Simple Key Enumeration (and Rank Estimation) Using Histograms: An Integrated Approach

Romain Poussier[1(✉)], François-Xavier Standaert[1], and Vincent Grosso[1,2]

[1] ICTEAM/ELEN/Crypto Group, Université catholique de Louvain, Louvain-la-Neuve, Belgium
{romain.poussier,fstandae}@uclouvain.be
[2] Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Bochum, Germany
vincent.grosso@ruhr-uni-bochum.de

Abstract. The main contribution of this paper, is a new key enumeration algorithm that combines the conceptual simplicity of the rank estimation algorithm of Glowacz et al. (from FSE 2015) and the parallelizability of the enumeration algorithm of Bogdanov et al. (SAC 2015) and Martin et al. (from ASIACRYPT 2015). Our new algorithm is based on histograms. It allows obtaining simple bounds on the (small) rounding errors that it introduces and leads to straightforward parallelization. We further show that it can minimize the bandwidth of distributed key testing by selecting parameters that maximize the factorization of the lists of key candidates produced by the enumeration, which can be highly beneficial, e.g. if these tests are performed by a hardware coprocessor. We also put forward that the conceptual simplicity of our algorithm translates into efficient implementations (that slightly improve the state-of-the-art). As an additional consolidating effort, we finally describe an open source implementation of this new enumeration algorithm, combined with the FSE 2015 rank estimation one, that we make available with the paper.

1 Introduction

Key enumeration and rank estimation algorithms have recently emerged as an important part of the security evaluation of cryptographic implementations, which allows post-processing the side-channel attack outcomes and determine the computational security of an implementation after some leakage has been observed. In this respect, key enumeration can be seen as an adversarial tool, since it allows testing key candidates without knowledge of the master key [4,8,10] (for example, it was an important ingredient of the best attack submitted to the DPA Contest v2 [5]). By contrast, rank estimation as an evaluation tool since it requires the knowledge of the master key. Its main advantage is that it allows efficiently gauging the security level of implementations for which enumeration is beyond reach (and therefore are not trivially insecure) [3,7,8,11,12].

Concretely, state-of-the-art solutions for key rank estimation are essentially sufficient to analyze any (symmetric) cryptographic primitive. Algorithms such

B. Gierlichs and A.Y. Poschmann (Eds.): CHES 2016, LNCS 9813, pp. 61–81, 2016.
DOI: 10.1007/978-3-662-53140-2_4

as [3,7,8] typically allow estimating the rank of a 128- or 256-bit key with an accuracy of less than one bit, within seconds of computation. By contrast, efficiency remained a concern for key enumeration algorithms for some time, in particular due to the inherently serial nature of the optimal algorithm of Veryat et al. [10]. This situation evolved with the recent (heuristic) work of Bogdanov et al. [4] and the more formal solution of Martin et al. [8]. In these papers, the authors exploit the useful observation that by relaxing (a little bit) the optimality requirements of enumeration algorithms (as one actually does in rank estimation), it is possible to significantly improve their efficiency, and to make them parallelizable. Since this relaxation is done by rounding the key (log) probabilities (or scores) output by a side-channel attack, it directly suggests to try adapting the histogram-based rank estimation algorithm from Glowacz et al. to the case of key enumeration based on similar principles.

In this paper, we follow this track, and describe a new enumeration algorithm based on histogram convolutions. As for rank estimation, using such simple tools brings conceptual simplicity as an important advantage. Interestingly, we show next that this simplicity also leads to several convenient features and natural optimizations of the enumeration problem. First, it directly leads to simple bounds on the rounding errors introduced by our histograms (hence on the additional workload needed to guarantee optimal enumeration up to a certain rank). Second, it allows straightforward parallelization between cores, since the workload of each core is directly available as the number of elements in each bin of our histograms. Third, it outputs the keys as factorized lists, such that by adequately tuning the enumeration parameters (i.e. the number of bins, essentially), we are able to use our enumeration algorithm for distributed key testing with minimum bandwidth (which is typically desirable if hardware/FPGA implementations are used). In this respect, our experiments show that the best strategy is not always to maximize the accuracy of the enumeration (especially when enumerating up to large key ranks). We note that such features could also be integrated to other recent enumeration algorithms (i.e. [8], and to some extent [4]). Yet, this would require some adaptations while it naturally comes for free in our histogram-based case. Eventually, the same observation essentially holds for the performances of our algorithm, which slightly improve the state-of-the-art.

In view of the consolidating nature of this work, an important additional contribution is an open source implementation of our key enumeration algorithm, combined with the histogram-based rank estimation algorithm of FSE 2015, that we make available with this paper in order to facilitate the dissemination of these tools for evaluation laboratories [1].

2 Background

2.1 Algorithms Inputs

Details on how a side-channel attack extracts information from leakage traces are not necessary to understand the following analysis. We only assume that for a n-bit master key k, an attacker recovers information on N_s subkeys $k_0, ..., k_{N_s-1}$ of

length $a = \frac{n}{N_s}$ bits (for simplicity, we assume that a divides n). The side-channel adversary uses the leakages corresponding to a set of q inputs $\mathcal{X}_q$ leading to a set of q leakages $\mathcal{L}_q$. As a result of the attack, he obtains N_s lists of 2^a probabilities $P_i = \Pr[k_i^* | \mathcal{X}_q, \mathcal{L}_q]$, where $i \in [0, N_s - 1]$ and k_i^* denotes a subkey candidate among the 2^a possible ones. TA (Template Attacks) and LR (Linear Regression)-based attacks directly output such probabilities. For other attacks such as DPA (Differential Power Analysis) or CPA (Correlation Power Analysis), one can use Bayesian extensions [10] or perform the enumeration directly based on the scores. Note that in this last case, the enumeration result will be correct with respect to the scores, but the corresponding side-channel attack is not guaranteed to be optimal [9]. For simplicity, our following analyses are based only on the optimal case where we enumerate based on probabilities. We leave the investigation of the overheads due to score-based enumeration as an interesting scope for further investigation. Eventually, the lists of probabilities are turned into lists of log probabilities, denoted as $LP_i = \log(P_i)$. This final step is used to get an additive relation between probabilities instead of a multiplicative one.

2.2 Preprocessing

Key enumeration (and rank estimation) algorithms generally benefit from the preprocessing which consists of merging m lists of probabilities P_i of size 2^a in order to generate a larger list $P'_i = \mathsf{merge}(P_0, P_1, \ldots, P_{m-1})$, such that P'_i contains the $2^{m \cdot a}$ product of probabilities of the lists $P_0, P_1, \ldots, P_{m-1}$. Taking again the previous notations where the n bits of master key are split in N_s subkeys of a bits, it allows to split them into $N'_s = N_s/m$ subkeys of $m \cdot a$ bits (or close to it when m does not divide N_s). We denote the preprocessing merging m lists as merge_m, with merge_1 meaning no merging. In the following, we assume that such a preprocessing is performed by default and therefore always use the notation N'_s for the number of subkeys.

2.3 Toolbox

We now introduce a couple of tools that we use to describe our algorithms, using the following notations: H will denote an histogram, N_b will denote a number of bins, b will denote a bin and x a bin index.

