Quantum Key Search with Side Channel Advice

Daniel P. Martin[1], Ashley Montanaro[2], Elisabeth Oswald[3], and Dan Shepherd[4]

[1] School of Mathematics, University of Bristol, Bristol, BS8 1TW, UK, and the Heilbronn Institute for Mathematical Research, Bristol, UK.**
dan.martin@bristol.ac.uk
[2] School of Mathematics, University of Bristol, University Walk, Bristol, BS8 1TW, United Kingdom. ashley.montanaro@bristol.ac.uk
[3] Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom.
elisabeth.oswald@bristol.ac.uk
[4] National Cyber Security Centre, Hubble Road, Cheltenham, GL51 0EX, United Kingdom
Daniel.S@ncsc.gov.uk

Abstract. Recently, a number of results have been published that show how to combine classical cryptanalysis with quantum algorithms, thereby (potentially) achieving considerable speed-ups. We follow this trend but add a novel twist by considering how to utilise side channel leakage in a quantum setting. This is non-trivial because Grover's algorithm deals with *unstructured* data, however we are interested in searching through a key space which has structure due to the side channel information. We present a novel variation of a key enumeration algorithm that produces batches of keys that can be *efficiently* tested using Grover's algorithm. This results in the first quantum key search that benefits from side channel information.

Keywords: Quantum Computation, Side Channel Attacks

1 Introduction

The announcement that NIST will embark on a post-quantum cryptography project has injected further enthusiasm into researching cryptography in the presence of quantum computers. At present there exist a number of algorithms that run efficiently on a quantum computer (see [22] for a survey of the current state of quantum computation). Some of these are a clear threat to existing cryptographic techniques and algorithms. For instance, Shor's algorithm [24] to factor integers leaves a host of cryptographic schemes insecure. Another example is Grover's algorithm [9], which can be used to achieve a quadratic speedup in the majority of unstructured search problems including brute force key search.

** This research was carried out while D. P. Martin was a member of the Department of Computer Science, University of Bristol.

Ongoing research in post quantum cryptography focuses on studying adversarial models alongside cryptographic constructions that include access to quantum algorithms (e.g. Anand *et al.* [1] investigate the quantum IND-CPA security of various block cipher modes of operation). Recent research [12,13] also studies how classical cryptanalytic techniques might benefit from quantum algorithms via appropriating Simon's algorithm [25], and enquire about how realistic, for example, a potential brute-force key search on AES would be [8]. Interestingly, current thinking about post quantum cryptography only marginally touches on adversaries that also have access to additional information.

We believe that considering how leakage might be exploited within the quantum setting should be a pressing research question. After all, since 1996 when Kocher [14] showed how side channels[5] can be used to to break implementations of otherwise secure schemes, the community has witnessed a host of effective side channel attacks.

Many side channel attacks operate in two steps: first the device/implementation leakage is turned into information leakage about the key resulting in probability or score vectors for each independent chunk of the key; second a search over the most likely keys is conducted. Our paper is not concerned with the specifics of the first step. It is the second step, which turns probability/score vectors on chunks of the key into information about the (whole) key, on which our work will focus, as we will motivate next.

Typical side channel attacks trade off data complexity (i.e. the number of queries to a device/implementation as part of the first step) and computational complexity (i.e. the effort that it takes to actually determine the secret in the second step). Given that many practical side channel attacks have a comparatively low data complexity, there is little to be gained from quantum speed-ups in that respect. However, if we consider side channel attacks that trade off using very few queries for a large computational effort (via some enumeration/key search following the key leakage extraction) it seems (intuitively) that access to a quantum algorithm could help.

The logical starting point for search problems is, of course, Grover's algorithm, which can speed up any unstructured search. However, we are interested in a highly structured search. At first sight, it seems hence impossible to 'marry up' Grover (which cannot, as written, benefit from structure on the search data) and side channel information (which is essentially structure on the search data).

The post-leakage search problem essentially is a quantum search problem where there is some additional information available about the likelihood of each element being the key. The conundrum of how to effectively use Grover in this context was first tackled by Montanaro [21]. The algorithm takes in a set of elements (to be searched and tested), as well as an advice distribution for the set, and inputs the most likely elements to Grover in 'batches' of increasing

[5] A side channel is some additional (unintended) channel that an adversary has access to. Beyond power and timing analysis, side channel attacks can be based on the electromagnetic emanation of a device [16], error messages communicated by a device [18] and even the sound that a device produces [6].

sizes, optimised to obtain an efficient quantum search algorithm. It would thus seem that this algorithm already gives the solution to the post-leakage search problem. However, one crucial implicit assumption was made in [21]: that the advice distribution was given in order of likelihood.

