How low can you go? Using side-channel data to enhance brute-force key recovery

Jake Longo, Daniel P. Martin, Luke Mather, Elisabeth Oswald, Benjamin Sach, and Martijn Stam

University of Bristol, Department of Computer Science,
Merchant Venturers Building, Woodland Road, BS8 1UB, Bristol, UK.
{jake.longo, dan.martin, luke.mather, elisabeth.oswald, ben.sach, martijn.stam}@bris.ac.uk

Abstract. Side-channel analysis techniques can be used to construct key recovery attacks by observing a side-channel medium such as the power consumption or electromagnetic radiation of a device while is it performing cryptographic operations. These attack results can be used as auxiliary information in an enhanced brute-force key recovery attack, enabling the adversary to *enumerate* the most likely keys first.
We use algorithmic and implementation techniques to implement a time- and memory-efficient key *enumeration* algorithm, and in tandem identify how to optimise throughput when bulk-verifying quantities of candidate AES-128 keys. We then explore how to best distribute the workload so that it can be deployed across a significant number of CPU cores and executed in parallel, giving an adversary the capability to enumerate a very large number of candidate keys.
We introduce the tool LABYNKYR, developed in C++11, that can be deployed across any number of CPUs and workstations to enumerate keys in parallel. We conclude by demonstrating the effectiveness of our tool by successfully enumerating 2^{48} AES-128 keys in approximately 30 hours using a modest number of CPU cores, at an expected cost of only 700 USD using a popular cloud provider.

1 Introduction

Brute-force attacks on cryptographic primitives or on systems with cryptographic components have been an important topic of research. The DES cipher, with an effective security parameter of 56 bits, was eventually demonstrated to be vulnerable to a direct brute-force attack using custom hardware in the order of days [6], and a brute-force attack on DES has been used to break the MS-CHAPv2 authentication protocol [14]. One can also consider brute-force attacks in which the adversary has *auxiliary information* available; consider dictionary-based password cracking tools such as oclHashcat[1], where an empirical prior distribution of popular passwords is used to facilitate the recovery of a hashed password.

[1] https://hashcat.net/oclhashcat/

Side-channel analysis (SCA) is a powerful tool for extracting cryptographic keys from secure devices. If an adversary can measure the power consumption of a device performing cryptographic operations, then the resulting power traces may subsequently lead to recovery of the secret key [10]. SCA attacks are typically *divide-and-conquer* strategies: targeting small portions of a key individually, obtaining information on the likelihood of each possible value for the portion of the key being the correct value, before combining these results to recover a full key.

Until recently, SCA attacks have been considered to be "all-or-nothing" strategies: 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. [18] 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 learns some information pertaining to the key, but not sufficient information to achieve key recovery and must hence *enumerate* the most likely candidate keys in order from most to least likely using a known plaintext and ciphertext pair. From this perspective, side-channel attacks are very similar to password cracking attacks—the auxiliary information is the result of a side-channel attack (respectively the password distribution), and the attacker's strength is constrained by the number of the most likely keys they are able to check (respectively passwords). Unlike password cracking, side channel attacks provide distributions on independent portions of the key, instead of on the entire key. This makes parallelisation of an enumeration algorithm recovering an entire key non-trivial.

Informally, the number of candidate keys an adversary must enumerate after an imperfect side-channel attack before learning the correct key is termed the *rank* of the key. The rank plays a very important role in determining the resistance of a device to a side-channel attack. Recent efforts [1,3,7,16,20,19] considered determining the rank of the correct (known) key after the side-channel phase of an attack. 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 defines Common Criteria security evaluation practice) to set up a specific working group to address the issue.

The rank is an extremely informative measure of security, but 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. However, it is also important to know how challenging this task is in practice, both in terms of the run-time for the enumeration and verification algorithms, as well as whether the adversary can parallelise their effort.

Our work focuses on exploring and optimising these properties. We build upon previous works (such as [3,18,20]) to create a highly parallel tool named LABYNKYR capable of enumerating large numbers of key candidates with the goal of minimising computation time, and demonstrate the effectiveness of our strategy by recovering a key at rank 2^{48} using a modest number of CPU cores in the order of *hours* of computation time. In Sec. 2 we provide formalisms and further motivation for the key enumeration problem. In Sec. 3 and Sec. 4 we identify fast algorithms for enumerating and verifying key candidates directly from side-channel data. We then move to exploring how best to parallelise these algorithms in Sec. 5, before concluding with some real-world benchmarks of the performance of our tool on a complex, real-world side-channel target in Sec. 6.

2 Key enumeration

Side-channel attacks. We use a bold type face 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 (we assume that all subkeys are of the same size). We denote this as $\mathbf{k} = (k^0, \ldots, k^{m-1})$ for a test key and $\mathbf{sk} = (sk^0, \ldots, sk^{m-1})$ for the target secret key.

We focus on SCA attacks where a distribution on the subkeys is output, so that key enumeration is applicable. There are myriad ways to construct SCA attacks (see [12] for an overview) and a whole host of accompanying literature dedicated to finding the 'best' attack. Formally, the attacker has N power measurements corresponding to encryptions of N known plaintexts $x_i \in \mathcal{X}$, $i = 1, \ldots, N$ and wishes to recover the secret key $\mathbf{sk}$. The attacker can accurately compute the internal values under each subkey (normally an independent part of the key) hypothesis. A distinguisher D^j is then some function which can be applied to the measurements and the hypothesis-dependent predictions (for the j^{th} subkey) in order to quantify the correspondence between them. The distinguisher D, on the complete key, simply runs the subkey distinguishers D^j for $j = 0 \ldots m-1$. The outputs from the (full) distinguisher and the subkey distinguishers are $\mathbf{D}$ and $\mathbf{D}^j$ respectively. The result of a side channel attack is a set of *distinguishing vectors* (referred to as a *distinguishing table*), which hold the information about subkeys (when studied individually) and the entire key (when studied jointly).

Each element in a distinguishing vector $\mathbf{D}^j$ contains a *score* associated with how likely the associated subkey value is to the be correct key. The score $d_{i,j}$ corresponds to the likelihood of subkey j taking value i. When an adversary uses a common class of attack strategies, these scores are often estimated correlation coefficients, but they can also take the form of probability-like values or entropies. In the case of a perfect SCA attack, the most likely score in each distinguishing vector will be associated with the correct subkey value, and to recover the key the adversary needs only to sort the vectors and read off the most likely values from each. When the correct subkey values have scores that are only *close to the most likely* of their respective vectors then the side-channel attack does not succeed perfectly, yet the scores still contains meaningful information on the value of the correct key.

An imperfect attack can occur for a variety of reasons. Many types of SCA attack require the adversary to make assumptions about the form of the information leakage produced in the side-channel, however, these assumptions are often imperfect. In other situations, there may be large amounts of noise in the data, forcing the adversary to gather more measurements. The quantity of measurements an adversary is able to require may be constrained: the adversary might have limited access to the target device, or the device may update the keys after a fixed number of encryptions have occurred. Often both issues are in-play: the adversary may have a only an approximation for the leakage and the measurements will contain noise. In this work we are not concerned with the particular type of SCA attack used; instead we focus on how to best exploit the information contained within $\mathbf{D}$ through enumeration.

Additive scores We assume that the score for an entire key can be calculated by adding the scores of its constituent subkeys. In practice, SCA attacks may produce scores that are additive by default, and in the event that this is not the case a suitable transformation such as taking the logarithm of the scores can usually be applied.

We denote the score of a key $\mathbf{k}_i$ as $S = \sum_{j=0}^{m-1} d_{k^j,j}$ where we assume k^j takes value $d_{k^j,j}$ in the distinguishing vector. We require that the scores are positive and that the most likely key has a *lower* score: if keys $\mathbf{k}_i$ and $\mathbf{k}_j$ have scores S_i, S_j and $S_i < S_j$ then $\mathbf{k}_i$ is judged by the SCA attack to more likely to be the correct candidate. If this is not automatically the case it is fairly straightforward to convert the scores such that this property holds.

Conversion to integer weights. In addition to assuming additive distinguishing scores, we also require that the (typically floating-point) scores $d_{i,j}$ be mapped to a set of integer weights $w_{i,j}$, mapping a distinguishing table $\mathbf{D}$ to a *weight table* $\mathbf{W}$ as per Fig. 1.

The conversion to weights is necessary to allow the application of the enumeration algorithms is described in Sec. 3. The particulars of the method "MapToWeight" used to do the conversion are described in Sec. 3.1.

Enumeration. Enumeration is a procedure that can be applied to the majority of SCA attacks; any divide-and-conquer SCA strategy is likely to produce an attack result to which enumeration can be applied. An enumeration capability is a boon for the adversary whenever the SCA attack does not succeed perfectly; allowing the adversary to trade-off a reduction in measurements for increased work enumerating through distinguishing vectors.

We are solely interested in how well enumeration can compensate for poor attacks. To this end, we describe the enumeration problem in terms of the output of a generic side-channel attack. After an attack has occurred an adversary has a weight table $\mathbf{W}$ which contains, for each subkey, a measure of the likelihood that subkey takes a particular value. A simple definition of a key enumeration algorithm takes in the weight table $\mathbf{W}$ and returns a list of the B most likely keys, where B is a predetermined budget. Additionally, it may be useful for the adversary to enumerate "at an offset" – to ignore the first O most likely keys (for instance which may have already been enumerated).

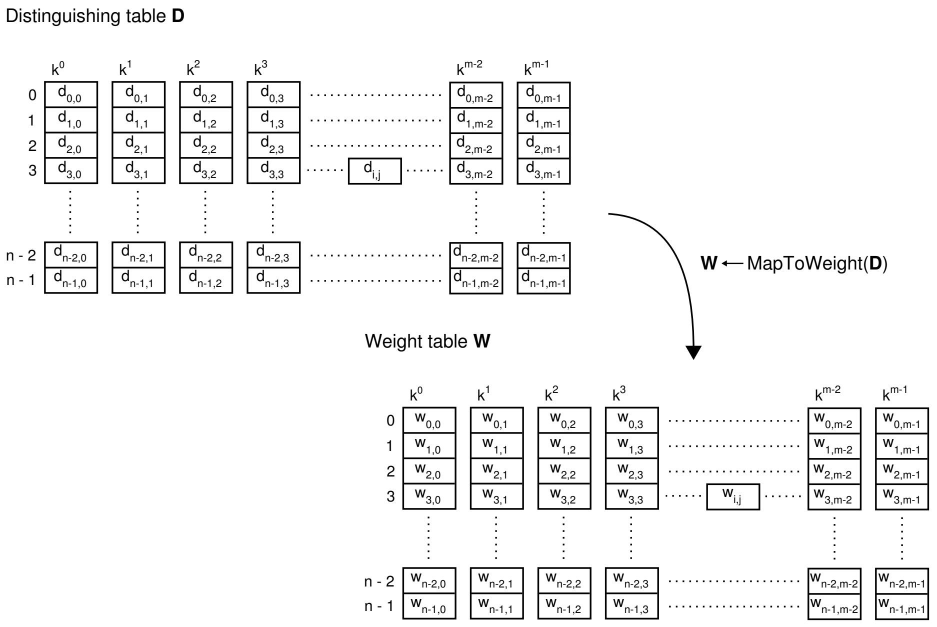

Fig. 1. The conversion of floating-point distinguishing scores to integer weights using a conversion function MapToWeight.