Linear histograms. The function $H = \mathsf{hist\_lin}(LP, N_b)$ creates a standard histogram from a list of (e.g.) log probabilities LP and N_b linearly-spaced bins. This is the same function as introduced in [9].

Convolution. This is the usual convolution algorithm which from two histograms H_1 and H_2 of sizes n_1 and n_2 computes $H_{1,2} = \mathsf{conv}(H_1, H_2)$ where $H_{1,2}[k] = \sum_{i=0}^{k} H_1[i] \times H_2[k-i]$. It is efficiently implemented with a FFT in time $\mathcal{O}(n \log n)$. In the rest of the paper we consider that the indexes start at 0.

Getting the size of a histogram. We defined by $\mathsf{size\_of}(H)$ the function that returns the number of bins of an histograms H.

Getting subkey candidates from a bin. We define $\mathcal{K} = \mathsf{get}(H, x)$ as a function that outputs the set of all subkeys contained in the bin of index x of an histogram H. Such a set can contain up to $2^{m \cdot a}$ elements depending on the merging value.

3 Enumeration Algorithm

In this section, we describe our new key enumeration algorithm. Since we join an open source code of this algorithm to the paper, our primary goal is to explain its main intuition. For this purpose, we combine a specification of the different enumeration steps with simple examples to help their understanding.

Concretely, our new key enumeration algorithm is an adaptation of the rank estimation algorithm of Glowacz et al. [7]. As in this previous work, we use histograms to efficiently represent the key log probabilities, and the first step of the key enumeration is a convolution of histograms modeling the distribution of our N'_s lists of log probabilities. This step is detailed in Algorithm 1. In the rest of the paper we will denote the initial histograms $H_0, ..., H_{N'_s-1}$ and the convoluted histograms $H_{0:1}, ..., H_{0:N'_s-1}$ as written in the output of Algorithm 1. For illustration, Fig. 1 shows an example of its application in the case of two 4-bit subkeys of which the log probabilities are represented by a 7-bin histogram, which are convoluted in the lower part of the figure.

Algorithm 1. Convolution.

Input. N'_s lists of log probabilities LP_i's, and number of bins N_b.
Output. Histograms of the log probabilities of each sub-key: $H_0, \ldots, H_{N'_s-1}$, and their convolutions $H_{0:1}, \ldots, H_{0:N'_s-1}$.

$H_0 \leftarrow \mathsf{hist\_lin}(LP_0, N_b)$;
$H_1 \leftarrow \mathsf{hist\_lin}(LP_1, N_b)$;
$H_{0:1} \leftarrow \mathsf{conv}(H_0, H_1)$;
for $i = 2$ to $N'_s - 1$ **do**
  $H_i \leftarrow \mathsf{hist\_lin}(LP_i, N_b)$;
  $H_{0:i} \leftarrow \mathsf{conv}(H_i, H_{0,i-1})$;
end forreturn $H = [H_0, \ldots, H_{N'_s-1}, H_{0:1}, \ldots, H_{0:N'_s-1}]$.

Based on this first step, our algorithm allows to enumerate keys that are ranked between two bounds B_{start} and B_{stop}. In the standard situation where the adversary wants to enumerate starting from the most likely key, we set $B_{start} = 0$. However, there are at least two cases where other starting bounds can be useful. First, it is possible that one wishes to continue an enumeration that has been started previously. Second, and more importantly, the selection of these bounds directly allows efficient parallel key enumeration, where the amount of computation performed by each core is well balanced.

In order to enumerate all keys ranked between the bounds B_{start} and B_{stop}, the corresponding indexes of $H_{0:N'_s-1}$ have to be computed, as described in Algorithm 2. It simply sums the number of keys contained in the bins starting

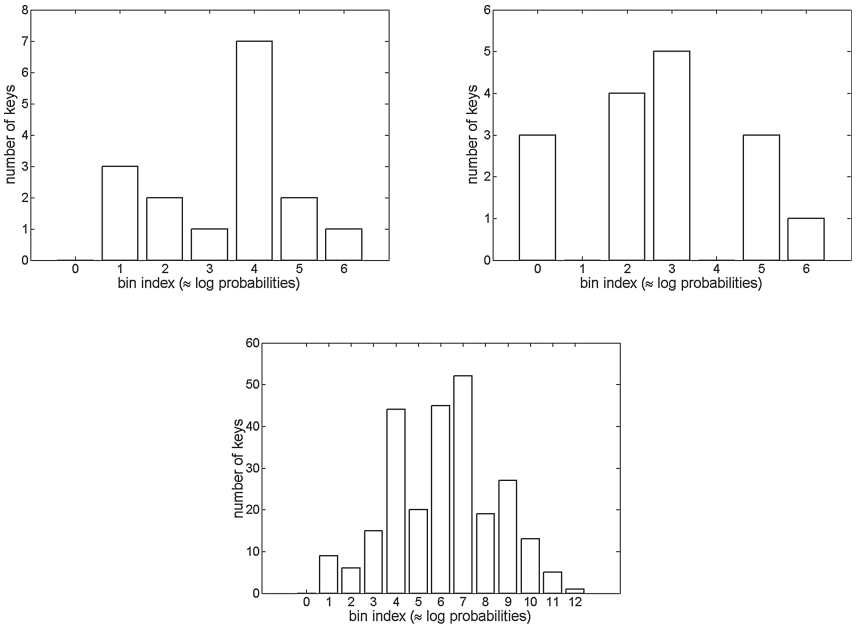

Fig. 1. Histograms representing the log probabilities of two 4-bit subkeys and their convolution. Upper left: $H_0 = [0, 3, 2, 1, 7, 2, 1]$. Upper right: $H_1 = [3, 0, 4, 5, 0, 3, 1]$. Bottom: $H_{0:1} = [0, 9, 6, 15, 44, 20, 45, 52, 19, 27, 13, 5, 1]$.

from the most likely one, until we exceed B_{start} and B_{stop}, and returns the corresponding indexes x_{start} and x_{stop}. That is, x_{start} (resp. x_{stop}) refers to the index of the bin where B_{min} (resp. B_{max}) is achieved (thus $x_{start} \geq x_{stop}$).

As in [7], a convenient feature of the histograms we use to represent the key log probabilities is that they lead to simple bounds on the "enumeration error" that is due to their rounding, hence on the additional workload needed to compensate this error. Namely, if one wants to be sure to enumerate all the keys of which the rank is between the bounds, then he should add $\lceil \frac{N'_s}{2} \rceil$ to x_{start} and substract it to x_{stop}.[1]

Figure 2 illustrates the computation of these indexes using the same example as in Fig. 1. In this case, the user wants to find the bins where the keys are ranked between 10 and 100. By summing up the number of keys contained in the bins of $H_{0:1}$ from the right to the left, we find that the bin indexed 10 starts with the rank 7 and the bin indexed 7 ends with the rank 117. Since the bin indexes 11 and 6 are out of the bounds (10 and 100), we know that the dark grey bins approximately contains the keys we want to enumerate, up to rounding errors. Furthermore, by adding the light grey bins, we are sure that all the keys

[1] The authors of [7] had a slightly worst bound of N'_s instead of $\lceil \frac{N'_s}{2} \rceil$. Indeed, they rounded the sum of all the subkeys' log probabilities, instead of summing the rounded subkeys' log probabilities.

Algorithm 2. Computation of the indexes' bounds.