Side channel attacks typically produce information about the independent chunks of the unknown key (rather than the whole key) and thus they do not conveniently output the kind of sorted list that the algorithm of [21] requires. Also, it would be impossible to do so in the case of many interesting practical scenarios, e.g. the minimum recommended key length today is 128 bits, thus it is clearly impossible to explicitly generate an ordered list containing 2^{128} elements.

1.1 Contribution and Outline

We give a novel version of Martin *et al.* [20] that is able to *efficiently generate* keys (to be tested) according to a side channel advice distribution. This novel version can output single keys with a specific weight. We then show how to define an efficient, distribution based quantum search algorithm inspired by the quantum algorithm of Montanaro [21].

Our contribution is organised as follows. In Sect. 2 we introduce notation and recap the latest developments in fast and parallel key search. The first contribution of this work (in Sect. 3) is then to take the key rank algorithm of Martin *et al.* [20] and show how to use it to return a single key (the r^{th}) with a weight in a particular range. Using this new insight, and varying the value of r, we are able to construct a new, more efficient, key enumeration/search algorithm in Sect. 3.1. Our main contribution is showing how the newly derived (classical) search algorithm can then be turned into a quantum key search algorithm in Section 4, which provides a quadratic speed-up over the classical algorithm. To our knowledge this is the first time that a side channel attack has been improved with the use of a quantum algorithm.

2 Preliminaries

Our work brings together recent advances from side channel research (key rank and enumeration) and quantum algorithms (quantum search with advice). To keep the paper reasonably self contained, we introduce and explain the necessary background regarding key enumeration/search, alongside introducing notation.

The cryptographic attack may work against the secret key in any kind of cryptography, *e.g.* symmetric cryptography (such as a block cipher) or public key cryptography (such as a signing algorithm). All we assume about the secret key, $\boldsymbol{k}$, is that some information is leaked by the implementation about $\boldsymbol{k}$, and that this splits up into m independent chunks, called subkeys $(k_1, \ldots, k_m)$, each of which can take one of n possible values. Whilst our algorithms do not require that each subkey is the same size, this assumption helps to ease explanation. We denote the secret key to be targeted by the attack as $\boldsymbol{t} = (t_1, \ldots, t_m)$.

Overall weight					
0	**1**	**2**	**3**	**4**	**5**
$(1,2)$	$(1,1)$	$(2,1)$	$(2,3)$	$(3,1)$	$(3,3)$
	$(2,2)$	$(1,3)$	$(3,2)$		

Table 1: All possible keys sorted by weight.

Our work is not concerned with how the leakage is obtained or how it is manipulated to infer information about the key. We refer to the established literature (e.g. [17]) for an in-depth explanation. We only assume that the result of a leakage attack is an n by m matrix $\boldsymbol{w} = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m)$, $w_{j,i} \in \mathbb{Z}^+$. Each column represents the likelihood information that we have about the values of a respective key chunk, whereby we adopt the convention that larger numbers correspond to smaller likelihoods. We also assume that there is a notion of 'adding' likelihoods, and this is defined by integer addition. Thus, we can determine the weight (likelihood) of any (sub)set of subkeys by simply adding up weights. The likelihood of a key $\boldsymbol{k}$ will be denoted $\rho_{\boldsymbol{k}} = \sum_{i=1}^{m} w_{k_i,i}$

Remark 1. Different types of attack techniques may lead to different types of matrix (i.e. some attacks might produce probabilities as outputs, others integers). There are existing techniques such as [3,23,26] that show that it is possible to 'convert' various side channel attack outputs to probabilities. Other papers [20,19,4] discuss converting probabilities to integers (i.e. they enquire regarding how much precision needs to be retained). In summary, whilst the conversion of outcomes from typical leakage attacks to integer values is normally lossy, previous work shows that in well understood scenarios it can be done and leads to sensible results.

2.1 Key Search with Additional Information

To ease further explanations, we now introduce a small example and use it to motivate the notions of key rank, enumeration and search.

Example 1. Our illustrative toy example, which will run throughout the paper, consists of a key that can be split into two subkeys, where each subkey can take three different values $\{1, 2, 3\}$. The target key $\boldsymbol{t}$ in this example is $\boldsymbol{t} = (2, 1)$. The observed leakage has been turned into the matrix that contains the information about how likely each of the values are:

$$\boldsymbol{w} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 3 & 2 \end{pmatrix}$$

Remember that lower weights indicate more likely values, and the weight of the key can be derived by adding the weights of the subkeys. We can thus sort the key combinations according to their overall weight, as shown in Table 1.