For completeness, we define the *rank* of a key. We denote the weight of the secret key **sk** as $W = \sum_{j=0}^{m-1} w_{sk^j,j}$ where we assume sk^j takes value $w_{sk^j,j}$ in the weight table.

Definition 1 (Key rank). *Given the weight table* **W**, *and the score* W *of the secret key* **sk**, *count the number of keys with a score strictly less than* W.

If multiple keys have the same scores as the secret key, we assume without loss of generality that it is ranked first. This is the worst case scenario for a device owner as this is the earliest an adversary will enumerate the key. We denote the key rank of the secret key, for a given weight table, as $\text{rank}_{\mathbf{sk}}(\mathbf{W})$.

Bounding keys. For an enumeration algorithm to be able to enumerate the B next-most likely keys given an offset O, it is necessary to determine how to identify which of the candidate keys fulfil these criteria. The solution is to use ranges of weights to define non-overlapping groupings of keys: for $W_0 < W_1 < W_2$, the ranges $[W_0, W_1]$ and $[W_1 + 1, W_2]$ define two sets of candidate keys, where all of the candidates in the first set are more likely than those in the second. Weights are used instead of keys as it avoids the issue of multiple keys having the same weight (therefore having to enumerate some keys for a given weight but not all).

Thus we arrive at a formalisation of an enumeration algorithm:

$$\{\mathbf{k}_i\}_i \leftarrow \textit{enumerate}(\mathbf{W}, W_{min}, W_{max}).$$

Given the weight table $\mathbf{W}$ and an inclusive range of weights $[W_{min}, W_{max}]$, return all keys with weight within the specified range.

Verification. Enumeration produces a list (or set) of the most likely keys, but the adversary still needs to check each one: to recover the secret key, each of the keys in the list is checked using a *verification* algorithm, which takes the set of enumerated keys and returns either the correct key or $\perp$ if the correct key is not found:

$$\mathbf{sk}/\perp \leftarrow \textit{verify}(\{\mathbf{k}_i\}_i).$$

Since side-channel attacks are typically known-plaintext attacks, verification of a single key can be performed by encrypting the known plaintext(s) under the candidate key and checking whether the output matches the given ciphertext(s). Some block ciphers will be easier to verify than others: in this work we consider the best-case verification scenario for the adversary, in which AES-128 encryptions can be accelerated using hardware-level instructions. If the targeted cipher is (for instance) DES, then the adversary must use a software implementation of the algorithm if they are to use standard non-specialised computing resources.

Search. We term the combination of both the verification and enumeration components as a key *search*:

$$\textit{search}(\mathbf{W}, W_{min}, W_{max}) = \textit{verify}(\textit{enumerate}(\mathbf{W}, W_{min}, W_{max})).$$

The total expected runtime of the full search process has to take into account both the time to enumerate the keys and the time to iterate through the enumerated list until verification returns a key. Since the latter allows an early abort, the order in which keys are checked matters (order-optimality will be discussed in more detail towards the end of the section). To maximize key recovery capability, given fixed resources, it is necessary to optimise both the enumeration and verification components for speed; after all, total run-time will depend primarily on the slowest of these two algorithms.

Parallelism. Any significant cryptanalytic effort benefits greatly from parallelism. We use a model where multiple parallel execution units (PEUs) operate as independently as possible. In this work we consider PEUs to be CPU cores, but any reasonable definition (specialised hardware, distributed computing systems) is meaningful.

Given a number of PEUs (and a table $\mathbf{W}$), we aim to minimize the expected time to recover the secret key. Parallelising the verification of the set of keys is relatively straightforward: the verification of each element in the set is an

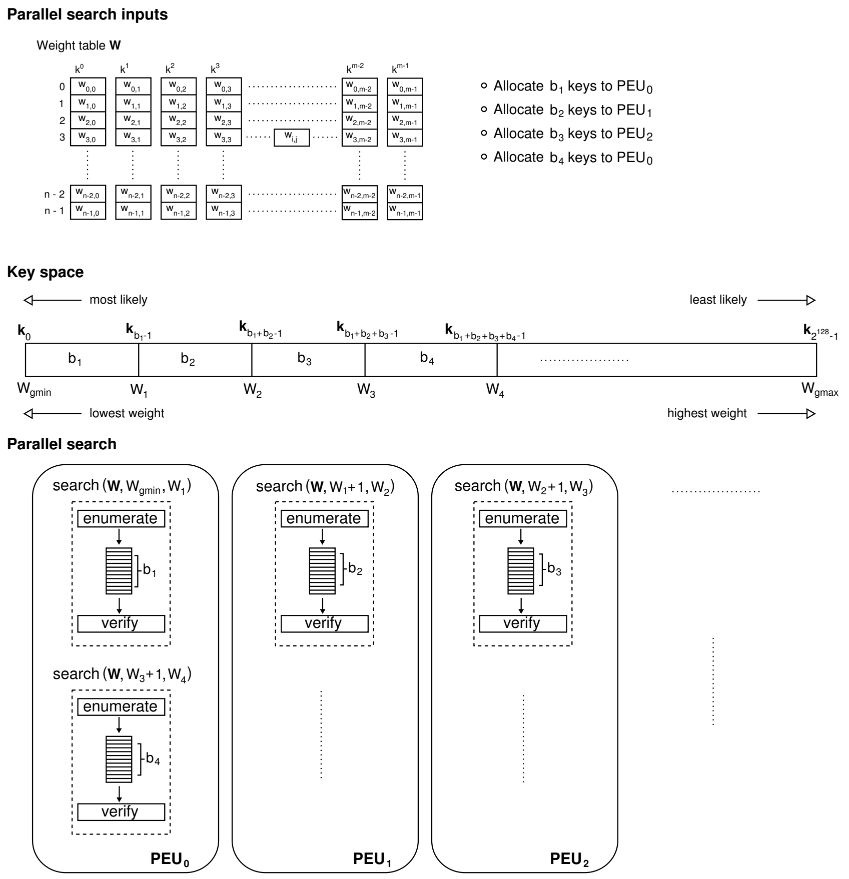
Parallel search inputs
Weight table W
Allocate b_1 keys to PEU_0
Allocate b_2 keys to PEU_1
Allocate b_3 keys to PEU_2
Allocate b_4 keys to PEU_0
Key space
most likely
least likely
lowest weight
highest weight
Parallel search
search (W, W_{gmin}, W_1)
search (W, W_1+1, W_2)
search (W, W_2+1, W_3)
search (W, W_3+1, W_4)
enumerate
verify
PEU_0
PEU_1
PEU_2

Fig. 2. Example overview of how a parallel adversary can search the most likely portions of the global key space by partitioning along ranges of weights and allocating distinct partitions to different parallel units.

independent event and thus scales linearly across increasing numbers of PEUs. For example, if we are able to verify 2^{26} keys per second on one PEU, then we will be able to verify 2^{27} keys per second using two. However, as will be discussed in Sec. 3, extending enumeration algorithms to distribute work is decidedly non-trivial.

Figure 2 illustrates how an adversary can construct a parallel search capability by dividing up the candidate key space along the range of key weights. If there are c PEUs, then the simplest strategy is to find c weight ranges that each define approximately the same number of candidate keys, distributing the workload as evenly as possible across the available parallelism. This assumes that it costs the same time to enumerate each key – in practice this may not always be the case, and consequently it may be preferable to assign different numbers of keys

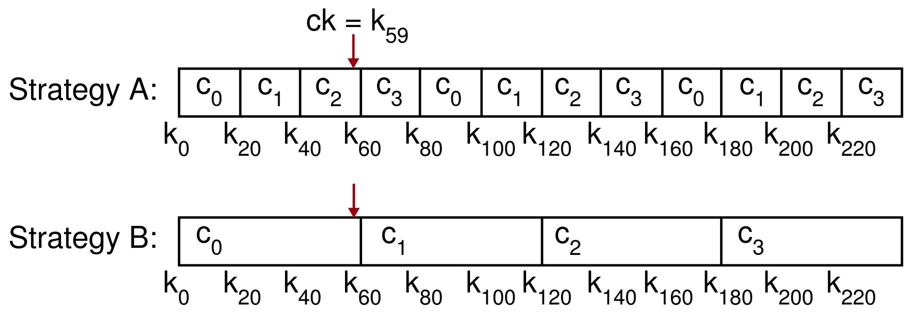
ck = k59
Strategy A:
c0 c1 c2 c3 c0 c1 c2 c3 c0 c1 c2 c3
k0 k20 k40 k60 k80 k100 k120 k140 k160 k180 k200 k220
Strategy B:
c0 c1 c2 c3
k0 k20 k40 k60 k80 k100 k120 k140 k160 k180 k200 k220

Fig. 3. Two strategies for the allocation of tasks when aiming to find $\mathbf{k}_{59}$ in a set of 240 candidate keys $\mathbf{k}_0, \ldots, \mathbf{k}_{239}$, in a parallel environment consisting of four PEUs $c_0, \ldots, c_3$.

to different search tasks, but with the overarching goal of evenly distributing the workload.

A naive approach is to divide the entire key-space by the number of PEUs, and assign that many keys to each PEU. In practice, it will be preferable to restrict the size of each individual search task. Figure. 3 shows two possible strategies for allocating search tasks in a hypothetical experiment. In this scenario there are 240 possible keys $\mathbf{k}_0, \ldots, \mathbf{k}_{239}$ and four PEUs $c_0, \ldots, c_3$. The correct key is $\mathbf{k}_{59}$, and let us assume that the adversary takes two hours to enumerate and verify 20 keys. Using strategy A, the adversary will identify $\mathbf{k}_{59}$ after 119 minutes has elapsed. Using strategy B, the adversary would have to wait 354 minutes to arrive at the result, nearly 3 times as long. This is suggestive for ensuring that the size of any one task is not too large, indicating that it is preferable for PEUs to process many smaller interleaved search tasks rather than a single large task.

Further justification for restricting the size of each search task is that if the size of the set of keys produced by the enumeration is too large then the memory footprint may become too high – for instance, a set of 2^{38} AES-128 candidates would, in the worst-case, consume 4 TiB of memory. As will be described in Sec. 3, it is possible to interleave the verification of smaller subsets of the candidates as they are produced within the enumeration algorithm. This is the approach adopted throughout this work.

A final consideration when building a parallel tool is the ratio of the number of enumeration components to verification components. In some circumstances, such as an AES-128 scenario, as explored in Sec. 4, keys can be verified extremely quickly. In these cases, it is unlikely that a enumeration algorithm can produce candidates fast enough to meet the throughput of a verification component. Alternatively, such as when attacking a cipher for which no hardware acceleration is available, the inverse scenario may occur. It will be difficult to predict the optimal ratio of enumeration to verification components prior to runtime, particularly as there would be additional synchronisation overhead. In the remainder of this work, we explore performance where each enumeration component is paired with a single verification component.