Input. Lower and upper bounds on the key rank B_{start}, B_{stop}.
Output. Indexes of the bins between the bounds x_{start}, x_{stop}.

$x_{start} \leftarrow \mathsf{size\_of}(H_{0:N'_s-1}) - 1$;
$cnt_{start} \leftarrow 0$;
while $cnt_{start} < B_{start}$ **do**
  $cnt_{start} \leftarrow cnt_{start} + H_{0:N'_s-1}(x_{start})$;
  $x_{start} \leftarrow x_{start} - 1$;
end while
$cnt_{stop} \leftarrow cnt_{start}$;
$x_{stop} \leftarrow x_{start}$;
while $cnt_{stop} < B_{stop}$ **do**
  $cnt_{stop} \leftarrow cnt_{stop} + H_{N'_s-1}(x_{stop})$;
  $x_{stop} \leftarrow x_{stop} - 1$;
end whilereturn x_{start}, x_{stop}.

between the ranks 10 and 100, as would be produced by an optimal enumeration algorithm (like [10]), are covered by the bins.

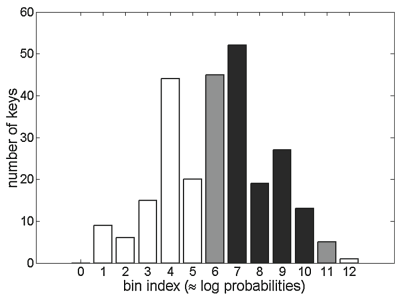

Fig. 2. Computation of the indexes' bounds for $B_{min} = 10$ and $B_{max} = 100$.

Given the histogram of the key log probabilities and the indexes of the bounds between which we want to enumerate, the enumeration simply consists in performing a backtracking over all the bins between x_{start} and x_{stop}. More precisely, during this phase we recover the bins of the initial histograms (i.e. before convolution) that we used to build a bin of the convoluted histogram $H_{0:N'_s-1}$. For a given bin b with index x of $H_{0:N'_s-1}$ corresponding to a certain log probability, we have to run through all the non-empty bins $b_0, ...b_{N'_s-1}$ of indexes $x_0, ...x_{N'_s-1}$ of $H_0, ..., H_{N'_s-1}$ such that $x_0 + ... + x_{N'_s-1} = x$. Each b_i will then contain at least one and at most $2^{m \cdot a}$ subkey(s) that we must enumerate. This leads to a *keyfactorization* which is a table containing N'_s subkey lists, such that each of these lists contains up to $2^{m \cdot a}$ subkeys associated to the bin b_i of the histogram H_i. Any combination of N'_s subkeys, each one being picked in a different list, results in a master key having the same rounded probability. Eventually, each time a factorization is completed, we call a fictive function process_key which

takes as input the result of this factorization. This function could test the keys on-the-fly or send them to a third party for testing (this function is essentially independent of the enumeration process).

Algorithm 3 describes more precisely this bin decomposition process. From a bin index $x_{0:i}$ of $H_{0:i}$, we find all the non empty bins of indexes x_i of H_i such that the corresponding bin of index $x_{0:i} - x_i$ of $H_{0:i-1}$ is non empty as well. All the bins b_i following this property will lead to valid subkeys for k_i that we add to the key factorization using the function $\mathsf{get}(H_i, b_i)$. This is done for all convolution results from the last histogram $H_{0:N'_s-1}$ to the first $H_{0:1}$, which then leads to a full key factorization.

Algorithm 3. Bin decomposition.

Input. H: the structure containing all the histograms output by Algorithm 1;
csh ($current\_small\_hist$): the index i of the current histogram H_i we target;
x_{bin}: The bin index of $H_{0:i}$ we want to decompose.

Output. kf ($key\_factorization$): the array of N'_s subkey lists containing factorized keys.

Inline comments are given in Table 1

```
if csh == 1 then
    x ← size_of(H_0) − 1;
    while (x ≥ 0) & (x + size_of(H_1)) ≥ x_bin) do                ▷ (1) and (2)
        if H_0(x) > 0 & H_1(x_bin − x) > 0 then                   ▷ (3)
            kf(1) ← get(H_0, x);
            kf(0) ← get(H_1, x_bin − x);
            process_key(kf);
        end if
        x ← x − 1;
    end while
else
    x ← size_of(H_csh) − 1;
    while (x ≥ 0) & (x + size_of(H_0:csh−1) ≥ x_bin) do           ▷ (4) and (5)
        if H_csh(x) > 0 & H_0:csh−1(x_bin − x) > 0 then           ▷ (6)
            kf(csh) ← get(H_i, x);
            Decompose_bin(csh − 1, x_bin − x, H, kf);
        end if
        x ← x − 1;
    end while
end if
```

In order to help the understanding of Algorithm 3, we provide an example of bin decomposition in Fig. 3. In this example, we want to enumerate all the keys having their log probability in the 7th bin of $H_{0:1}$ (represented with black stripes in the bottom part of the figure). Since this bin is not empty, we know that such keys exist. Hence, in order to enumerate all of them, we iterate over all the bins b_0 with indexes x_0 of H_0 and look if the corresponding bins b_1 with indexes $7 - x_0$ of H_1 are non-zero (i.e. contain at least one key). Whenever this

Table 1. Comments for Algorithm 3.

(1)	If $x < 0$ we looked at all the bins of H_0
(2)	If $x > x_{bin} - \mathsf{size\_of}(H_1)$ we looked at all the bins of H_1.
(3)	If $H_0(x) > 0$ & $H_1(x_{bin} - x) > 0$ we have two non-zero bins such that the sum of the indexes matches, thus we found valid subkeys lists for k_0 and k_1.
(4)	If $x < 0$ we looked at all the bins of H_{csh}.
(5)	If $x > x_{bin} - \mathsf{size\_of}(H_{0:csh})$ we looked at all the bins of $H_{0:csh}$.
(6)	If $H_{csh}(x) > 0$ & $H_{0:csh-1}(x_{bin} - x) > 0$ we have two non-zero bins such that the sum of the indexes matches, thus we found a valid subkeys list for k_{csh}

happens, we found a key factorization which corresponds to all the combinations of the subkeys contained in the bin b_0 of H_0 and the bin b_1 of H_1. These possible bin combinations are represented in the same color for H_0 (resp. H_1) in the top left (resp. top right) part of the figure. The bins in white are those for which no such combination is possible. For example, subkeys with log probability in the fourth bin of H_0 would require subkeys with log probability in the fifth bin of H_1 (so that $3 + 4 = 7$), but this fifth bin of H_1 is empty.

The generalization of this algorithm simply follows a recursive decomposition. That is, in order to enumerate all the keys within a bin b of index x in $H_{0:N'_s-1}$, we find two indexes $x_{N'_s-1}$ and $x - x_{N'_s-1}$ of $H_{N'_s-1}$ and $H_{0:N'_s-2}$ such that the corresponding bins are not empty. All the keys in the bin index $x_{N'_s-1}$ of $H_{N'_s-1}$ will be added to the key factorization. We then continue the recursion with the bin $x - x_{N'_s-1}$ of $H_{0:N'_s-2}$ by finding two non-empty bin indexes $x_{N'_s-2}$ and $x - x_{N'_s-1} - x_{N'_s-2}$ of $H_{0:N'_s-3}$, etc.