The weight of the target key $\boldsymbol{t}$ is $\rho_{\boldsymbol{t}} = w_{2,1} + w_{1,2} = 1 + 1 = 2$. Thus in an ordered list, it would appear after the keys with weights 0 and 1. There are three keys with weights 0 and 1, hence the rank of the target key will be 3 (the number of more likely keys[6]).

As should become clear from the example, we can define the rank of a key $\boldsymbol{t}$ with respect to a weight matrix $\boldsymbol{w}$ in a natural manner.

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

$$\mathsf{rank}_{\boldsymbol{t}}(\boldsymbol{w}) = |\{\boldsymbol{k} = (k_1, \ldots, k_m) : \rho_{\boldsymbol{k}} < \rho_{\boldsymbol{t}}\}|$$

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

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

Note that this definition only asks for the e most likely keys, and not that they are returned in likelihood order. Optimal key enumeration would require exactly that, i.e. output the e most likely keys $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_e$ in the order of their weights.

In certain scenarios (such as restarting an enumeration algorithm) the adversary may require e keys from an arbitrary position in the key space. This is captured by Extended Key Enumeration.

Definition 3 (Extended Key Enumeration). *Given an $n \times m$ weight matrix $\boldsymbol{w}$ and $e, f \in \mathbb{Z}$, output the e keys with the lowest weights (breaking ties arbitrarily), after ignoring the first f keys.*

In this scenario the algorithm will output keys $\boldsymbol{k}_{f+1}, \ldots, \boldsymbol{k}_{e+f}$.

Clearly to succeed in an attack, an adversary needs not just to enumerate the most likely keys, but needs to check which one is the target key. This is achieved using a testing function T which behaves as follows:

$$\mathsf{T}(\boldsymbol{k}) = \begin{cases} 1 \text{ if } \boldsymbol{k} = \boldsymbol{t} \\ 0 \text{ otherwise} \end{cases}$$

More concretely, in the context of symmetric encryption; the testing function could utilise one or more plaintext/ciphertext pairs together with the underlying scheme.

[6] Rank could be defined as keys with a lower or equal weight but considering a strictly lower weight favours the adversary.

Example 2. Consider an attack on the block cipher AES with 128 bit keys. We assume that the adversary has access to a plaintext/ciphertext pair $(m, c = \mathsf{AES}_{\boldsymbol{t}}(m))$, and an implementation of AES. In this situation T can be constructed as follows:

$$\mathsf{T}(\boldsymbol{k}) = \begin{cases} 1 \text{ if } \mathsf{AES}_{\boldsymbol{k}}(m) = c \\ 0 \text{ otherwise} \end{cases}$$

We can now define key search.

Definition 4 (Key Search). *Given an $n \times m$ weight matrix $\boldsymbol{w}$, a testing function T and $e \in \mathbb{Z}$, output any $\boldsymbol{k}_i$, with $i \leq e$, such that $\mathsf{T}(\boldsymbol{k}_i) = 1$ and $\boldsymbol{k}_i$ would be output from enumeration, on input $\boldsymbol{w}$ and e. If no such i exists output $\perp$.*

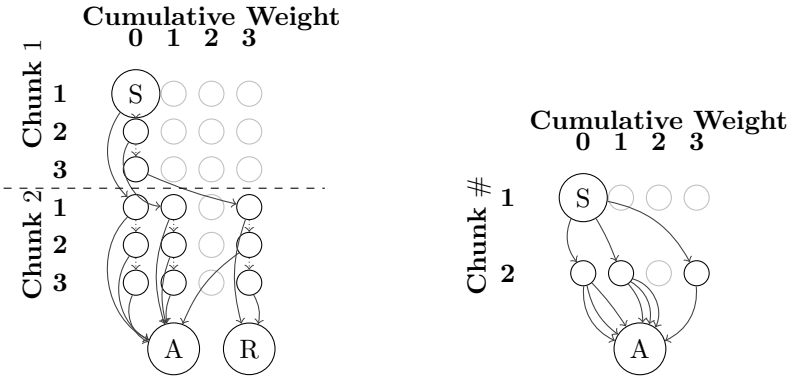
A similar definition can be given for Extended Key Search.