Order-optimality. Each call to an enumeration algorithm produces a list of key candidates. This list can be returned in-order or out-of-order. Given a list of B keys $k_1, \ldots, k_B$, an *order-optimal* enumeration algorithm orders the list such that if $i < j$ then k_i is more likely than k_j ($W_i < W_j$). A non-optimal enumeration algorithm must enumerate all keys with weights within a defined range, but may return the list in any order.

At first glance, the order-optimal approach appears to be more desirable, as an adversary will be able verify more likely keys before less likely ones. However, the definition does not extend to a parallel setting in a way that allows for an efficient instantiation of an algorithm – each PEU would have to synchronise to ensure the correct ordering of candidates across all enumerated lists is maintained, which becomes increasingly expensive as the number of PEUs increases. A weaker definition is achievable in which each PEU is order-optimal on the subset of the key space the PEU is enumerating, but the global synchronisation of the search across many PEUs is not required.

Building a parallel search tool. In this work we will focus on the following three areas, with the aim of building a fast and highly-parallel tool for running key enumeration:

1. Identifying fast algorithms for enumerating key candidates, in both order-optimal and non-optimal settings, described in Sec. 3.
2. Identifying fast methods for verifying 128-bit AES key candidates, described in Sec. 4.
3. Exploring and identifying challenges associated with how best to select weights for parallelism, described in Sec. 5.

The ultimate goal is to allow the adversary to search as many key candidates as possible within a fixed amount of time: informally, the more keys an adversary can enumerate, the more likely the adversary is to recover the key. Simultaneously, we would also like to maximise the number of PEUs that an adversary can deploy.

3 Accelerating enumeration

Our starting point was to identify fast enumeration algorithms. We are additionally concerned with ensuring that the memory complexity of the algorithms is kept low—if too high, this might impact on the number of PEUs that may be run on a single host.

3.1 Conversion of scores into weights.

We define a function, MapToWeight, that converts positive floating-point distinguisher scores ($0 < d_{i,j} \leq 1$) into non-negative integer weights. We require a mapping to achieve this, and also require MapToWeight to transform the raw scores such that the highest score is mapped to the lowest weight: in raw score format, the greater the score the more likely the key candidate, in the integer weight format, the smaller the weight the more likely the key candidate. This is necessary to fulfil the requirements of the some of the enumeration algorithms used throughout this work. Our starting point is the conversion function as described in [16,15]: as such, the MapToWeight function is defined as follows:

$$w_{i,j} = \lfloor 2^{p-\alpha} \cdot d_{i,j} \rfloor - \min_{l} \lfloor 2^{p-\alpha} \cdot d_{l,j} \rfloor + 1,$$

where α is the precision of the largest score and p is the required precision. The multiplication by $2^{p-\alpha}$ scales the weights to have the required precision and the floor converts the scores to integers. By the subtracting the smallest weight in each vector, the scores maintain their relationship while having the smallest value as 1. The runtime of the enumeration algorithms to be described is normally dependent on the weight, and so having a smaller weight increases efficiency.

If the precision p is less than the precision in the raw distinguishing scores, then the MapToWeight process is *compressive* – keys that may have unique scores in their raw form maybe be "compressed" together by MapToWeight, and assigned equal likelihood after the conversion is complete. This compression has an impact on the viability of an order-optimal algorithm: an order-optimal algorithm that enumerates over weights can only be order-optimal in the weights and not the underlying scores. Therefore, a higher compression ratio resulting from the MapToWeight function, the further away from being 'truly optimal' the algorithm will be.

Search tasks are defined using a range of weights, which in turn defines the number of keys to be enumerated. At higher levels of compression, the more keys will be allocated the same weight, and thus it is more difficult for an adversary to find ranges of weights for which approximately the same number of keys will be enumerated. At the extreme end under very high levels of compression, the number of PEUs that may be deployed will be restricted, even when each search task enumerates keys with a single unique weight value. This will factor into the quantity of precision required to be retained by MapToWeight.

3.2 Threshold

Our starting point is an algorithm denoted THRESHOLD. Given a weight table **W** containing additive integer weights (computed using MapToWeight) and a *threshold weight* W_t, THRESHOLD returns a list of all keys with a weight *less than* than W_T.

Algorithm 1 shows how this is achieved. Since the weights are sorted it is possible to work down each distinguishing vector, enumerating full keys, until

```
Algorithm Threshold($W_T$, $\mathbf{W}$):
$\mathbf{W}' \leftarrow sortAscending(\mathbf{W})$
$partialSum \leftarrow [0]^m$
$partialSum[m-1] \leftarrow 0$
for $j$ from $m-2$ down to $0$ do
  $partialSum[j] \leftarrow partialSum[j+1] + w'_{0,j}$
end for
$L \leftarrow \{\}$
$L \leftarrow$ThresholdRecurse($0, 0,$ "", $W_T, \mathbf{W}', L$)
return $L$

Algorithm ThresholdRecurse($j, w, key, W_T, \mathbf{W}', L$):
for $i$ from $0$ up to $n-1$ do
  $newW \leftarrow w + w'_{i,j}$
  if $newW + partialSum[j] > W_t$ then
    break
  else if $j = m-1$ then
    $K \leftarrow setKeyBytes(\{key\}, j, i)$
    $L \leftarrow L \cup \{K\}$
  else
    $nK \leftarrow setKeyBytes(\{key\}, j, i)$
    $L \leftarrow$ ThresholdRecurse($j+1, newW, nK, W_T, \mathbf{W}', L$)
  end if
end for
return $L$
```

Algorithm 1. The Threshold enumeration algorithm.

the total score is too large. The next vector is then incremented and the previous are reset. This process repeats until all keys with weight less than W_t have been enumerated. The function $setKeyBytes(L, j, i)$ takes in a list L of (partially constructed) keys and sets the j^{th} subkey (in each key) to be $\Phi_j(i)$ (where Φ_j is the permutation corresponding to the inverse of the sorting). While in this case *setKeyBytes* will only take in a single key at a time, a generalisation is required for algorithms given in the next section. *partialSum* is storing partial sums of the most likely key for a subset of the key bytes. This is used as an early termination condition, since if the most likely key can not meet the weight restrictions, it is not worth recursing further as the less likely keys won't meet the conditions either.

A key rank algorithm can be used to select the threshold W_T such that only the B most likely keys are enumerated – the weight $W_{\mathbf{k}}$ for a given key $\mathbf{k}$ can be computed through summation of the individual weights associated with each subkey value for the key. Given a key $\mathbf{k}$, if the rank of $\mathbf{k}$ as estimated by a rank algorithm is B, then the threshold W_T can be chosen to be the weight of the key $W_{\mathbf{k}}$. This achieved using a binary search and repeatedly re-ranking.

The time complexity of Threshold is $\mathcal{O}(rank \cdot \log W_{gmax} + m \cdot n \cdot \log n + m \cdot B \cdot \log n)$, where $\mathcal{O}(rank)$ is the time complexity of the chosen key rank algorithm and the weight table $\mathbf{W}$ takes $\mathcal{O}(m \cdot n \cdot \log n)$ to be sorted. A technique to be given in Sec. 5.2 can be utilised for the key rank of Martin et al. [16] to eliminate the multiplicative $\log W_{gmax}$ factor. The memory complexity is that of the key rank algorithm. In experiments that follow, we use the rank algorithm from [16] with the improvements from [15].

The primary drawback of Threshold is that it must always start enumerating from the most likely key. Consequently, whilst the simplicity and relatively strong time complexity of Threshold is desirable, in a parallelised environment it can only serve as the first enumeration algorithm (or can only be used in the first search task).

3.3 Parallelisable enumeration

To build a viable parallelisable enumeration tool, we need a complement to Threshold that can enumerate keys that have weights in a range $[W_{T_1}, W_{T_2})$, for $W_{T_1} < W_{T_2}$. With this in mind, we attempted to improve upon the enumeration algorithm of Martin et al. (denoted as MOOS from this point forward) as it has this capability [16]. The MOOS algorithm, including some improvements detailed in [15] is given in Algorithm 2, and is hereafter referred to as MOOS+.

For completeness, we provide a description of the MOOS+ algorithm. A key enumeration problem with integer weights $\mathbf{W}$ and weight boundaries W_1, W_2 can be expressed as a graph with $W_2 \cdot m + 2$ nodes (the + 2 correspond to an accept and reject node denoted A and R respectively). A node $N_{i,j}$ corresponds to subkey i having (partial weight) j. Each node has n outgoing edges, one for each value the subkey can take. For value l in subkey i the edge (labelled l) goes from $N_{i,j}$ to $N_{i+1,j+w_{i,l}}$. If the weight $j + w_{i,l}$ is greater than W_2 the edges goes to R instead. For $i = m-1$ the edges are treated slightly differently. If $j+w_{m-1,l}$ is in the range $[W_1, W_2)$ the edge goes to A, else it goes to R. The algorithm RC given in Algorithm 2 is used to calculate which node an edge goes to.

A path from $N_{0,0}$ to A corresponds to a key with weight in the range $[W_1, W_2)$, where the key is determined by the labels on the edges taken. It is these keys that will be enumerated. The algorithm starts with assigning values to the last subkey and works backwards to the first subkey (over all weights and possible value for the given subkey). Using the underlying graph structure it is possible to efficiently calculate all the keys with a single pass over all (active) nodes in the graph.

A node $N_{i,j}$ is called active if there is a path from it to $N_{0,0}$ (which stores the final answer). Only active nodes contribute to the final solution (identifying active nodes is discussed in more detail towards the end of the section).

Martin et al. [16] state that the time complexity of their algorithm is $\mathcal{O}(m^2 \cdot n \cdot W \cdot B \cdot \log n)$. It is argued that this is the case because all $m \cdot n \cdot W$ nodes in the graph are visited and for each node at most B keys have a new value appended to them, where each key has length $m \cdot \log n$. However as the algorithm is described it is possible n^{m-1} keys are stored and then only narrowed down to the required

```
**Algorithm** MOOS+($m, n, W_1, W_2, \mathbf{W}$):
for $j$ from $m-1$ down to $0$ do
  for $w$ from $0$ up to $W_2 - 1$ do
    for $i$ from $n-1$ down to $0$ do
      $K[w] \leftarrow K[w] || setKeyBytes(Old[RC(j, w, i, W_1, W_2, \mathbf{W})], j, i)$
    end for
  end for
  $Old \leftarrow Keys$
end for
return $K[0]$

Algorithm RC($j, w, i, W_1, W_2, \mathbf{W}$):
if $w + w_{i,j} > W_2$ then
  return Reject
else if $j = m - 1$ then
  if $w + w_{i,j} < W_1$ then
    return Reject
  else
    return Accept
  end if
else
  return $(j+1, 0, w + w_{i,j})$
end if
```

Algorithm 2. The Martin et al. algorithm [16] plus improvements as detailed in [15].

B keys in the last distinguishing vector. Consider the case where all partial keys, without a value for the first distinguishing vector, are valid and it is only considering the last distinguishing vector that narrows the keys to the required B keys. This gives a time complexity of $\mathcal{O}(m^2 \cdot n^m \cdot W \cdot \log n)$. To achieve the time complexity expressed within the paper, only the B most likely partial keys are stored at each stage.