Finally, the main loop of our new enumeration is given in Algorithm 4. It simply calls Algorithm 3 for all the bins of $H_{0:N'_s-1}$ which are between the enumeration bounds.

Algorithm 4. Histogram-based enumeration.

Input. H: the structure containing all the histograms output by Algorithm 1;
x_{start}: the bin index of $H_{0:N'_s-1}$ from which we start the enumeration;
x_{stop}: the bin index of $H_{0:N'_s-1}$ from which we end the enumeration.
Output. kf ($key\_factorization$) :the array of N'_s subkey lists containing the factorized keys.

```
for x = x_start to x_stop do
    Decompose_bin(N'_s − 1, x, H, kf);
end for
```

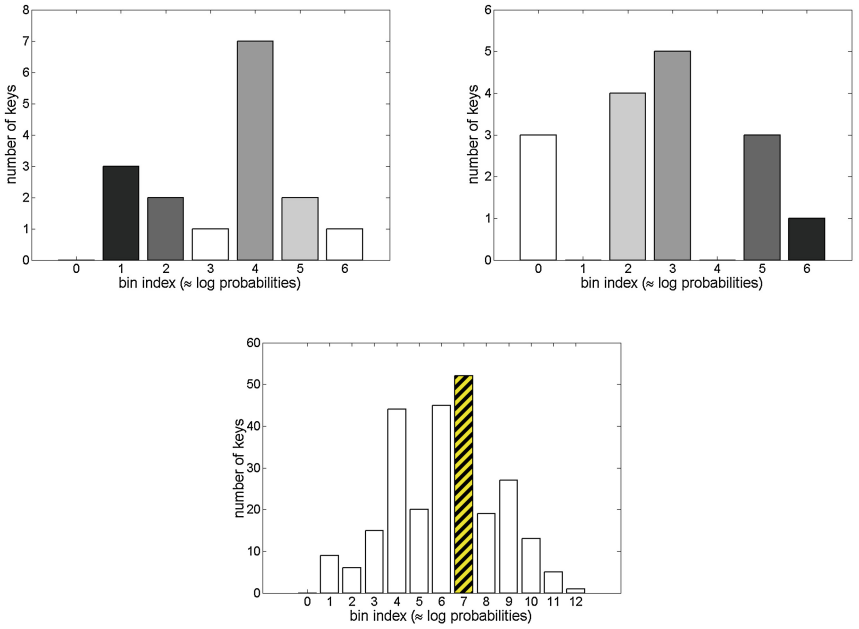

Fig. 3. Enumeration with histogram for a (shifted) log probability of 7.

4 Open Source Code

For usability, we join an open source implementation of our key enumeration algorithm to this paper. For completeness and in view of the similarity of the techniques they exploit, we also include the key rank estimation algorithm of [7] in this tool. The corresponding program is compiled using `G++` and uses the NTL library [2] in order to compute the histogram convolutions. It works on Windows and Linux operating systems (and probably on MAC). In this section, we describe the inputs and outputs that have to be set before running key enumeration and rank estimation. The code is provided as supplementary material to the paper, with an example of utilization.

Note that while the previous section only describes the general idea of the algorithm, its implementation contains a couple of additional optimizations, typically involving precomputations, iterating only over non-zero bins of the histograms and ordering the convolution, which allows significant performance improvements.

Inputs of the Rank Estimation Algorithm.

- log_proba: the $N_s \times 2^a$ matrix encoded in double precision containing the subkey log probabilities obtained during thanks to the attack.
- real_key: the N_s-element vector containing the real subkeys values.

- nb_bin : the number of bins for the initial histograms H_i.
- merge: the value of merging. A value of 1 will not do any merging. A value of 2 will merge lists by two, this gives us $N'_s = \lceil \frac{N_s}{2} \rceil$ lists of 2^{2a} elements. The current version supports only a maximum merging value of 3, which means $N'_s = \lceil \frac{N_s}{3} \rceil$ lists of 2^{3a} subkeys.

Inputs of the Key Enumeration Algorithm.

- All the inputs of the rank estimation algorithm (with real_key being optional).
- bound_start: the starting bound of the enumeration. If this is e.g. set to 2^{10}, the enumeration will start from the closest bin of $H_{0:N'_s}$ such that at most 2^{10} keys are contained in the next bins.
- bound_stop: the ending bound of the enumeration. If this is e.g. set to 2^{32}, the enumeration will start from the closest bin of $H_{0:N'_s}$ such that at least 2^{32} keys are contained in the next bins.
- test_key: this is a boolean value. If set to 1, the enumeration algorithm will test the keys on-the-fly using an AES implementation, by recombining them from the factorizations (and stop when the key is found); if set to 0, it will keep the keys factorized, and the user should implement himself the way he wants to test the keys in the process_key function.
- texts: a $4 \times N_s$ matrix containing two plaintexts and their associated ciphertexts. These two plaintexts/ciphertexts are used to test on-the-fly if the correct key is found. This parameter does not have to be initialized if test_key is set to 0.
- to_bound: This is a boolean value. If set to 1, the enumeration algorithm will remove (resp. add) $\frac{N'_s}{2}$ to *index_max* (resp. *index_min*) as described in the previous section, to ensure that we enumerate all the keys between bound_start and bound_stop.
- to_real_key: additional parameter for comparisons with previous works, that can take 4 values in $[0, 4]$. If set to 0, this parameter is ignored. If set to to $1, 2, 3$, it allows the user to measure the timing of enumerating up to the real key in different settings, ignore the value of bound_start and test_key and enumerate up to the bin that contains the real key. It then requires real_key to be initialized. If set to 1, the keys will neither be recombined nor tested. If set to 2, the keys are recombined but not tested with AES (it simply tests if the key is equal to the real one provided by the user). If set to 3, the keys are recombined and tested with the AES. If the real key rank is bigger than bound_end, the enumeration is aborted.

Algorithms Outputs.

- **Rank estimation informations:** returns the rank of the real key according to its rounded log probabilities and the min and max bounds on the actual rank of the real key. Also returns the time needed for rank estimation (including the preprocessing time).

– **Enumeration informations:** If the key has been found, returns the rank of the real key according to its rounded log probabilities and the min and max bounds on the actual rank of the real key. Also returns the time needed for preprocessing and the time needed for enumeration.

Examples. Together with our code, we provide different examples of key enumeration which are written in a file main_example.cpp and listed in Table 2. The first example (first line in the table) enumerates all the keys of rounded rank between 2^{10} and 2^{40} (taking the rounding bounds into account) and tests them using a reference AES-128 software implementation. The second example enumerates all the keys of rounded rank between 2^0 and 2^{40} without testing them. A user would then have to define the way he wants to implement the process_key function (e.g. by sending the factorized lists to a powerful third testing party). The last three examples enumerate all the keys up to the real one if its rounded rank is lower than 2^{32}. For the third one, the recorded timing will correspond to the enumeration time with factorization. For the fourth one, the recorded timing will correspond to the enumeration time including the recombination of the factorized lists. For the last one, the recorded timing will correspond to the enumeration time with key testing (with our reference AES-128 implementation) and thus with recombination.