2.2 Efficiently computing the rank of a key

We base our work on the key *rank* algorithm by Martin *et al.* [20] (along with the improvements [19,15]). This might be surprising at first as we are aiming to construct a quantum key *search* algorithm. However, Martin *et al.*'s rank algorithm directly enables the construction of a quantum-compatible key search algorithm. Therefore we now briefly sketch the working principle of their algorithm.

An integer parameter W is fixed, which denotes the target weight, or the largest weight that should be considered. A graph is specified with $n \cdot m \cdot W + 2$ vertices, according to the following simple rules (described informally). Two vertices are called 'Accept' and 'Reject' and these are sink vertices. The other vertices are called $v_{i,j,w}$ for $i \in [1..m], j \in [1..n]$, and $w \in [0..W-1]$. Each has out-valency two, so that each such vertex $v_{i,j,w}$ has a 'right child' that represents the idea that $k_i = j$ (consider the ith subkey selected) and a 'left child' that represents the idea that $k_i \neq j$ (consider the ith subkey yet to be determined). A path from $v_{1,1,0}$ to 'Accept' will take exactly m 'right' forks, so that each subkey is selected exactly once on the path, so that the path effectively selects a whole key. A path will only reach 'Accept' if the total accumulated weight from these selections is kept below W, otherwise it will divert to 'Reject'. The number of paths from $v_{1,1,0}$ to 'Accept' is therefore constructively identical with the number of keys having weight strictly less than W, and therefore is actually the *rank* of any key having weight exactly W, if such one exists.

Example 3. We construct a graph for our running example and choose the target weight W to equal 4, i.e. we want to know how many combinations of subkeys lead to a key with weight strictly smaller than 4. Our graph hence contains $2 \cdot 3 \cdot 4 + 2$ vertices and can be drawn in a 'flattened' version, as shown in Fig. 1a. The upper 'half' corresponds to the first subkey, and the lower half to the second subkey. The vertices in each column represent the current weight. To draw the graph, we begin at the start node S ($v_{1,1,0}$), and then draw the right child (it points to a vertex representing the first value of the second subkey with the



(a) The original graph structure of [20] (b) The more efficient graph structure of [19]

Fig. 1: Two possible graph constructions for our running example (with $W = 4$).

correct weight $v_{2,1,0}$) and the left child (points to a vertex representing the next value in the subkey $v_{1,2,0}$, unless it is the last value in which case it points to reject – these are omitted for readability).

The right child of S points to weight 0 in the next subkey (because the weight of having $k_1 = 1$ equals zero in our example), and the left child points at the weight 0 in the next row (because we are not choosing the element so the weight remains unchanged). Suppose we now consider the vertex $v_{2,1,0}$. This again has two children. The right child corresponds to choosing the first value of subkey 2, which has weight 1. Hence the total weight is 1, which is smaller than 4 and thus the right child goes into the accept node. The left child corresponds to not choosing the first value, but considering the second value ($v_{2,2,0}$). The other paths in the graph are generated according to the same principles.

The algorithm to compute the key rank counts all paths that lead to the accept node. Consequently, by augmenting the algorithm to also store the corresponding subkeys that are visited on those paths that lead to accept, this algorithm immediately gives rise to a key enumeration algorithm. There are different considerations (in particular the choice of ordering, which impacts on memory complexity) when implementing this principle and [20] discusses these in great depth. In recent work, the algorithm was further simplified and made more efficient by slightly changing the recurrence relation that iterates through the graph [19]. Further work gave evidence that there might be still a (significantly) faster key rank algorithm possible: [15] contains an algorithm 'Threshold' which proves to be the fastest among the compared algorithms, but at the significant disadvantage that it does not support extended key enumeration. Since

the Threshold algorithm does not support ranking between two weights, it is not suitable for our purpose.

3 Key Ranking Leading to Faster Enumeration

The key rank algorithm in the previous section constructed a graph (and counted paths in it) by using right children to move 'down the graph into the next chunk' and left children to indicate that a value had not been selected. Thus every node had exactly two outgoing edges. However, the graph could be compressed by allowing vertices to have multiple outgoing edges, resulting in a two, instead of three, index system. This was explored, and shown to be more efficient, in [19].