Improving time and memory complexity The time and memory complexity of the MOOS algorithm is dependent on the data structure and method used to store the partial key candidates as they are enumerated. The naive method is to use a list containing each (partial) candidate. Martin et al. give an adjustment to the way keys are stored (using a tree) to reduce the memory footprint [16]. This strategy also improves the time complexity of the algorithm: the result has time complexity $\mathcal{O}(m \cdot n \cdot W \cdot \log n + B \cdot m \cdot \log n)$. The first term occurs because each node in the graph is touched once and a single value (between 0 and $n-1$) is written. The second term corresponds to taking the tree of key values and converting them into keys to be tested. There are B keys stored within the tree and each key contains $m \cdot \log n$ bits, giving the desired result. Alongside the initial Martin et al. algorithm MOOS, these optimisations allow us to define two

```
**Algorithm** ActiveNodeFinder($W, \mathbf{W}$):
$activeNodes \leftarrow [\{\}]_{j=0}^{m-1}$
$activeNodes[0] \leftarrow \{0\}$
**for** $j$ **from** $1$ **up to** $m-1$ **do**
  **for** $i$ **from** $n-1$ **down to** $0$ **do**
    **for** $w$ **in** $activeNodes[j-1]$ **do**
      **if** $w + w_{RC(j,w,i),j} < W$ **then**
        $activeNodes[j] \leftarrow activeNodes[j] \cup \{w + w_{RC(j,w,i),j}\}$
      **end if**
    **end for**
  **end for**
**end for**
**return** $activeNodes$
```

Algorithm 3. Algorithm to calculate the active nodes in graph.

further algorithms: List, using lists to store partial candidates, and Forest, using tree-like data structures.

Finding active nodes The MOOS paper touches on the fact that any node that does not contribute to the final answer can be skipped. We provide an efficient algorithm to calculate the graph nodes that must be computed on. Whilst skipping of nodes does not change the asymptotic time (or memory) complexity, in practice it may provide a performance benefit. The algorithm is given in Algorithm 3. Since only nodes whose partial keys will be enumerated are visited, each node can contain at most B keys. Consequently, this solution fixes the issue with the initial algorithm given by Martin et al. whilst also increasing the efficiency. Since the active nodes do not vary based on the underlying memory structure used by the algorithm, this improvement can be incorporated irrespective of how the partial key candidates are stored in memory, allowing for the construction of two further algorithms: ANF/List and ANF/Forest, denoting that the list and forest data structure versions are paired with the active node finder optimisation.

Using node parents A final observation is that instead of storing the partial keys within the graph, the parents of each node can be stored instead. While a node has at most two children, it can have at most $n \cdot W$ parents. In the algorithm, when a node is visited, its right child is calculated and the node's index is stored within the right child. This has memory requirement $\mathcal{O}(n \cdot m \cdot W)$, instead of the originals complexity of $\mathcal{O}(B \cdot W)$. Depending on the number of keys enumerated and the implementation being attacked, this algorithm may be more desirable. Given the resulting memory structure keys can be computed by following the parents from the accept node to the start node to construct valid keys. This allows for one final candidate, ANF/Backtracking. Due to the backtracking technique, it is vital that only valid paths are considered and thus this algorithm can not be constructed without ActiveNodeFinder.

3.4 Order-optimal enumeration

The algorithms described so-far meet the weaker definition of enumeration: the B most likely keys are enumerated but the order in which they are produced is arbitrary. In the following section we consider ways of producing order-optimal variants of the algorithms discussed so far.

The THRESHOLD algorithm cannot be converted into an order-optimal form without significant modification. However, there is potential to convert the various modifications to the MOOS algorithm. An initial starting point is to replace the list of partial keys, stored at each node, with a priority queue. A priority queue is a data structure which allows the minimum value to be removed efficiently, and so by repeatedly removing the key with the smallest weight we are able to test the keys constructed by the algorithm in an optimal order. Using a priority queue increases the time complexity by a factor of $\mathcal{O}(\log B)$. The partial key is added to the priority queue with weight $w + w_{i,j}$, where w is the weight of the current node. When the partial keys have their value updated the weight does not change. This is because the weight originally selected will be the weight of the completed key and thus does not need to be adjusted. This allows us to define an order-optimal priority-queue based algorithm ORDEREDQUEUE, and an equivalent using the ActiveNodeFinder optimisation ANF/ORDEREDQUEUE.

The priority queue adds some algorithmic overhead. A more algorithmically efficient solution is to observe that, after the MOOS enumeration algorithm has finished running, $K[0]$ contains all keys with weight less than W. However, more generally $K[i]$ contains all keys with weight less than $W - i$ (the subgraph without the first i rows is exactly the graph when the algorithm is called with weight $W - i$). Hence, by testing keys from $K[W - 1]$ to $K[0]$ the keys will be enumerated in order of likelihood. The downside is that keys will be tested multiple times. A key with weight less than $W - 1$ will appear in the sets $K[1]$ and $K[0]$ and thus will be tested twice. To avoid testing keys multiple times the right child function must be modified to only accept keys with weight *exactly* $W - 1$. This restriction means that all keys will appear in the 'top row' of the graph at most once. Enumerating the keys from $K[W - 1]$ to $K[0]$ enumerates all keys optimally. This variation does not change the time complexity of the original algorithm (regardless if the data-structure is a list or tree). This allows us to define four new order-optimal algorithms (using a list or tree, and with or without ActiveNodeFinder): ORDEREDLIST, ANF/ORDEREDLIST, ORDEREDFOREST and ANF/ORDEREDFOREST.

3.5 Comparison of enumeration algorithms

We now have multiple algorithms to compare: the THRESHOLD algorithm and the variations of the MOOS+ algorithm using different underlying memory structures, each of which can be deployed with or without the active node finder optimisation. Table 1 summarises the time and memory complexity of the candidate algorithms we considered in this research. To best explore the real-world performance of these, and since some of the improvements do not affect either

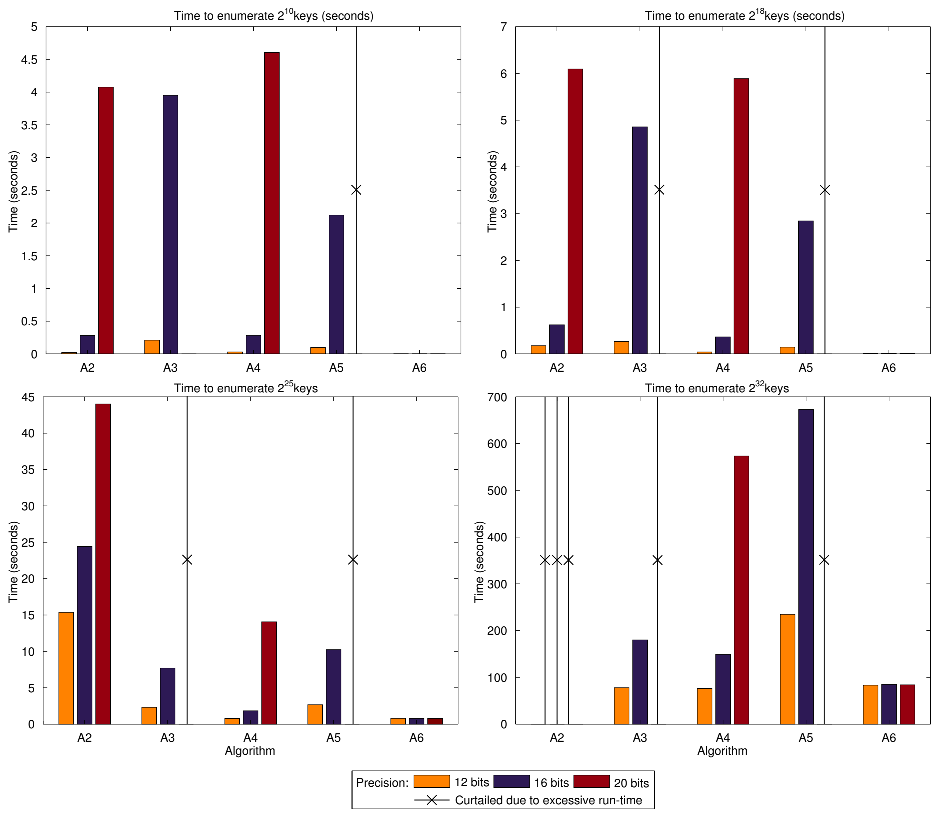

Fig. 4. Benchmarking information for candidate enumeration algorithms using on a single core of a Xeon E5-1650 v2 CPU. Algorithms listed in Table 1 that are not shown were not benchmarked due to excessive run-time.

the asymptotic memory or time complexity, we will investigate the algorithms empirically. To support the replication of our results, the full configuration used to generate the following data is outlined in Appendix A.

Enumeration performance Figures 4 and 5 give a speed and memory comparison of the non-order-optimal enumeration algorithms (respectively). The run-time was measured as the time taken to enumerate the first B keys, for $B = 2^{10}, 2^{18}, 2^{25}$ and 2^{32}. As many of the algorithms have a run-time dependent on the weight parameter, we also controlled for the amount of precision used in the MapToWeight process, re-running each algorithm using attack results converted using 12, 16 and 20 bits of precision. The run-time experiments were performed on a single core of a Xeon E5-1650 v2 CPU, and averaged over 50 repetitions. We are not concerned with the absolute value of the observed times as this is subject to the particular hardware and the quality of the implementation, and instead focus of the degree of relative performance between the

Algorithm	**ID**	**Time Complexity**
Threshold	A6	$\mathcal{O}(rank + m \cdot n \cdot \log n + m \cdot (O+B) \cdot \log n)$
MOOS [16]	A0	$\mathcal{O}(m^2 \cdot n^m \cdot W_{\mathbf{k}_{O+B}} \cdot \log n)$
List	A1	$\mathcal{O}(m^2 \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
ANF/List	A2	$\mathcal{O}(m^2 \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
Forest	A3	$\mathcal{O}(m \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot \log n + m \cdot B \cdot \log n)$
ANF/Forest	A4	$\mathcal{O}(m \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot \log n + m \cdot B \cdot \log n)$
ANF/Backtracking	A5	$\mathcal{O}(m \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot \log(m \cdot n \cdot W_{\mathbf{k}_{O+B}}) + m \cdot B \cdot \log n)$
OrderedQueue	OA1	$\mathcal{O}(m^2 \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n \cdot \log B$
ANF/OrderedQueue	OA2	$\mathcal{O}(m^2 \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n \cdot \log B$
OrderedList	OA3	$\mathcal{O}(m^2 \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
ANF/OrderedList	OA4	$\mathcal{O}(m^2 \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
OrderedForest	OA5	$\mathcal{O}(m \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot \log n + m \cdot B \cdot \log n)$
ANF/OrderedForest	OA6	$\mathcal{O}(m \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot \log n + m \cdot B \cdot \log n)$
Algorithm	**ID**	**Memory Complexity**
Threshold	A6	$\mathcal{O}(rank)$
MOOS [16]	A0	$\mathcal{O}(m \cdot n^m \cdot W_{\mathbf{k}_{O+B}} \cdot \log n)$
List	A1	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
ANF/List	A2	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
Forest	A3	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
ANF/Forest	A4	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
ANF/Backtracking	A5	$\mathcal{O}(m \cdot n \cdot W_{\mathbf{k}_{O+B}} \cdot \log(m \cdot n \cdot W_{\mathbf{k}_{O+B}}))$
OrderedQueue	OA1	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
ANF/OrderedQueue	OA2	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
OrderedList	OA3	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
ANF/OrderedList	OA4	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
OrderedForest	OA5	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
ANF/OrderedForest	OA6	$\mathcal{O}(m \cdot W_{\mathbf{k}_{O+B}} \cdot B \cdot \log n)$
Algorithm	**ID**	**Supports enumeration with offset**
Threshold	A6	×
MOOS [16]	A0	✓
List	A1	✓
ANF/List	A2	✓
Forest	A3	✓
ANF/Forest	A4	✓
ANF/Backtracking	A5	✓
OrderedQueue	OA1	✓
ANF/OrderedQueue	OA2	✓
OrderedList	OA3	✓
ANF/OrderedList	OA4	✓
OrderedForest	OA5	✓
ANF/OrderedForest	OA6	✓

Table 1. Properties of the candidate enumeration algorithms. To enumerate between keys with rank O and $O+B$, the weights corresponding to these keys ($W_{\mathbf{k}_O}$ and $W_{\mathbf{k}_{O+B}}$) are input into the algorithms.