Table 2. Running examples for key enumeration

to_real_key	real_key	test_key	to_bound	texts	bound_start	bound_stop	
0	*optional*	1	1	*given*	2^{10}	2^{40}	(1)
0	*optional*	0	0	–	2^0	2^{40}	(2)
1	*needed*	–	–	–	–	2^{32}	(3)
2	*needed*	–	–	–	–	2^{32}	(4)
3	*needed*	–	–	–	–	2^{32}	(5)

5 Performance Evaluations

In this section we evaluate the performances of our enumeration algorithm and discuss its pros and cons compared to previous proposals. For this purpose, we consider a setting of simulated leakages for an AES-128 implementation, which has been previously used for comparison of other enumeration algorithms [4,8,10]. Namely, we target the output of an AES S-box, leading to 16 leakages of the form $l_i = \mathsf{HW}(\mathsf{S}(x_i, k_i)) + N$ for $i \in [0, 15]$, with HW the Hamming weight leakage function and N a random noise following a Gaussian distribution. We stress that the main experimental criteria influencing the complexity of an enumeration is the rank of the key (that we can control thanks to the noise variance). So other experimental settings would not lead to significantly different conclusions with respect to the performances of the enumeration algorithm.

Besides, the two main parameters of our algorithm are the number of bins and the amount of merging. Intuitively, a smaller number of bins leads to a faster execution time at the cost of an increased quantization error, and merging accelerates the enumeration at the cost of more memory requirements and preprocessing time. All the following experiments were performed with 256, 2048 and 65536 bins, and for an amount of merging of 1, 2 and 3. These values were chosen to allow comparisons with the results of [8]. That is, 256 (resp. 2048 and 65536) bins is similar to choosing a precision of 8 (resp. 11 and 16) bits for their algorithm. We limited the amount of merging to 3 because the memory requirements of this preprocessing then becomes too large for our AES-128 case study (a merging of 4 would require to store $4 \times 2^{32} \times 8$ bytes for the lists of log probabilities in double precision).

5.1 Enumeration Accuracy

One convenient feature of our algorithm is its ability to compute easily the quantization bounds related to the mapping from floating to integers. Since accuracy is usually the main concern when enumerating keys, we start our evaluations by analyzing the impact of the number of bins on these quantization bounds. For this purpose, we first recall that these quantization errors are related to the rounding, which was the key idea to improve the performance and parallelism of recent works on enumeration. Hence, our goal is to find the level of quantization errors that are acceptable from the enumeration accuracy point-of-view.

Figure 4 illustrates this impact for a precision of 256, 2048 and 65536 bins. Since the impact of merging is minor for such experiments, we only report the results with a merge_1 preprocessing. The Y-coordinate represents the number of keys one has to enumerate in order to guarantee an enumeration up to an exact key rank given by the X-coordinate. Optimal enumeration is shown in black (for which $X = Y$) and corresponds to the application of the algorithm in [10]. The red, blue and green curves respectively represent the maximum, average and minimum results we found based on a sampling of 1000 enumeration experiments. These experiments lead to two interesting observations. First, a lower precision (e.g. 256 bins) leads to larger enumeration overheads for small key ranks, but these overheads generally vanish as the key ranks increase. Second, increasing the number of bins rapidly makes the enumeration (rounding) error low enough (e.g. less than one bit) which is typically observed for the 2048- and 65536-bin cases, especially for representative ranks (e.g. beyond 2^{32}) where the enumeration cost becomes significant. This is in line with the observations made with histogram-based rank estimation [7].

Note that other algorithms such as [4,8] lead to similar accuracies with similar parameters (e.g. our 2048-bin case roughly corresponds to their 11-bit precision case). Besides, finding bounds on the rounding error should be feasible for [8] too, despite probably more involved than with histograms for which such bounds come for free.

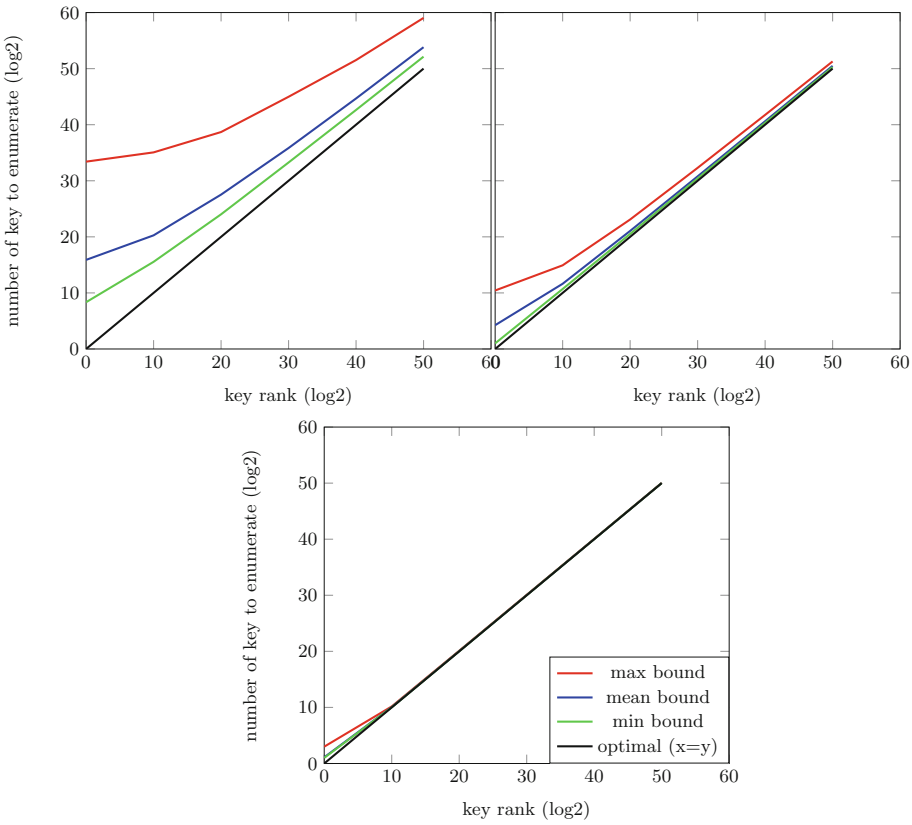

Fig. 4. Enumeration overheads due to rounding errors with merge_1 (i.e. no merging). Upper left: 256 bins. Upper right: 2048 bins. Bottom: 65536 bins. (Color figure online)

5.2 Factorization

Another important feature of our method is its intrinsic ability to output factorized keys instead of a single key at a time. Studying why and how this factorization evolves with our main parameters is important for two reasons. Firstly, it allows a better understanding of how our main parameters affect the performances of histogram-based enumeration, since a better factorization always reduces its amount of processing. Secondly, the number of keys per factorization may be important for the key testing phase, e.g. in case one wants to distribute the lists of key candidates to multiple (hardware) devices and therefore minimize the bandwidth of this distributed part of the computations. This second point will be discussed in Sect. 6.

Intuitively, increasing the amount of merging or decreasing the number of bins essentially creates more collisions in the initial histograms, thus increases the size of the factorized keys, and thus accelerates the enumeration process.

Interestingly, increasing the merging does not decrease the accuracy (by contrast with decreasing of the number of bins). Hence, this type of preprocessing should (almost) always be privileged up to the memory limits of the device on which the enumeration algorithms is running.