Example 4. We refer again to our running example. Let $\boldsymbol{v}_i$ correspond to the row, in the graph, for the i^{th} key chunk. The start node now points to 3 vertices representing the three possible values the subkey could take. The vertices for the second subkey have edges going to accept if and only if adding the weight for the respective value results in a total weight smaller than W. Figure 1b shows the corresponding graph. There are three edges from $v_{2,0}$ (and from $v_{2,1}$) to the accept node because all three weights in $\boldsymbol{w}_2$ are smaller than 4 (and 3 resp.). There are two edges from $v_{2,2}$ to the accept node because two weights of $\boldsymbol{w}_2$ are smaller than $4-2=2$, however no edge connects into $v_{2,2}$ so we don't draw it in our graph. There is only one edge possible from $v_{2,3}$ because only one value of $\boldsymbol{w}_2$ is small enough such that the overall weight is smaller than 4.

Our key observation is that the number of vertices from the edge to the accept node can be written down in a simple and elegant manner. Let us consider the vertex $v_{i,w}$ for the pair (i,w). The vertices $v_{i,w}$, for $i<m$, have out degree at most n ($v_{i,w}$ has an edge to $v_{i+1,w+w_{j,i}}$ for $1\leq j\leq n$ when $w+w_{j,i}<W$). Let there also be an accept node (which is a sink) such that $v_{m,w}$ has edges to the sink when $w_{j,m}<W-w$. With this we can define a matrix $\boldsymbol{b}$, where $b_{i,w}$ stores how many paths there are from $v_{i,w}$ to the sink. Since each path from $v_{1,0}$ corresponds to a key with weight at most W, this gives a representation that is equivalent to the graph. The equations for constructing $\boldsymbol{b}$ are given below.

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

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

where $\mathbf{1}\{\cdot\}$ returns 1 if the expression in the curly brackets evaluates to true and 0 otherwise.

The array index $b_{1,0}$ contains the rank of the key with score W. It is assumed that $b_{i,w}=0$ for all $1\leq i\leq m$ if $w\geq W$. Correctness follows from [20].

In order to compute $b_{1,0}$ we start by filling in the values for $b_{m,w}$ for $0 \leq w < W$ (using Eq. 2) and then fill in $b_{i,w}$ working backwards over the i's (using Eq. 1). Each $b_{i,w}$ is computed and stored once. Since there are $m \cdot W$ matrix entries, each of which look at n $b_{i,w}$'s and then writes an integer of size $m \cdot \log n$ (since there are n^m total keys), the total time complexity is $\mathcal{O}(m^2 \cdot n \cdot W \cdot \log n)$.

As $\boldsymbol{b}$ contains $m \cdot W$ elements, each of which contains an integer of size $m \cdot \log n$, the required space is $\mathcal{O}(m^2 \cdot W \cdot \log n)$.[7]

It is possible to change the rank algorithm such that it counts all keys with weight in a particular range, instead of weight less than a target. We refer to this algorithm as $\mathsf{Rank}(\boldsymbol{w}, W_1, W_2)$, and define it formally in Alg. 6 (App. A). This helps to meet the extended key enumeration definition and will be required for our new enumeration algorithm. To achieve this Eq. 2 is replaced with the following:

$$b_{m,w} := \sum_{j=1}^{n} \mathbf{1}\{W_1 - w \leq w_{j,m} < W_2 - w\} \tag{3}$$

We assume that an algorithm exists that 'fills' $\boldsymbol{b}$ with the correct values for weights $[W_1, W_2)$, called $\mathsf{Initialise}(\boldsymbol{w}, W_1, W_2)$, which is formally defined in Alg. 4 (App. A).

The getKey algorithm We will require an algorithm $\mathsf{getKey}(\boldsymbol{b}, \boldsymbol{w}, W_1, W_2, r)$ which returns the r^{th} key with weight between W_1 and W_2 to design a quantum search algorithm with side channel advice.[8] This can be achieved utilising the data structure $\mathbf{b}$, as shown in Alg. 1.

Correctness of getKey follows from the correctness of $\boldsymbol{b}$. Since the algorithm is deterministic it is clear that given the same r twice it will return the same key and that, due to its similarity to Depth First Search, no key will be returned twice, for different r. Thus we indeed have a uniquely determined r^{th} key. This is also important for the quantum and classical enumeration algorithms that follow. The algorithm has to assign values to each of the m subkeys, which can involve up to n comparisons of integers of size $m \cdot \log n$. This gives the algorithm a time complexity of $\mathcal{O}(m^2 \cdot n \cdot \log n)$.

3.1 A Faster Classical Enumeration Algorithm

The getKey algorithm given in Alg. 1 can trivially be converted into an algorithm which enumerates all keys, with weight in the range $[W_1, W_2)$.

[7] Martin *et al.* [20] show how to tweak their algorithm such that the entirety of $\boldsymbol{b}$ does not have to be stored. However, for enumeration, repeat access to $\boldsymbol{b}$ is required and thus this is not applicable.