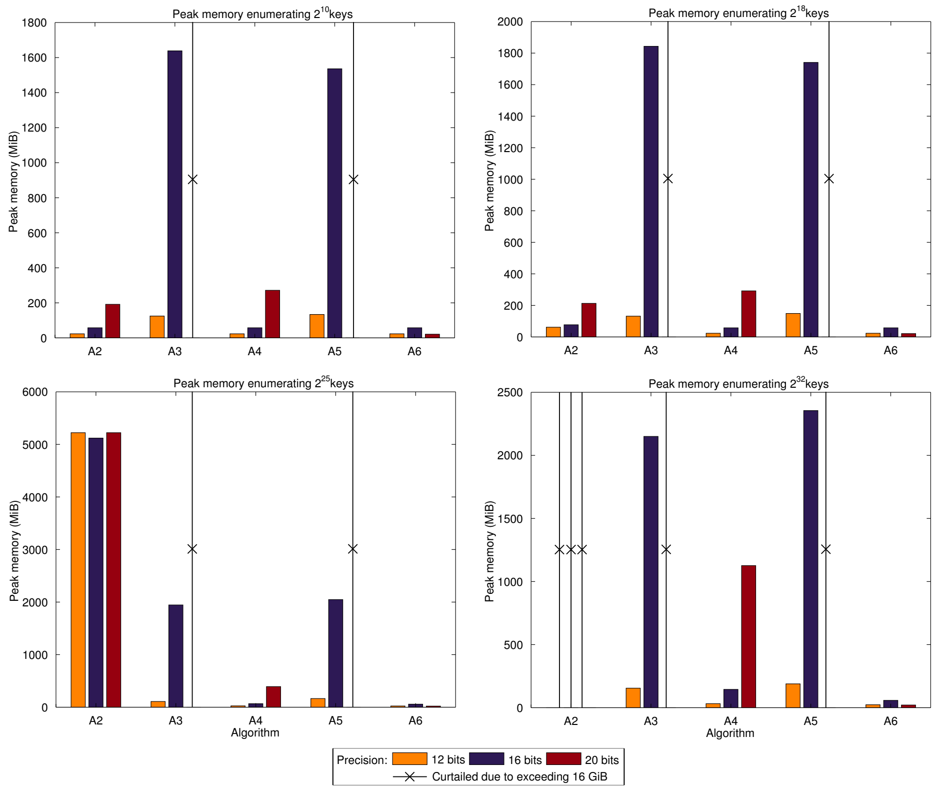

Fig. 5. Memory usage benchmarking information for candidate enumeration algorithms using on a single core of a Haswell i7-4770K CPU. Algorithms listed in Table 1 not shown were not benchmarked due to excessive run-time and/or memory usage beyond the 16 GiB available on the host.

algorithms. The memory experiments were performed on a machine with 16 GiB RAM, and we used the valgrind tool Massif with the `pages-as-heap` option set to record the peak memory usage.

Order-optimal enumeration performance Figures 6 and 7 give a speed and memory comparison of the order-optimal enumeration algorithms (respectively). An immediate observation is that the overhead of supporting order-optimality is significant, with measured run-times of the algorithms far exceeding that of the non-order-optimal equivalents. As the number of keys in a single enumeration task decreases, the parallel system will begin to approach the weaker definition of order-optimality. Consequently, by selecting a suitable parallelisation strategy (with reference to Fig. 3), and adversary can achieve a more desirable trade-off in terms of run-time. As such, for the remainder of this work, we no longer consider order-optimal algorithms.

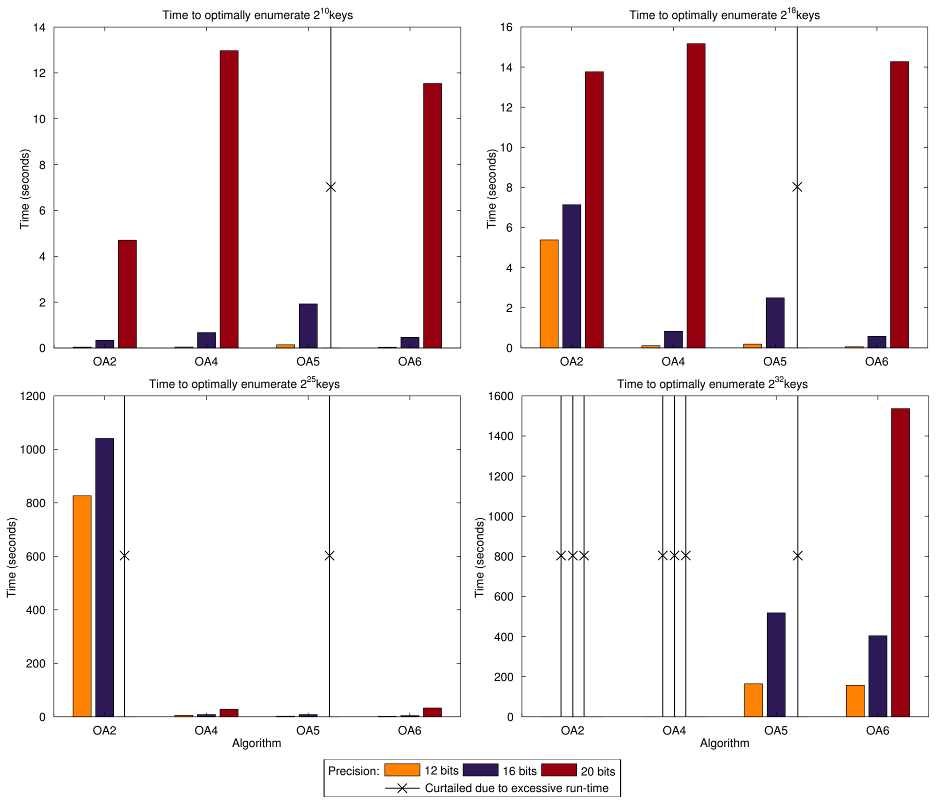

Fig. 6. Benchmarking information for candidate order-optimal enumeration algorithms using on a single core of a Xeon E5-1650 v2 CPU. Algorithms listed in Table 1 that are not shown were not benchmarked due to excessive run-time.

Evaluation It can be seen that the two most desirable algorithms for enumeration are Threshold and ANF/Forest. Threshold is by far the most efficient of the algorithms, both asymptotically and in practice. However, it can only be run once and can only enumerate the first batch of most likely keys. ANF/Forest is the best candidate to run in tandem with Threshold; it manages to both minimise run-time and memory usage for at all levels of effort and MapToWeight precision.

4 Accelerating verification

As described in Sec. 2, the adversary needs to be able to verify the correctness of candidate keys as quickly as they are enumerated. In most cases, to verify a candidate requires a check to see if the encryption of a known plaintext under the candidate key matches a known ciphertext. In some SCA attacks, the attacker may for instance recover the final round key of AES. To verify keys in this

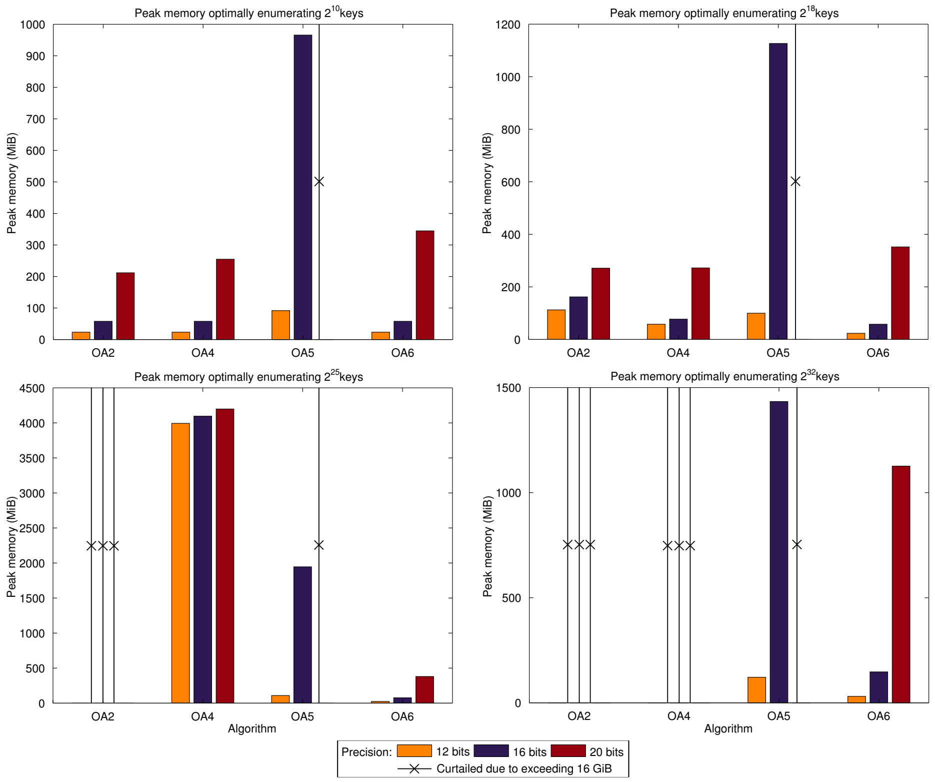

Fig. 7. Memory usage benchmarking information for candidate order-optimal enumeration algorithms using on a single core of a Haswell i7-4770K CPU. Algorithms listed in Table 1 not shown were not benchmarked due to excessive run-time and/or memory usage beyond the 16 GiB available on the host.

scenario, the adversary must do additional work by first reversing the AES key expansion before performing the actual encryption check.