To confirm this intuition, Fig. 5 illustrates an evaluation of the factorization results for 256 (left) and 2048 (right) bins, and merging values from 1 to 3. The top figures represent the number of keys per factorization (Y-coordinate). The bottom figures represent the memory cost of the corresponding lists in bytes (Y-coordinate). The dashed curves represent the average value (over 1000 experiments) and the plain curves represent the maximum that occurred on our 1000 experiments. As we can see, using 256 bins leads to a lot of collisions and the merging value always amplifies the number of collisions. This increases the number of keys per factorization along with the memory size of the corresponding lists. The memory cost is anyway bounded by $N'_s \times 2^{m \cdot a}$, and the number of keys per factorization by $2^n = 2^{N'_s \cdot m \cdot a}$ (this extreme case would occur if all the subkeys have the same rounded probability and thus are within the same bins for all histograms H_i). We did not plot the results for 65536 bins since few collisions appear (and thus not many of factorizations).

Note that the algorithm in [8] has a similar behavior as it stores the keys having the same probabilities within a tree. So although the open source implementation joined to this previous work recombines the keys, it could also convert this tree representation into a factorized representation that is more convenient for distributed key testing with limited bandwidth.

5.3 Time Complexity

We finally discuss the performances of our algorithm in terms of timing. For this purpose, all our experiments were performed using i7-3770 CPU running at 3.40 GHz with 16 GB of RAM on Ubuntu. We start by comparing our results to the C++ implementation of the optimal key enumeration algorithm of Veyrat et al. [10], and consider the costs of enumeration only (i.e. we exclude the key testing and measure the time it takes to output factorized lists of keys). We then discuss the comparison with the work of Martin et al. at the end of the section.

Results for 256 and 65536 bins are given in Fig. 6. The Y-coordinate represents the time (in seconds) taken to enumerate keys until finding the correct one, for different ranks represented in the X-coordinate (in log2). As expected, the enumeration time without key testing is extremely fast for a (low) precision of 256 bins (in the upper part of the figure). For a merge_1 preprocessing, it takes less than 10 s to enumerate up to 2^{35} on average. For a merge_2 preprocessing, it does not even take a second. The bottom part of the figure then shows the results for 65536 bins with merge_1 and merge_2 preprocessings. Interestingly (and exceptionally), using the merge_2 preprocessing is worst than using merge_1 in this case. This is due to the fact that the 65536 bins do not bring enough collisions.

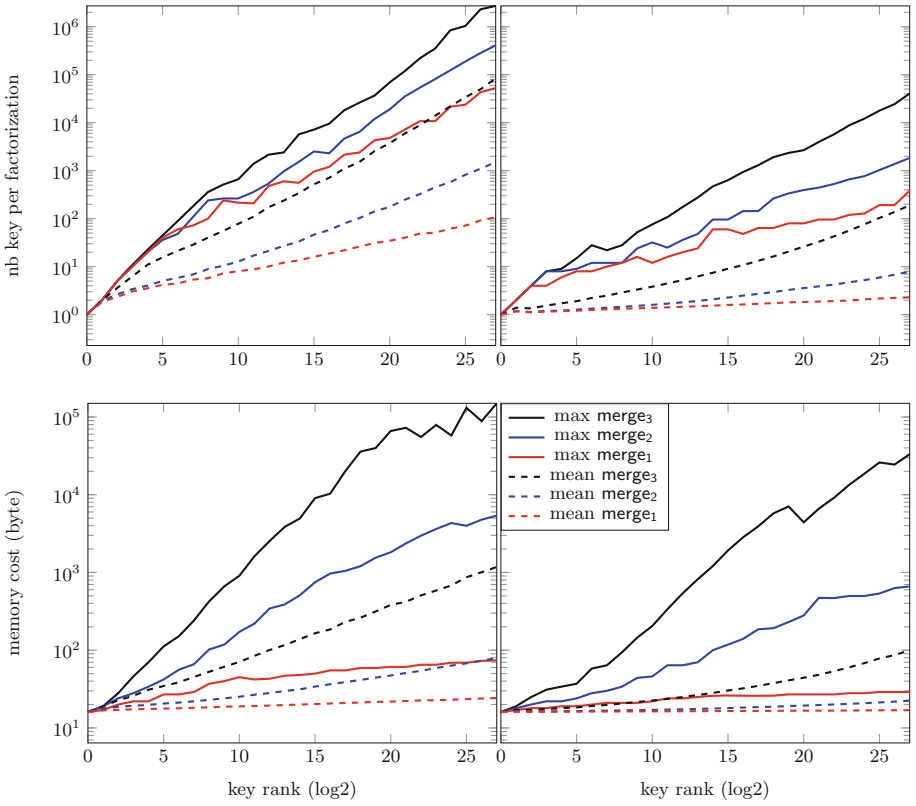

Fig. 5. Key factorization for different levels of merging and number of bins. Left: number of keys per factorization (top) and memory cost of the associated list in bytes (bottom) for 256 bins. Right: same plots for 2048 bins.

Hence, we loose more by iterating over all the non-empty bins than what we win from the collisions. Additional results for 2048 bins and other merging values are given in Appendix A.

We next discuss a number of additional issues related to these performances.

Preprocessing and Memory. The preprocessing time and the memory requirements of the algorithm are almost null for merge_1 and merge_2 preprocessings (i.e. less than a second and less than 30 Mb). However, merge_3 is more expensive in time and memory. Indeed, the algorithm has to keep the merged scores and the subkeys lists that fall into each bins in memory. In our experiments done for an AES-128 implementation, we have to process 5 lists of 2^{24} elements and one of 2^8. This requires approximatively 3.5 Gb of memory and 45 s of preprocessing. As for the other key enumeration algorithm based on rounded log probabilities (i.e. [4,8]), the memory requirements are independent of the enumeration depth.