[8] The r^{th} key does not have to be the r^{th} most likely key in this range, any arbitrary ordering will suffice.

Algorithm 1 An algorithm for requesting particular keys

function getKey($\boldsymbol{b}, \boldsymbol{w}, W_1, W_2, r$)
  if $r > b_{1,0}$ **then return** $\perp$ **end if**
  $\boldsymbol{k} \leftarrow [0]^m$
  $w \leftarrow 0$
  for $i = 1$ to $m - 1$ **do**
    for $j = 1$ to n **do**
      if $r \leq b_{i+1,w+w_{j,i}}$ **then**
        $k_i \leftarrow j$
        $w \leftarrow w + w_{j,i}$
        break j
      end if
      $r \leftarrow r - b_{i+1,w+w_{j,i}}$
    end for
  end for
  for $j = 1$ to n **do**
    if $r \leq \mathbf{1}\{W_1 - w \leq w_{j,m} < W_2 - w\}$ **then**
      $k_m \leftarrow j$
      break
    end if
    $r \leftarrow r - \mathbf{1}\{W_1 - w \leq w_{j,m} < W_2 - w\}$
  end for
  return $\boldsymbol{k}$
end function

If there are e keys in the range $[W_1, W_2)$, the keyEnumerate algorithm simply runs getKey e times, giving a total time complexity of $\mathcal{O}(m^2 \cdot n \cdot W_2 \cdot \log n + e \cdot m^2 \cdot n \cdot \log n)$. The original algorithm by Martin *et al.* [20] has time complexity $\mathcal{O}(e \cdot m^2 \cdot n \cdot W_2 \cdot \log n)$. Therefore, the new algorithm is considerably faster. Since our algorithm can be split into enumeration ranges, it can be made highly parallelisable using techniques from [15]. As there is a trade off between range size and runtime, we will discuss this is more detail (for a single machine) below. A formal description can be found in Alg. 5 (App. A). Correctness of keyEnumerate follows from the correctness of getKey[9].

To convert the enumeration algorithm into a key search algorithm keySearch, rather than storing the keys they would be tested using T. Upon finding the correct key the algorithm terminates, otherwise (if all keys in the budget have been tested but the key was not found) the algorithm returns $\perp$.

[9] The keyEnumerate algorithm could be made more efficient by directly adjusting getKey instead of calling it multiple times in a disjoint manner. The bottleneck that arises is that getKey$(\boldsymbol{b}, r)$ and getKey$(\boldsymbol{b}, r + 1)$ might perform a lot of similar work to output the key, for example they may have the same $m - 1$ first subkeys. This can be avoided using backtracking to produce keys in a manner similar to depth first search.

Algorithm 2 The key search algorithm

function $\mathsf{KS}(\boldsymbol{w}, e, \mathsf{T})$
  $W_1 \leftarrow W_{min}$
  $W_2 \leftarrow W_{min} + 1$
  $step \leftarrow 0$
  Choose W_e such that $\mathsf{Rank}(\boldsymbol{w}, 0, W_e)$ is approx e
  while $W_1 \leq W_e$ **do**
    $\boldsymbol{k} \leftarrow \mathsf{keySearch}(\boldsymbol{w}, W_1, W_2, \mathsf{T})$
    if $\boldsymbol{k} \neq \perp$ **then return** $\boldsymbol{k}$ **end if**
    $step \leftarrow step + 1$
    $W_1 \leftarrow W_2$
    Choose W_2 such that $\mathsf{Rank}(\boldsymbol{w}, W_1, W_2)$ is approx a^{step}
  end while
  return $\perp$
end function

Combining together the above algorithm with the techniques for searching over partitions independently gives the key search algorithm in Alg. 2. To construct our algorithm, we draw inspiration from the algorithm of Montanaro [21]. It works by partitioning the search space into sections whose size follows a geometrically increasing sequence using a size parameter $a = \mathcal{O}(1)$. This parameter is chosen such that the number of loop iterations is balanced with the number of keys verified per block. It is fairly straightforward to see that this is the optimal choice (it follows similar ideas to the Exponential Search Algorithm [2]).

3.2 Total Runtime

The algorithm starts by finding W_e, which takes $\mathcal{O}(m^2 \cdot n \cdot W_{max} \cdot \log n + \log W_{max})$ time,[10] where W_{max} is the key with the largest weight. Since the algorithm searches e keys such that approximately a^s keys are tested at each iteration s, the loop will iterate $\mathcal{O}(\log_a e)$ times.

