, Heidelberg (Aug 2016)
5. Glowacz, C., Grosso, V., Poussier, R., Schüth, J., Standaert, F.X.: Simpler and more efficient rank estimation for side-channel security assessment. In: Leander, G. (ed.) FSE 2015. LNCS, vol. 9054, pp. 117–129. Springer, Heidelberg (Mar 2015)
6. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (Aug 1999)
7. Longo, J., Martin, D.P., Mather, L., Oswald, E., Sach, B., Stam, M.: How low can you go? Using side-channel data to enhance brute-force key recovery. Cryptology ePrint Archive, Report 2016/609 (2016), http://eprint.iacr.org/2016/609
8. Mangard, S., Oswald, E., Standaert, F.X.: One for All – All for One: Unifying Standard DPA Attacks. IET Information Security 5(2), 100–110 (2011), http://eprint.iacr.org/2009/449
9. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer (2007)
10. Martin, D.P., Mather, L., Oswald, E., Stam, M.: Characterisation and estimation of the key rank distribution in the context of side channel evaluations. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 548–572. Springer, Heidelberg (Dec 2016)
11. Martin, D.P., Montanaro, A., Oswald, E., Shepherd, D.: Quantum key search with side channel advice (2017)

12. Martin, D.P., O'Connell, J.F., Oswald, E., Stam, M.: Counting keys in parallel after a side channel attack. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015, Part II. LNCS, vol. 9453, pp. 313–337. Springer, Heidelberg (Nov / Dec 2015)
13. Mather, L., Oswald, E., Whitnall, C.: Multi-target DPA attacks: Pushing DPA beyond the limits of a desktop computer. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part I. LNCS, vol. 8873, pp. 243–261. Springer, Heidelberg (Dec 2014)
14. Pan, J., van Woudenberg, J.G.J., den Hartog, J., Witteman, M.F.: Improving DPA by peak distribution analysis. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 241–261. Springer, Heidelberg (Aug 2011)
15. Poussier, R., Standaert, F.X., Grosso, V.: Simple key enumeration (and rank estimation) using histograms: An integrated approach. In: Gierlichs, B., Poschmann, A.Y. (eds.) CHES 2016. LNCS, vol. 9813, pp. 61–81. Springer, Heidelberg (Aug 2016)
16. Veyrat-Charvillon, N., Gérard, B., Renauld, M., Standaert, F.X.: An optimal key enumeration algorithm and its application to side-channel attacks. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 390–406. Springer, Heidelberg (Aug 2013)
17. Veyrat-Charvillon, N., Gérard, B., Standaert, F.X.: Security evaluations beyond computing power. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 126–141. Springer, Heidelberg (May 2013)
18. Ye, X., Eisenbarth, T., Martin, W.: Bounded, yet Sufficient? How to Determine Whether Limited Side Channel Information Enables Key Recovery. In: CARDIS 2014. LNCS, vol. 7707. Springer (2014)

A Algorithms

A.1 The MOOS Ranking Algorithm

The array begins in the row corresponding to the last key chunk, and works across the weights. For each subkey value, if adding the weight of this value to the current weight is between W_1 and W_2 then the algorithm sets the count to 1, else it is set to zero. Working backwards over the other subkeys, the algorithm sums the counts for assigning each possible subkey value. The function RC looks up the corresponding position in the array for the subkey - this is just the weight or either reject or accept depending on the weight boundaries and if it is the final subkey. The copying of $\mathbf{K}$ into $\mathbf{O}$ is so that the previous subkey's counts are available when the current counts are calculated. For ranking a key $W_1 = 0$ but can be set to other values to count the number of keys between two given keys. This can be useful for parallelising enumeration.

A.2 The Forest Enumeration Algorithm

The forest enumeration algorithm works in a similar manner to the key ranking algorithm described above. However, instead of looking up partial counts in the array $\mathbf{O}$, partial key representations are stored. To update the representations the forest of keys in take and the subkey value is applied to the head of the forest, to turn it into a tree.

Algorithm 1 The Martin *et al.* algorithm [12] (MOOS) plus improvements as detailed in [10] (MOOS+).

```
Algorithm MOOS(m, n, W_1, W_2, W):
O[Accept] ← 1
O[Reject] ← 0
for i from m down to 1 do
   for w from 0 up to W_2 − 1 do
      K[w] ← 0
      for j from n down to 1 do
         K[w] ← K[w] + O[RC(j, w, i, W_1, W_2, W)]
      end for
   end for
   O ← K
end for
return O[0]

Algorithm RC(j, w, i, W_1, W_2, W):
if w + W_{i,j} > W_2 then
   return Reject
else if j = m − 1 then
   if w + W_{j,i} < W_1 then
      return Reject
   else
      return Accept
   end if
else
   return w + W_{j,i}
end if
```

A.3 The Single Key Enumeration Algorithm

Given an index of the key to output the algorithm works as follows. It looks at how many keys have an initial subkey value, if the required key is contained in this range then the algorithm assigns this value to the key and looks at the next subkey, else it increments the value being considered. This is repeated until the key has been constructed, at which point it is returned.