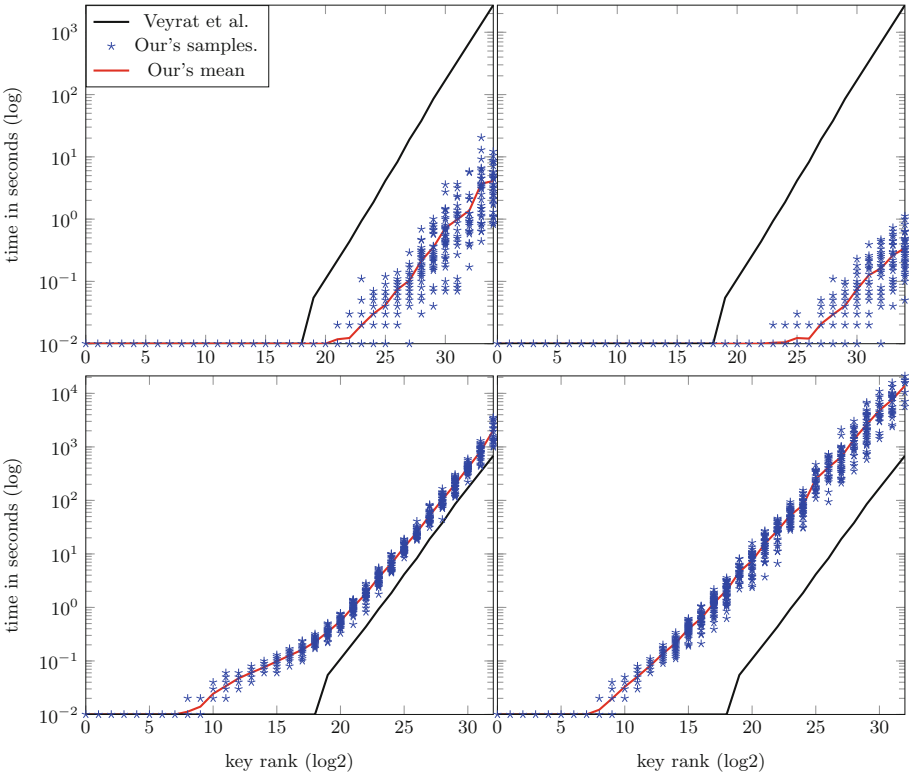

Fig. 6. Execution time for 256 and 65536 bins with factorized lists. The blue stars are the samples for our algorithm, the red curve is the corresponding mean, and the black curve is for the optimal enumeration algorithm in [10]. Upper left: 256 bins / merge_1. Upper right: 256 bins / merge_2. Bottom left: 65536 bins / merge_1. Bottom right: 65536 bins / merge_2. (Color figure online)

Parallelization. Our algorithm allows a very natural parallelization with minimum communication. Namely, after $H_{0:N'_s-1}$ has been computed, if the user wants to parallelize the enumeration among c cores, he simply has to split the bins into c sets which will contain approximatively the same amount of keys, so that each core will have approximatively the same workload. From our preliminary experiments, the gain of such a parallelization is linear in the number of threads, as expected.

Note that other key enumeration algorithms such as [4,8] can also be easily parallelized, but balancing the effort done by each core may be slightly more involved. This difference is again due to our treatment with histograms. That is, while we can directly select the ranks between which we want to enumerate with our starting and ending bounds, the solution in [8] rather has to select the minimum and maximum probabilities of the keys between which we want to enumerate, without a priori knowledge of the amount of keys it represents.

So good balance between the cores requires an (admittedly light) preprocessing to estimate the workload of the different cores. Besides, the heuristic nature of [4] (which is not aimed at optimal enumeration) also makes it difficult to compare from this point of view.

Comparison with the Enumeration Algorithm of Martin et al. To conclude this section, we add a comparison with the work in [8] and produce performance evaluations using their Java open source (using their "shift to 1 trick" to speed up the enumeration). We insist that this comparison is informal, since comparing implementations with different programming languages and optimization efforts, and therefore is only aimed to highlight that the simplicity of our enumeration algorithm reasonably translates into efficient implementations compared to the best state-of-the-art enumeration algorithms.

Figure 7 shows our comparison results for both 8 and 11 bits of precision (corresponding to 256 and 2048 bins for our algorithm). Since the implementation of Martin et al. measures the time to output the keys one by one (represented by the red curve in the figure), we consider a similar scenario for our algorithm (represented by the green curve in the figure). After some (quite irrelevant) time overheads of the implementation from [8] for low key ranks, we see that both algorithms reach a similar slope when the key ranks increase – yet with a sight constant gain for our implementation. Furthermore, increasing the precision and number of bins amplifies the initial overheads of the implementation from [8] while making the performances of both algorithms more similar for larger ranks. The additional black curve corresponds to the Java adaptation of the algorithm in [10], which allows us to check consistency with the previous comparisons from Martin et al. Since our algorithm allows to output factorized keys, we plotted in blue the associated timing. Eventually, we have no timing comparison with the work of [4] since the authors did not release their implementation. Extending the comparing enumeration algorithms in a more systematic manner is anyway an interesting scope for further research.

6 Application Scenarios

In this section we finally discuss the impact of our findings for an adversary willing to exploit enumeration in concrete application scenarios, which complements the similar discussion that can be found in [4]. Without loss of generality we focus on the case of the AES. We separate this discussion in the cases of adversaries having either a small or big computing infrastructure to mount an attack.

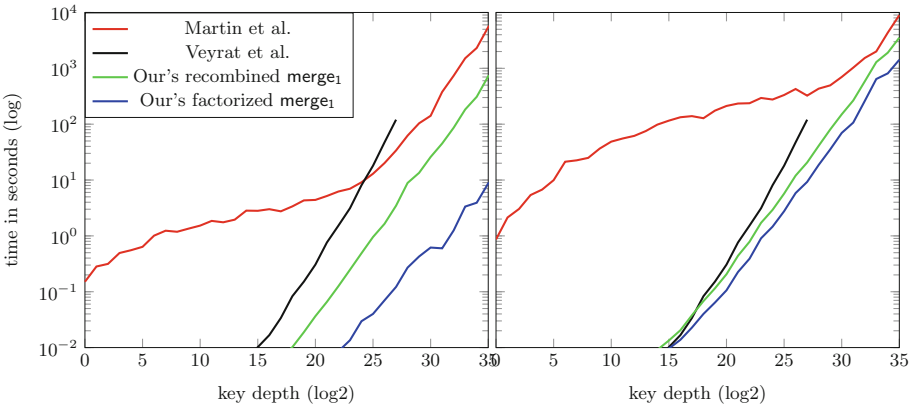

Fig. 7. Execution time for the java implementation of Veyrat et al., Martin et al. and ours with 8 bits of precision (left) and 11 bits of precision (right). (Color figure online)

In the first case we assume a local attacker with only one "standard" computer. Roughly, his computing power will be bounded by 2^{40}. In that case, he will simply use all his available cores to launch the key enumeration with key testing on-the-fly. Since it is likely that it will take more time to compute an AES encryption than to output a key candidate, this adversary will prefer a higher precision than a higher number of collisions. In that respect, and depending on the AES implementation's throughput, using 2048 bins could be a good tradeoff. Indeed, as the adversary's computing power is bounded and as the AES computation is the costly part, he should minimize the bounds overhead as seen in Sect. 5.1. Since the merging value has no impact on the accuracy, this value should always be maximized (ensuring we do not fall in a case where it slows down the enumeration process as shown in Sect. 5.3).

By contrast, the strategy will be quite different if we assume the adversary is an organization having access to a big computing infrastructure. For example, let assume that this organization has powerful computer(s) to launch the key enumeration along with many hardware AES implementations with limited memory. The adversary's computing power is now bounded by a much higher capability (e.g. 2^{64}). As we saw in Sect. 5.1, the gap between the optimal enumeration and the efficient one (using less bins) vanishes as we consider deeper key ranks. In that case, the attacker should maximize the enumeration throughput and minimize the bandwidth requirement (per single key), which he can achieve by decreasing the number of bins and increasing the merging value as much as possible (e.g. 256 bins with merge_3). All the key factorizations would then be sent to the hardware devices for efficient key testing. This could be done easily since a factorized key can be seen as a small effort distributor as in [9,12].

7 Related Work

A recent work from David et al. available on ePrint [6] allows one to enumerate keys from real probabilities without the memory issue of the original optimal algorithm from [10]. This gain comes at the cost of a loss of optimality which is different from the one introduced by the rounded log-probabilities.