On iteration s, the call to $\mathsf{keySearch}$ takes $\mathcal{O}(m^2 \cdot n \cdot W_2 \cdot \log n + a^s \cdot m^2 \cdot n \cdot \log n)$. Finally, the call to calculate W_2 costs $\mathcal{O}(\log W_e)$ look ups in the array generated when choosing W_e, as $W_2 \leq W_e$ we can binary search up to W_e instead of W_{max}. Putting it all together gives an asymptotic time complexity of $\mathcal{O}(m^2 \cdot n \cdot \log n(W_{max} + e + W_e \cdot \log e))$. See App. B for the derivation details.

4 Quantum Key Search

Finally we are in a position to give the novel quantum search with side channel advice algorithm, which achieves a quadratic speed-up over the classical key

[10] As shown by Martin *et al.* [19]. The initial $\mathcal{O}(m^2 \cdot n \cdot W_{max} \cdot \log n)$ can be reused by future queries reducing their work to $\mathcal{O}(\log W_{max})$.

Algorithm 3 The quantum key search algorithm

function $\mathsf{QKS}(\boldsymbol{w}, e, \mathsf{T})$
  $W_1 \leftarrow W_{min}$
  $W_2 \leftarrow W_{min} + 1$
  $step \leftarrow 0$
  Choose W_e such that $\mathsf{Rank}(\boldsymbol{w}, 0, W_e)$ is approx e
  while $W_1 \leq W_e$ **do**
    $\boldsymbol{b} \leftarrow \mathsf{Initialise}(\boldsymbol{w}, W_1, W_2)$
    $\mathsf{f}(\cdot) \leftarrow \mathsf{T}(\mathsf{getKey}(\boldsymbol{b}, \boldsymbol{w}, W_1, W_2, \cdot))$
    Call Grover using f for one or zero marked elements in range $[W_1, W_2)$
    if marked element t found **then**
      return $\mathsf{getKey}(\boldsymbol{b}, \boldsymbol{w}, W_1, W_2, t)$
    end if
    $step \leftarrow step + 1$
    $W_1 \leftarrow W_2$
    Choose W_2 such that $\mathsf{Rank}(\boldsymbol{w}, W_1, W_2)$ is approx a^{step}
  end while
  return $\perp$
end function

search. We heavily rely on Grover's algorithm [9], which is a quantum algorithm to solve the following problem: *Given a black box which returns 1 on a single input x, and 0 on all other inputs, find x.* If there are X possible inputs to the black box, the classical algorithm uses $\mathcal{O}(X)$ queries to the black box – the correct input might be the very last input tested. However, a version of Grover's algorithm solves the problem using $\mathcal{O}(\sqrt{X})$ queries, with certainty [10,11,5]. It is easy to generalise this to the case where we have either zero or one inputs on which the testing function returns 1 (which is our setting), at the cost of one extra query. To do this, run the algorithm of [10,11,5] and apply the testing function to the answer obtained. If it returns 0, there must have been no input on which the testing function would return 1.

Our QKS algorithm based on this subroutine is given in Alg. 3. The algorithm is nearly identical to the classical KS one given in Alg. 2. The crucial difference is the work done within the loop. Since Grover's algorithm is being called instead of keySearch, some of the work classically done in keySearch must be done within the loop, so that it is compatible with Grover. The algorithm must generate the array $\boldsymbol{b}$, construct a testing function which takes in a 'key index' instead of a key and convert the index output back to a key. Otherwise, the algorithm behaves exactly the same as the classical algorithm.

4.1 Total Runtime

We assume we have access to a coherently addressable quantum RAM (QRAM) [7], which allows us to efficiently read the data structure b in quantum superposition.

Such a QRAM can be initialised in time proportional to the size of b. We stress that in our case b is relatively small, so this does not substantially affect the time complexity of the algorithm.

Most of the time complexity of the quantum algorithm can be assessed in the same way as for the classical algorithm. The only exception is that at iteration s, the algorithm makes $\mathcal{O}(a^{\frac{s}{2}})$ calls to getKey instead of the a^s calls classically.

We show (details are in App. B) that the time complexity of the total calls that Grover's algorithm makes to getKey is $\mathcal{O}(\sqrt{e} \cdot m^2 \cdot n \cdot \log n)$. Combining this with the classical analysis of the rest of the algorithm gives the total time complexity of $\mathcal{O}(m^2 \cdot n \cdot \log n(W_{max} + \sqrt{e} + W_e \cdot \log e))$.