A.4 The Convolution Based Ranking Algorithm

The algorithm calculates the histogram $\mathbf{H}_i$ for each distinguishing vector $\mathbf{D}^i$. It then convolutes them all together. Based on the properties of convolution, the key rank is then simply the sum of bins from the bin representing the secret key in this final convolution.

A.5 The Convolution Based Enumeration Algorithm

The convolution algorithm works recursively where if a key of weight w is required all subkeys of weight x are paired with remaining partial keys of weight

Algorithm 2 The Martin *et al.* enumeration algorithm FOREST [12] plus improvements as detailed in [10]. Where makeTree(r, F) turns the forest F into a tree by appending r as a root. The constructKeys algorithm takes a forest of trees and traverses them to construct the keys.

Algorithm FOREST$(m, n, W_1, W_2, \mathbf{W})$:
for i **from** m **down to** 1 **do**
  for w **from** 0 **up to** $W_2 - 1$ **do**
    $\mathbf{K}[w] \leftarrow \{\}$
    for j **from** 1 **up to** n **do**
      if $RC(j, w, i, W_1, W_2, \mathbf{W}) = Accept$ **then**
        $\mathbf{K}[w] \leftarrow \mathbf{K}[w] \cup \{j\}$
      end if
      if $RC(j, w, i, W_1, W_2, \mathbf{W}) \neq Reject$ **then**
        $\mathbf{K}[w] \leftarrow \mathbf{K}[w] \cup$ makeTree$(j, \mathbf{O}[RC(j, w, i, W_1, W_2, \mathbf{W})])$
      end if
    end for
  end for
  $\mathbf{O} \leftarrow \mathbf{K}$
end for
return constructKeys$(\mathbf{O}[0])$

$w - x$. This is considered for all possible weights x. The remaining partial calls are generated using a recursive call to the algorithm.

Algorithm 3 The Martin *et al.* enumeration algorithm SingleKey [11]. It takes in the vector $\mathbf{b}$ as output by the rank algorithm and returns the rth key, for arbitrary ordering. To produce an enumeration algorithm to output multiple keys, this algorithm can just be run in a loop multiple times. See the original paper for more details.

```
Algorithm SingleKey(b, W, W_1, W_2, r):
if r > b_{1,0} then
  return ⊥
end if
k ← ε
w ← 0
for i = 1 up to m − 1 do
  for j = 1 up to n do
    if r ≤ b_{j+1, w+w_{j,i}} then
      k ← k||j
      w ← w + w_{j,i}
      break j
    end if
    r ← r − b_{j+1, w+w_{j,i}}
  end for
end for
for j = 1 up to n do
  if r ≤ 1{W_1 − w ≤ w_{m,j} < W_2 − w} then
    k ← k||j
    break
  end if
  r ← r − 1{W_1 − w ≤ w_{j,m} < W_2 − w}
end for
return k
```

Algorithm 4 The Glowacz *et al.* algorithm [5] (GGPSS).

```
Algorithm GGPSS(m, D):
c_0 ← toHist(D^m)
for i from m − 1 down to 1 do
  H_i ← toHist(D^i)
  c_i ← conv(H_i, c_{i+1})
end for
r ← Σ_{l=bin(s)}^{m·(β−1)+1} c_{1,l}
return r
```

Algorithm 5 The Poussier *et al.* enumeration algorithm HISTOGRAM [15]. csh is used in the recursive call and is initially set to 1, y is initially set as the bin to be decomposed and k is initially empty. $\mathsf{get}(\mathbf{H}, x)$ is used to get all items of weight x from histogram $\mathbf{H}$ and $\mathsf{size}(\mathbf{H})$ gets the size of histogram $\mathbf{H}$. The algorithm $\mathsf{processKey}(k)$ converts the factorised keys k into a list of keys. Enumeration simply decomposes bins in order, see the original paper for further details.

```
Algorithm HISTOGRAM(c, {H_i}_{i=1}^m, csh, y, k):
if csh = m − 1 then
    x ← H_m − 1
    while x ≥ 0 and x + size(H_{m−1} ≥ y do
        if H_{m,x} > 0 and H_{m−1,y−x} > 0 then
            k(m) ← get(H_m, x)
            k(m − 1) ← get(H_{m−1}, y − x)
            processKey(k)
        end if
        x ← x − 1
    end while
else
    x ← H_csh − 1
    while x ≥ 0 and x + size(c_{csh+1}) ≥ y do
        if H_{csh,x} > 0 and c_{csh+1,y−x} > 0 then
            k(csh) ← get(H_csh, x)
            HISTOGRAM(c, {H_i}_{i=1}^m, csh + 1, y − x, k)
        end if
        x ← x − 1
    end while
end if
```