The process of key verification can be considered utilitarian to the problem of enumeration, however, the result of accelerating verification is very much congruent to the overall attack goal (i.e., minimising time required to achieve key recovery). Whilst enumeration is largely independent of the underlying 'target algorithm' (requiring only independence of subkeys), the verification stage is intrinsically dependent on it. In this section we discuss how best to leverage the hardware available for maximising key verification throughput by making extensive use of the micro-architectural features at our disposal. A typical workstation will feature specialised instructions with the explicit purpose of acceleration common cryptographic operations. If we consider the specific case of AES-128, this is true for any workstation featuring an Intel ($\geq$ Westmere) or AMD ($\geq$ Bulldozer) processor. We focus on the former platform in attempting to accelerate verification using the Intel AES-NI instructions [9]. Our strategy is two-fold: we first

Instruction	Latency	Reciprocal Throughput
`aesimc`	12	2
`aesenc`	7	1
`aesdec`	7	1
`aesenclast`	7	1
`aesdeclast`	7	1
`aeskeygenassist`	10	2

Table 2. AES-NI instructions and corresponding performance on the Sandy Bridge microarchitecture.

attempt to minimise the number of instructions required to perform verification of a single key candidate. We then attempt to maximise pipeline occupancy via time-sliced parallelism over multiple candidates.

4.1 Minimising work

Intel benchmarks a vanilla AES-NI accelerated AES-128-ECB encryption application parallelising over 4 data blocks (64 bytes) with an average throughput of 1.28 cycles/byte. This figure relies on pre-computation of the key expansion and the round keys residing in the register space, we hence assume this figure approximates the best throughput achievable for key verification. The AES-NI instruction performance metrics (see Table 2) suggest they were designed to reflect a typical use-case (i.e., encryption/decryption under a single key) and not inherently optimised for key verification. This is primarily attributed to the `aeskeygenassist` instruction that facilitates key expansion operations, specifically, it performs the required `SubBytes`, `RCON` and rotation functions to the first 32-bits of the key as part of the AES `ScheduleCore`. Albeit convenient, the instruction has a high latency (relative to other associated AES-NI instructions). We opt instead to leverage the `aesenclast` instruction in combination with computing multiple key expansions as proposed by Gstir and Schläffer [8] and improved by Bogdanov et al. [3]. We develop on their approach to further minimise the number of associated instructions per AES round as shown in Pseudocode 1[2]. We note that in a pure brute-force setting it is possible to optimise the key verification algorithmically by computing on partial differential trails as described by Bogdanov et al. [2]. However, this is largely not applicable in our setting as the enumeration process has an implicit ordering that is independent of the key structure.

4.2 Filling the pipeline

Taking into account the instruction latency and associated reciprocal throughput for the AES-NI extensions, the proposed pseudocode does not make efficient use

[2] The pseudocode is intended to illustrate structure only: a working implementation will need to consider the RCON overflow in round 8 and special structure of the final round.

```
Pseudocode: Key verification of 4 keys.
Input: keys[4], data[1]
RC ← {0x00010000,0x00010000,0x00010000,0x00010000}
MS ← {0x0906030C,0x05020f08,0x010E0B04,0x0D0A0700}
rkeys[0–3] ← transpose(keys)
state[0–3] ← keys[0–3] ⊕ data
for i from 1 up to 10 do
    tmp ← aesenclast(rkeys[3], RC)
    tmp ← shufflebytes(rkeys[3], MS)
    RC ← shiftleft(RC, 1)
    rkeys[0] ← rkeys[0] ⊕ tmp
    rkeys[1] ← rkeys[1] ⊕ rkeys[0]
    rkeys[2] ← rkeys[2] ⊕ rkeys[1]
    rkeys[3] ← rkeys[3] ⊕ rkeys[2]
    keys ← transpose(rkeys)
    state ← aesenc(rkeys)
end for
```

Pseudocode 1. Verification using AES-NI.

Implementation	Processor		
	E5-1650	E5-2670	i7-4790
Intel ref [9]	34.32	28.57	28.85
Intel on-the-fly 4x [9]	5.04	4.65	4.45
Bogdanov et al. [3] 4x	2.72	2.59	2.97
Pseudocode 1 4x	2.52	2.42	2.65
Pseudocode 1 8x	2.01	2.11	2.32
Pseudocode 1 8x (interleave + prefetch)	1.76	1.89	2.26

Table 3. Key verification single-core throughput in cycles/byte (as per [9]) averaged over 10,000 trials.

of the processor pipeline. A bottleneck originates in the dependency chain generated during the key expansion operations and propagates into the encryption operations resulting in lost cycles whilst the processor waits for the round keys to become available. To mitigate these dependencies, we follow the Intel performance guidelines to effectively compute on 8 key candidates in parallel. The parallelism is achieved by carefully interleaving the key expansion operations with round operations.

4.3 Memory caching

A keen reader will notice that parallelising over 8 key candidates has effectively exhausted the register space for holding the set of cipher states and keys. In fact, we do indeed spill into memory and store portions of the state information temporarily out of register space. To utilise memory without a significant performance penalty we rely on the L1 data cache (~ 5 cycle delay) and, once again, interleave the AES-NI operations with the memory operations to ensure the pipeline does not stall. This is of course not without risk; data in the cache may be evicted at any point. We attempt to manage the risk by way of memory management hint instructions (i.e., `prefetcht0`) to minimise the chance of a cache miss.

Table 3 compares the performance throughput of our proposed implementation with previous literature. Whilst the degree is somewhat platform dependent, we do observe a consistent improvement over previously proposed optimisations.

5 Parallelism

There is sufficient data in Sec. 3 to conclude that the optimal arrangement for our parallel enumeration tool is to combine the Threshold and ANF/Forest algorithms; the first search task uses Threshold, and the remainder use ANF/Forest. If the global minimum and maximum weights assigned to key candidates after the weight conversion process are W_{gmin} and W_{gmax}, then Threshold must enumerate keys with weight from W_{gmin} up to W_T, where W_T is chosen such that the number of candidate keys with weights in this range is *not too large*. In parallel, ANF/Forest can enumerate candidate keys with weight greater than W_T. Pertinent questions are now to understand how best to choose the threshold W_T given to the Threshold, how much precision to retain in the MapToWeight process, and how best to allocate ranges of weights to ANF/Forest to be run in parallel.

5.1 The role of weight

The level of precision retained in the weight conversion function MapToWeight is the key to leveraging the amount of parallelism available to an adversary – it determines the maximum number of PEUs the adversary can deploy.

Recall the MapToWeight process for converting floating-point distinguishing scores to integer weights is *compressive* as described in Sec. 3.1. The level of this compression controls the degree of parallelism available. If the global minimum and maximum weights assigned to keys after the conversion process are W_{gmin} and W_{gmax}, then the maximum number of PEUs that can be used is $W_{gmax} - W_{gmin} + 1$, as each search task must be assigned at least one unique weight value w, for $W_{gmin} \leq w \leq W_{gmax}$. If the level of compression is too large (the bits of precision retained is low), then the adversary will not be able to put sufficient search tasks in flight to use all the available PEUs. There is a secondary disadvantage resulting from a high level of compression – the number of keys with the same weight may potentially become extremely large. Taking for instance AES-128, and a precision of 6 bits, there are 2^{128} possible keys and only $16 \cdot 2^6$ possible weight values a key can be assigned to. Therefore even if the first PEU only enumerates keys with weight W_{gmin}, in the worst case scenario the

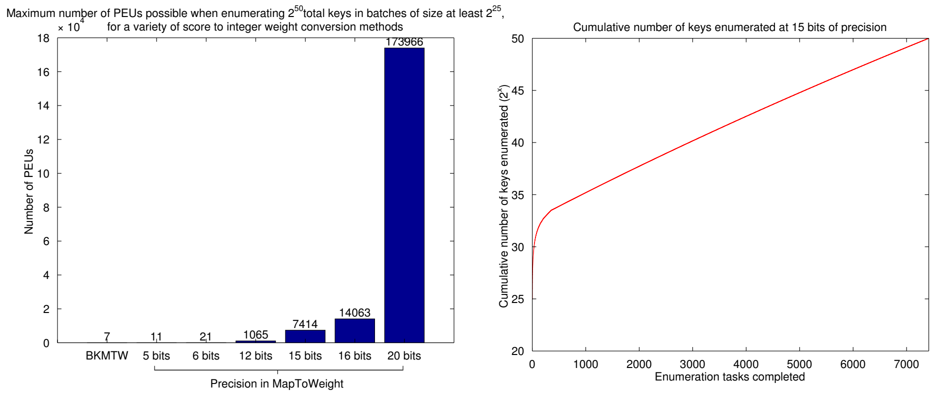

Fig. 8. (Left) The maximum number of PEUs available when MapToWeight function is run at varying degrees of precision and when the adversary wishes to enumerate the first 2^{50} most likely keys. (Right) The cumulative number of keys enumerated as single enumeration algorithm invocations complete, when MapToWeight is supplied with 15 bits of precision.

second core may have to begin enumerating keys with a very low rank, reducing the effectiveness of the parallelism.

There is a secondary implication: once the conversion of scores to weights is applied, it may not be possible to select a range of weights such that approximately the same number of keys will be enumerated by multiple PEUs (in essence, making it difficult to evenly distribute work). This is further evidence for the dangers of running a high level of compression.

However, running too low a level of compression can be harmful—the run-time of ANF/Forest is linear in W, and thus the time for any single invocation of the enumeration algorithm to complete will increase. Reconciling the goals of minimising the run-time of an enumeration algorithm and increasing parallelism is now difficult: the more precision we retain in MapToWeight, the more PEUs can be used, but at the cost of increasing the sequential run-time of each search task.

To illustrate this relationship between precision and the degree of parallelism, we used a sample SCA attack and calculated the maximum number of PEUs that could be run in parallel when the MapToWeight function is run at varying degrees of precision, and when the adversary wishes to enumerate the 2^{50} most likely keys. We also compare an alternative conversion process to MapToWeight termed BKMTW, which is fully described in Bogdanov et al. [3].

The left-hand side of Fig. 8 shows how the level of precision affects the degree of deployable parallelism. We can observe that when MapToWeight uses fewer than 12 bits of precision that the number of available PEUs is heavily restricted. The process detailed by Bogdanov et al. does poor job at facilitate parallelism, proving to be more compressive than MapToWeight at 5 bits of precision[3]. The right-hand side of the figure illustrates how the workload is distributed at 15 bits of precision, showing how the number of keys assigned the same weight increases as the depth of the keys increases.

Balancing these concerns is non-trivial. Figure 8 gives important evidence, suggesting that the level of precision should be at least 12 bits (and that the technique of [3] is flawed). The prior thought experiment of Fig. 3 suggests that going higher than 12 bits is beneficial. Using the benchmarks calculated in Sec. 3 it can be seen that 20 bits of precision results in a significant overhead, and thus we propose compromise to use around 15-18 bits of precision. It may be possible to adjust the weight for different search tasks: having higher precision for tasks that enumerate deeper keys to increase the available parallelism, and lower weights for earlier tasks to improve the runtime. However, care must be taken to perform this in a manner that does not skip over keys or result in keys being tested multiple times. Therefore, for the remainder of this work we will consider the same precision for all tasks.