8 Conclusion

This paper provides a simple key enumeration algorithm based on histograms along with an open source code implementing both the new enumeration method and the rank estimation algorithm from FSE 2015. In addition to its simplicity, this construction allows a sound understanding of the parameters influencing the performances of enumeration based on rounded probabilities. Additional convenient features include the easy computation of bounds for the rounding errors, and easy to balance parallelization. Our experiments also illustrate how to tune the enumeration for big distributed computing efforts with hardware co-processors and limited bandwidth. We believe the combination of efficient key enumeration and rank estimation algorithms are a tool of choice to help evaluators to understand the actual security level of concrete devices, and the actual capabilities of computationally enhanced adversaries.

Acknowledgements. François-Xavier Standaert is a research associate of the Belgian Fund for Scientific Research. This work has been funded in parts by the ERC project 280141 (acronym CRASH), the ERA-Net CHIST-ERA project SECODE and the DFG Research Training Group GRK 1817 Ubicrypt.

A Additional Time Complexites

Figure 8 shows timing results for different number of bins and amounts of merging. The two figures on the top are the results for 256 (left) and 65536 (right) bins with merge_3 which are lacking in Fig. 6. As for the 65536-bin case, we saw in Fig. 6 that the merging can be detrimental (e.g. using merge_1 was better than using merge_2) when not enough collision are occur. However we see that we still benefit from using merge_3 in that case. The 3 other figures show the results of experiments with 2048 bins and a merge_1 preprocessing (middle left), merge_2 preprocessing (middle right) and merge_3 preproicessing (bottom).

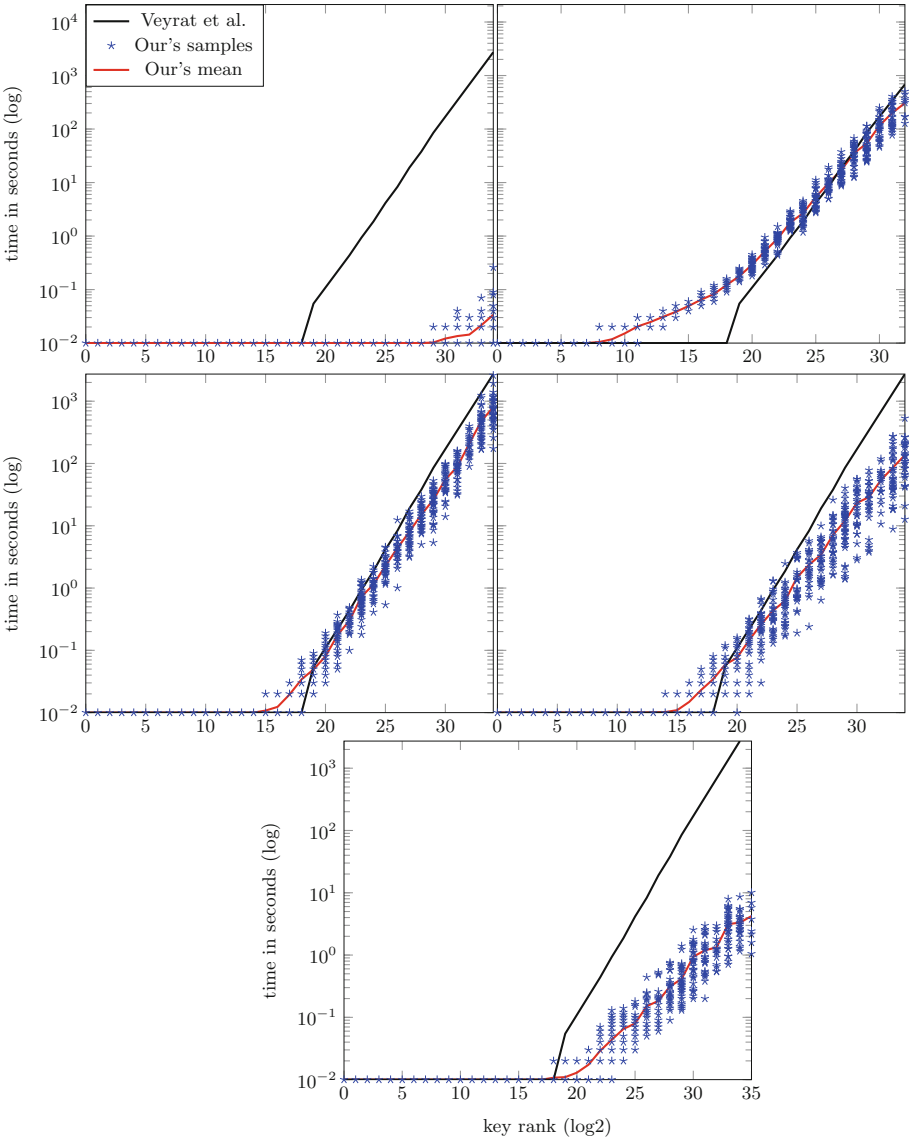

Fig. 8. Additional execution times with factorized lists. Upper left: 256 bins / merge_3. Upper right: 65536 bins / merge_3. Middle left: 2048 bins / merge_1. Middle right: 2048 bins / merge_2. Bottom: 2048 bins / merge_3.

References

1. http://perso.uclouvain.be/fstandae/PUBLIS/172.zip
2. http://www.shoup.net/ntl/
3. Bernstein, D.J., Lange, T., van Vredendaal, C.: Tighter, faster, simpler side-channel security evaluations beyond computing power. IACR Cryptol. ePrint Arch. **2015**, 221 (2015)
4. Bogdanov, A., Kizhvatov, I., Manzoor, K., Tischhauser, E., Witteman, M.: Fast and memory-efficient key recovery in side-channel attacks. IACR Cryptol. ePrint Arch. **2015**, 795 (2015)
5. Clavier, C., Danger, J.-L., Duc, G., Elaabid, M.A., Gérard, B., Guilley, S., Heuser, A., Kasper, M., Li, Y., Lomné, V., Nakatsu, D., Ohta, K., Sakiyama, K., Sauvage, L., Schindler, W., Stöttinger, M., Veyrat-Charvillon, N., Walle, M., Wurcker, A.: Practical improvements of side-channel attacks on AES: feedback from the 2nd DPA contest. J. Cryptograph. Eng. **4**(4), 259–274 (2014)
6. David, L., Wool, A.: A bounded-space near-optimal key enumeration algorithm for multi-dimensional side-channel attacks. IACR Cryptol. ePrint Arch. **2015**, 1236 (2015)
7. 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 (2015)
8. Martin, D.P., O'Connell, J.F., Oswald, E., Stam, M.: Counting keys in parallel after a side channel attack. In: Iwata, T., et al. (eds.) ASIACRYPT 2015. LNCS, vol. 9453, pp. 313–337. Springer, Heidelberg (2015)
9. Poussier, R., Grosso, V., Standaert, F.-X.: Comparing approaches to rank estimation for side-channel security evaluations. In: Homma, N. (ed.) CARDIS 2015. LNCS, vol. 9514, pp. 125–142. Springer, Heidelberg (2016). doi:10.1007/978-3-319-31271-2_8
10. 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 (2013)
11. 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 (2013)
12. Ye, X., Eisenbarth, T., Martin, W.: Bounded, yet sufficient? How to determine whether limited side channel information enables key recovery. In: Joye, M., Moradi, A. (eds.) CARDIS 2014. LNCS, vol. 8968, pp. 215–232. Springer, Heidelberg (2015)