A final concern is how best to choose how many key candidates to allocate to the initial search task that uses the Threshold algorithm. The algorithm is significantly faster than the alternatives, and thus it seems logical to assign as many keys as possible. However, eventually one would arrive at a similar problem as with strategy B in Fig. 3 – if the Threshold algorithm will enumerate an extremely large portion of the key space, and the correct key is at the end of that portion, then the adversary will have to wait longer than if a smaller portion was allocated and more parallel instances of ANF/Forest were used instead. The suggested strategy for an adversary would be to assign only as many keys to the initial task as a function of the maximum length of time they are prepared to wait for a result. Fortunately, the run-time of the algorithm is approximately linear in the number of keys it is expected to enumerate, and thus it is possible for an adversary to extrapolate the expected run-time from small trial runs.

5.2 How to allocate weights

We have proposed parallelising by dividing up the total key space into smaller portions each defined by a range of weights, but so far have not explained how an adversary can determine the appropriate weight values. Martin et al. [16] recommend choosing a weight, ranking using the weight[4] and then binary searching in this manner to find a suitable weight for a desired enumeration effort. This will use $\mathcal{O}(\log W_{gmax})$ calls to the rank function. We are able to reduce the number

[3] The suggested process is to take $\lfloor 50 \cdot |d| + 0.5 \rfloor$ for a floating-point score d. This has several issues; the scores are not distributed evenly, and perhaps more importantly the process does not convert multiplicative scores to additive weights – the enumeration algorithm of [3] assumes multiplicative distinguishing tables but requires *additive* scores.

[4] In the rank algorithm of Martin et al. [16], finding the rank of a weight is equivalent to finding the rank of a key with that weight.

of calls to the rank algorithm by using the algorithm only once on the maximum global weight W_{gmax}, and by returning the top row of the resulting graph. Position i in the resulting array contains the number of keys with weight less than $W_{gmax} - i$. This is due to the fact that the sub-graph without the first i rows, is exactly the graph generated by the algorithm when the weight $W_{gmax} - i$ is used. This array can the be binary searched over to choose the correct weight for the enumeration effort, using only a single call to the key rank algorithm.

The same observation can be used to reduce the number of calls to the ActiveNodeFinder algorithm. Due to the structure of the algorithm, the "active columns" (within the weight limit) are almost identical. By computing the active columns for weight W_{gmax} all other weights will have a set of active columns which is a subset of this active set. However, it is more efficient to test a few extra columns per weight than it is to recalculate the ActiveNodeFinder algorithm for every weight. Thus we use a single call to the ActiveNodeFinder algorithm for all calls to the enumeration algorithm.

5.3 Related work

The work most closely aligned with the goals of our own tool is that of Bogdanov et al. from SAC 2015 [3]. In addition to exploring how best to verify AES-128 keys, they describe an enumeration algorithm called SKEA that requires the conversion of distinguisher scores to integer weights. Similarly to the algorithms we explored, SKEA is parallelisable along the weight dimension. As already discussed in Sec. 5.1, their algorithm requires an extremely high level of compression in the conversion process, resulting in a very low level of parallelism comparable to running MapToWeight at less than 5 bits of precision.

It would be possible to replace their conversion process with MapToWeight, but it is unclear as to how this would affect the run-time complexity. The asymptotic complexity of SKEA is dependent on the maximum weight W_{gmax}, but there is no further asymptotic or empirical information available as to how the run-time of SKEA scales with increasing W other than the recommendation to run at very low precision. The only real point of comparison that can be made is to compare the run-time of our ANF/Forest at 5 bits of precision. Bogdanov et al. report a enumeration throughput of $2^{25.68}$ keys per second on a single core of an i5 2400 CPU. Averaged over 200 experiments, the enumeration throughput of ANF/Forest on an i7 4770K CPU when enumerating the first 2^{28} most likely keys at 5 bits of precision was $2^{25.92}$ keys per second and $2^{25.85}$ keys per second at 6 bits of precision.

Approximately at the time of the submission of this work, Poussier et al. [17] published a key enumeration algorithm based on the key rank algorithm of Glowacz et al. [7]. The algorithm uses histogram convolution techniques to enumerate keys with a certain weight. Figure 7 in the referenced paper suggests that their approach can enumerate 2^{32} keys in approximately 100 to 1000 seconds (exclusive of the cost of verification) at an equivalent of MapToWeight using 11 bits of precision, using an i7-3770 CPU. Our best comparison is in the bottom-right hand quadrant of Fig. 5, which shows that ANF/Forest (A4) and Threshold (A6) can enumerate 2^{32} keys in under 100 seconds at 12 bits of precision, suggesting that ANF/Forest (A4) and Threshold (A6) are more efficient.

6 A tool for key enumeration

Using the performance improvements and experimental results from the previous sections, we have developed a tool named labynkyr capable of performing highly-parallelised searching of the key space, using any number of CPU cores and any number of compute nodes as PEUs. We will make this code publicly available to allow side-channel evaluators to better assess the strength of their adversaries.

6.1 Implementation

labynkyr is written in C++11 and uses threading to schedule search tasks defining small numbers of keys to search between CPU cores within a single compute node. The allocation of search tasks over multiple nodes can be precomputed (running a rank algorithm as per Sec. 5.2 and binary searching) if the total number of nodes and their associated CPU core counts is known, and so no real inter-node communication is necessary. This allows us to build a scalable tool that is optimal to run in both single-machine workstation and supercomputing cluster environments without any additional effort, as well as across highly distributed systems such as compute clouds.

In high-level abstract terms, labynkyr consists of two main modules: an "allocation" component to be run once, that finds the best allocation of portions of the candidate key space (search tasks) across the available parallelism, and a "search" component, run once per compute node, that enumerates and verifies keys given the list of search tasks assigned to that node.

Allocation A function `allocate(node_count, cores_per_node, budget_offset, max_budget)` which takes in a number of compute nodes, the number of CPU cores each node has, the total desired enumeration budget B (e.g to enumerate 2^{50} candidates), and a starting budget offset O (e.g skip enumerating the first 2^{30} keys) to allow for a resumption capability. The enumeration budget can be set to the entire key space if the adversary wishes to run indefinitely. The allocation component calculates the minimum and maximum weights associated with each incremental batch of keys to search, and distributes these tasks evenly across the requisite number of nodes and cores.

Search A function `search(search_tasks, maximum_time)` which is run in a new process once per unique compute node. This call instantiates a queuing system and one worker thread per core. Each worker thread listens for new search tasks of keys to enumerate and verify. When a search task is queued, a worker thread enumerates *and* verifies all the key candidates. The search process can run indefinitely and attempt to execute all tasks necessary to search all keys below a specified budget, or can be given a time threshold in seconds after which all worker threads will cease computing tasks. Giving, as input, a `maximum_time` of zero seconds allows the node to run until all allocated search tasks are completed. If the allocation function is called with an excessively large budget, then LABYNKYR will effectively run indefinitely until the time exceeds the maximum. LABYNKYR outputs the number of keys searched over time and their associated weight ranges, and so a resumption capability is easily achieved by re-running the allocation component with the `budget_offset` set to the total number of keys searched in the first run.

6.2 Benchmarking

We tested the performance of our tool in two scenarios. In the first, we attempted to see how many keys we could search in a single day using a standard workstation, and in the second we attempted to see how fast we could search a large number of keys in a highly-parallelised environment using a supercomputing cluster.

We utilise a interesting real-world data attack described by the authors of Longo et al. at CHES 2015 [11] to illustrate how to integrate enumeration using a Differential Power Analysis (DPA) attack on a complex device. We selected the most challenging scenario listed in the CHES 2015 paper: an attack on a hardware AES implementation utilising EM measurements. We associated each enumeration algorithm instance with the AES-128 verification technique outlined in Sec. 4.

Single workstation benchmark. In the first benchmark we aimed to explore how many keys an adversary could reasonably expect to search on a single workstation in a 24 hour period. The workstation in question consisted of a hexa-core Ivy Bridge-EP Intel Xeon E5-1650v2 CPU and 32 GiB of 1600 MHz DDR3 RAM.

The DPA attack in question used 41,300 EM traces and we converted the distinguishing vectors into integer weights using MapToWeight at 15 bits of precision. The rank of the correct key was approximately $2^{44.99}$, and we allocated search tasks using a budget $B = 2^{50}$, an amount we would not expect to be able to cover in a single day. The effort was distributed evenly by LABYNKYR across six cores, with the initial search task (to be executed using the THRESHOLD algorithm) set to cover the first 2^{36} key candidates, and the remaining tasks consisting of at least 2^{30} candidates to be executed using the ANF/FOREST algorithm.

In the 24 hour period the workstation searched a total $2^{41.825}$ keys. Figure 9 illustrates how the keys were searched over time. The large peak in the middle and bottom graphs is associated with the instance at which the keys enumerated by the THRESHOLD algorithm were verified. Of most interest is how the time spent in each task increases as the rank of the keys tested increases, in tandem with the number of keys that must be enumerated in a given task increasing.

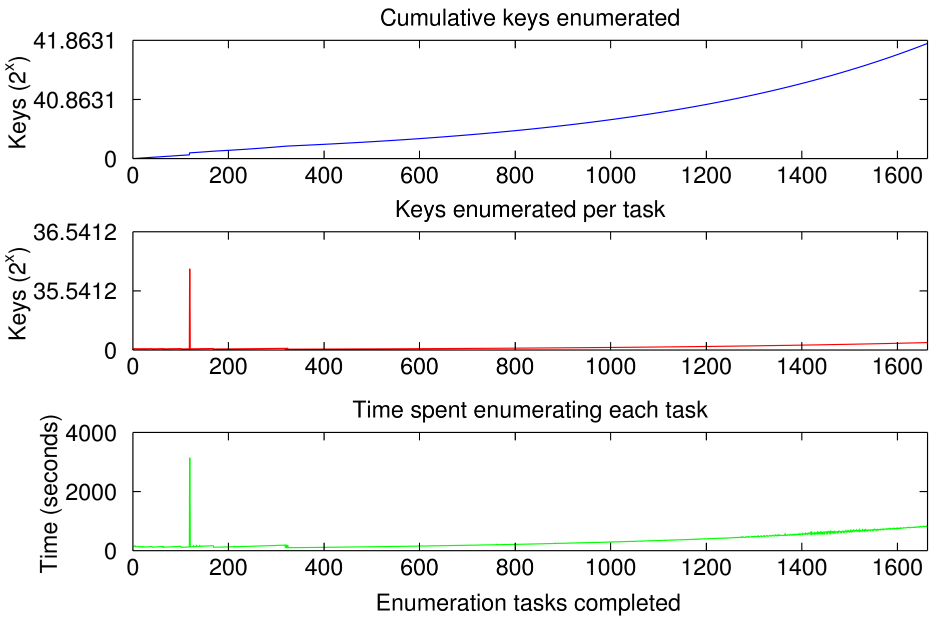

Fig. 9. (Top) The cumulative number of key candidates searched as tasks complete. (Middle) The number of key candidates searched per task. (Bottom) The time spent in each search task.

The gentle slope of the curve suggests that the distribution of work across the cores is very even—evidence that LABYNKYR is able to distribute work evenly at 15 bits of precision.

Compute cluster benchmark. In the following benchmark we attempted to confirm whether we could very quickly recover a low-ranked key by leveraging a large quantity of PEUs. Here we used the same data set as in the previous example, but used a slightly worse attack for which fewer traces were available.

We used the Martin et al. rank algorithm at 20 bits of precision to determine that the rank of the key was $2^{47.9979}$, and consequently would expect to have to search approximately that many keys before finding the correct one. We allowed each core to run indefinitely until the correct key was found (by setting the total budget to be 2^{48}).

In total we used 400 Sandy Bridge Intel Xeon E5-2670 CPUs, clocked at 2.6 GHz and with 4 GiB RAM available per core. The effort was distributed evenly by LABYNKYR across these cores, creating 4284 distinct search tasks, with the initial task (to be executed using the THRESHOLD algorithm) containing the first 2^{36} key candidates, and the remaining tasks consisting of between $2^{30} - 2^{38.6}$ candidates to be enumerated using the ANF/FOREST algorithm. We converted the distinguishing vectors produced by a DPA attack using 40,324 traces into integer weights using MapToWeight at 15 bits of precision.

LABYNKYR found the correct key after 32.42 hours of wall-clock time. In total $2^{47.992}$ or approximately 280 trillion keys were enumerated and verified. Amazon EC2 compute-optimised "c3.4xlarge" instances provide 16 Intel Xeon E5-2680 v2 cores at a cost of 0.84 USD per hour as of the writing[5]. These CPU cores are superior to the ones used in this benchmark, but are likely close enough to serve as a useful guide. If our experiment was performed on Amazon's cloud, we would have required 25 of these instances for (at most) 32.42 hours, equating to a predicted cost of approximately 680.82 USD.

Veyrat-Charvillon et al. project that their enumeration algorithm of SAC 2012 may be able to enumerate 2^{40} keys using a single core and 70 GiB RAM in nine days [18]. Bogdanov et al. report searching up to 2^{44} keys using 8 CPU cores. Details on the exact running time are unclear, but interpreting the results of Figures 8, 9 and 10 in the referenced work suggests the task took less than approximately four days [3]. We have no direct comparison for a parallel implementation of the Poussier et al. algorithm that includes the cost of verification and that benchmarks run-time over multiple cores [17]. We believe LABYNKYR fares favourably to these efforts: we are able to support a much larger degree of parallelism, and as reported in Sec. 5.3, are likely to be at least as efficient in straight-line code. We hope that our benchmarks serve as a useful point of reference for future efforts.

7 Conclusion

In this work we have explored the capabilities of brute-force adversaries armed with side-channel information and an auxiliary key enumeration and verification capability. We have identified how best to allocate the brute-force workload across massively parallel systems and have explored how best to optimise the necessary algorithms for speed and memory complexity. With this knowledge we have developed a powerful tool "LABYNKYR" for running the enumeration phase that can be run on a small handful of workstations, across a large supercomputing cluster or in the cloud.

Our real-world benchmarks demonstrate that it is perfectly possible for an adversary equipped with relatively modest levels of computing resources (relative to say a government agency or an owner of a botnet), or with a small amount of cash and access to a computing cloud, to enumerate and verify the $2^{48} - 2^{50}$ most-likely keys in the order of hours using the results of a side-channel attack on AES-128. This demonstrates that these 'enhanced' side-channel adversaries are realistic threat and must be incorporated into the evaluation and certification procedures such as Common Criteria [5]. Our benchmarks are a useful aid that allows an evaluator to more accurately assess the strength of a particular adversary.

Future research directions are twofold. Firstly, it is important to focus on exploring how best to distribute computing resources between key enumeration

[5] https://aws.amazon.com/ec2/pricing/

and verification. Various factors, including the depth of the keys being enumerated and hardware support for verification control the optimal ratio. Gains from optimising this ratio may be significant, and it is highly likely that a tool will have to adapt this changing ratio during the computation itself.

Secondly, the level of precision retained when converting floating-point distinguishing vectors into integer weights is crucial. Increasing the amount of precision retained allows for more parallelism, but at a cost of increasing the run-time of key enumeration algorithms. It will be important for future efforts to further explore the nature of this relationship.

Acknowledgements This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol –
http://www.bris.ac.uk/acrc/.
This work has been supported in part by EPSRC via grant EP/N011635/1.

Data access Information necessary to repeat the benchmarking is supplied in Appendix A. The parallel attacks described in Sec. 6 use the attack as described by Longo et al. [11].

References

1. D. J. Bernstein, T. Lange, and C. van Vredendaal. Tighter, faster, simpler side-channel security evaluations beyond computing power. *IACR Cryptology ePrint Archive*, 2015:221, 2015.
2. A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique cryptanalysis of the full aes. In *ASIACRYPT 2011*, pages 344–371. Springer, 2011.
3. A. Bogdanov, I. Kizhvatov, K. Manzoor, E. Tischhauser, and M. Witteman. Fast and memory-efficient key recovery in side-channel attacks. *IACR Cryptology ePrint Archive*, 2015:795, 2015.
4. E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In M. Joye and J.-J. Quisquater, editors, *CHES 2004*, volume 3156 of *LNCS*, pages 135–152. Springer Berlin / Heidelberg, 2004.
5. BSI. Common Criteria, Techical editor. Application of Attack Potential to Smart Cards. http://www.commoncriteriaportal.org/files/supdocs/CCDB-2009-03-001.pdf, 2009.
6. EFF. Frequently Asked Questions (FAQ) About the Electronic Frontier Foundation's DES Cracker Machine `https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html` (accessed 21st may 2016).
7. C. Glowacz, V. Grosso, R. Poussier, J. Schüth, and F. Standaert. Simpler and more efficient rank estimation for side-channel security assessment. In *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, pages 117–129, 2015.
8. D. Gstir and M. Schläffer. Fast software encryption attacks on aes. In *AFRICACRYPT 2013*, pages 359–374. Springer, 2013.
9. S. Gueron. White Paper: Intel Advanced Encryption Standard (AES) New Instructions Set. Technical report, Intel, 2012.
10. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Proceedings of CRYPTO 1999*, pages 388–397, London, UK, 1999. Springer-Verlag.

11. J. Longo, E. D. Mulder, D. Page, and M. Tunstall. Soc it to EM: electromagnetic side-channel attacks on a complex system-on-chip. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 620–640, 2015.
12. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards.* Springer, 2007.
13. S. Mangard, E. Oswald, and F.-X. Standaert. One for All – All for One: Unifying Standard DPA Attacks. *IET Information Security*, 5(2):100–110, 2011.
14. M. Marlinspike, D. Hulton, and M. Ray. Defeating pptp vpns and wpa2 enterprise with ms-chapv2. *Defcon, July*, 2012.
15. D. P. Martin, L. Mather, E. Oswald, and M. Stam. Characterisation and Estimation of the Key Rank Distribution in the Context of Side Channel Evaluations. *IACR Cryptology ePrint Archive*, 2016:491, 2016.
16. D. P. Martin, J. F. O'Connell, E. Oswald, and M. Stam. Counting keys in parallel after a side channel attack. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, pages 313–337, 2015.
17. R. Poussier, F.-X. Standaert, and V. Grosso. Simple key enumeration (and rank estimation) using histograms: an integrated approach. *IACR Cryptology ePrint Archive*, 2016:573, 2016.
18. N. Veyrat-Charvillon, B. Gérard, M. Renauld, and F.-X. Standaert. An Optimal Key Enumeration Algorithm and Its Application to Side-Channel Attacks. In L. R. Knudsen and H. Wu, editors, *Selected Areas in Cryptography*, volume 7707 of *LNCS*, pages 390–406. Springer, 2012.
19. N. Veyrat-Charvillon, B. Gérard, and F.-X. Standaert. Security Evaluations beyond Computing Power. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *LNCS*, pages 126–141. Springer, 2013.
20. X. Ye, T. Eisenbarth, and W. Martin. Bounded, yet Sufficient? How to Determine Whether Limited Side Channel Information Enables Key Recovery. In *CARDIS 2014*, volume 7707 of *LNCS*. Springer, 2014.

A Replication of benchmarking

In the following section we outline the necessary information to replicate the enumeration algorithm benchmarking in Sec. 3.

Side-channel data was *simulated* according to the 'standard' DPA scenario as in Mangard et al. [13]. An attacker has N power measurements or traces T_i corresponding to encryptions of N known plaintexts $x_i \in \mathcal{X}$, $i = 1, \ldots, N$ and wishes to recover the secret key $\mathbf{sk}$. For each subkey (so $j = 0, \ldots, m-1$) we assume that each trace T_i is condensed to a single point of interest $P_{i,j}$ and that this value $P_{i,j}$ decomposes additively as $P_j = P_{\text{exp}} + P_{\text{noise}}$. Here P_{exp}, called the *signal*, is a deterministic function of the value of the subkey sk^j and the relevant input x_i, whereas P_{noise}, called the *noise*, is drawn at random according to some distribution that does not depend on any of the input values (including the secret key $\mathbf{sk}$). The signal-to-noise ratio (SNR) is then defined as the ratio of the variance in the signal (when ranging over secret keys and plaintexts) divided by the variance in the noise:[6]

$$\mathrm{SNR} = \frac{Var(P_{\mathrm{exp}})}{Var(P_{\mathrm{noise}}).}$$

The SNR is used to quantify the amount of leakage within a given measurement: the higher the SNR, the more information within the trace that can be exploited.

A single point of interest $P_{i,j}$ can be simulated using a randomly chosen plaintext x_i, a target function $f_{\mathbf{sk_j}}$ defined over the j-th subkey, a leakage function L, and a noise value ϵ sampled from a distribution, where:

$$P_{i,j} = L \circ f_{\mathbf{sk_j}}(x_i) + \epsilon.$$

In our experiments we set the target functions to be the output of the j-th SubBytes operation in the first round of the AES. The leakage function L was taken to be the Hamming weight of the output of these target functions, and the noise was sampled from a Gaussian distribution $\mathcal{N}(0, \sigma^2)$ with zero mean and a variance σ^2 chosen such that the SNR was a low value of 0.03125 (or 2^{-5})[7]. We used 100 randomly generated plaintexts for each repeat experiment, and then launched a correlation DPA attack using the Hamming-weight power model (as per [4]) to create 16 distinguishing vectors, one for each byte of an AES-128 key. The distinguishing table was then passed into the MapToWeight function with a precision of 12, 16 or 20 bits.

Speed measurements were taken using the difference of two C++11 `<chrono>` library `std::chrono::high_resolution_clock::now()` function calls. Peak memory recorded was taken to be the peak memory usage identified by the valgrind tool Massif, with the `pages-as-heap` option set. Each sample point on the graphs in Figures 6 and 4 is of the average of 50 repeated experiments using data simulated as above and supplied with freshly chosen random plaintexts and sampled noise. Figures 5 and 7 use 10 repeated experiments in which the maximum observation over the 10 is recorded, as the time needed for Massif to evaluate the memory usage is expensive and we were primarily concerned with memory usage in terms orders of magnitude, as implementation quality and optimisation can always be tweaked.

[6] Strictly speaking the SNR is defined relative to a subkey and should be indexed by j; however when we refer to the SNR it will be the same for all subkeys.

[7] In this context of targeting the AES SubBytes operation and using Hamming weight leakage, the signal P_{exp} has variance 2.0, and thus for a specified SNR the requisite variance of the Gaussian distribution used to sample P_{noise} is 2.0 divided by this SNR.