While the classical and quantum time complexities look fairly similar, we get a quadratic speed-up because the parameters m, n, W are attack dependent and tend to be fairly small. For example, for typical attacks on AES-128, $m = 16$ and $n = 256$. The weights W are normally controlled by the attacker using a precision parameter and thus unlikely to grow large. Thus the dominating variable is the number of keys enumerated, which gains a quadratic improvement in a quantum setting.

Conclusion We demonstrated that it is possible to leverage the power of a side channel attack in the quantum setting. Our quantum key search with side channel advice thus benefits from a quadratic improvement over a classical key search. Clearly our work is restricted to the setting of 'classical' side channel attacks that follow a divide and conquer principle, which result in information about subkeys independently. However, this setting is very common and applies to attacks such as differential and simple power (EM, timing, cache) analysis.

Acknowledgements and Disclaimer Ashley Montanaro was supported by an EPSRC Early Career Fellowship EP/L021005/1. Elisabeth Oswald was in part supported by EPSRC via grant EP/N011635/1 (LADA). No research data was created for this paper.

References

1. Mayuresh Vivekanand Anand, Ehsan Ebrahimi Targhi, Gelo Noel Tabia, and Dominique Unruh. Post-quantum security of the CBC, CFB, OFB, CTR, and XTS modes of operation. Cryptology ePrint Archive, Report 2016/197, 2016. `http://eprint.iacr.org/2016/197`.
2. Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.
3. Daniel J. Bernstein, Tanja Lange, and Christine van Vredendaal. Tighter, faster, simpler side-channel security evaluations beyond computing power. Cryptology ePrint Archive, Report 2015/221, 2015. `http://eprint.iacr.org/2015/221`.
4. Andrey Bogdanov, Ilya Kizhvatov, Kamran Manzoor, Elmar Tischhauser, and Marc Witteman. Fast and memory-efficient key recovery in side-channel attacks. In *International Conference on Selected Areas in Cryptography*, pages 310–327. Springer, 2015.

5. Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.
6. Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, Heidelberg, August 2014.
7. Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical review letters*, 100(16):160501, 2008.
8. Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying grover's algorithm to AES: quantum resource estimates. In *PQCrypto 2016*, volume 9606 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2016.
9. Lov K. Grover. A fast quantum mechanical algorithm for database search. In *28th Annual ACM Symposium on Theory of Computing*, pages 212–219. ACM Press, May 1996.
10. Lov K Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physical review letters*, 79(2):325, 1997.
11. Peter Høyer. Arbitrary phases in quantum amplitude amplification. *Physical Review A*, 62(5):052304, 2000.
12. Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Breaking symmetric cryptosystems using quantum period finding. In *CRYPTO 2016 - Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 207–237. Springer, 2016.
13. Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Quantum differential and linear cryptanalysis. In *ToSC*. Springer, 2017.
14. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, Heidelberg, August 1996.
15. Jake Longo, Daniel P. Martin, Luke Mather, Elisabeth Oswald, Benjamin Sach, and Martijn Stam. How low can you go? Using side-channel data to enhance brute-force key recovery. Cryptology ePrint Archive, Report 2016/609, 2016. `http://eprint.iacr.org/2016/609`.
16. Jake Longo, Elke De Mulder, Daniel Page, and Michael Tunstall. SoC it to EM: ElectroMagnetic side-channel attacks on a complex system-on-chip. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 620–640. Springer, Heidelberg, September 2015.
17. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.
18. James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer, Heidelberg, August 2001.
19. Daniel P Martin, Luke Mather, Elisabeth Oswald, and Martijn Stam. Characterisation and estimation of the key rank distribution in the context of side channel evaluations. In *Asiacrypt 2016*, volume 10031, pages 548–572. Springer, 2016.
20. Daniel P. Martin, Jonathan F. O'Connell, Elisabeth Oswald, and Martijn Stam. Counting keys in parallel after a side channel attack. In Tetsu Iwata and Jung Hee

Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 313–337. Springer, Heidelberg, November / December 2015.
21. Ashley Montanaro. Quantum search with advice. In *Conference on Quantum Computation, Communication, and Cryptography*, pages 77–93. Springer, 2010.
22. Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.
23. Jing Pan, Jasper G. J. van Woudenberg, Jerry den Hartog, and Marc F. Witteman. Improving DPA by peak distribution analysis. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *SAC 2010: 17th Annual International Workshop on Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 241–261. Springer, Heidelberg, August 2011.
24. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Computer Society Press, November 1994.
25. Daniel R Simon. On the power of quantum computation. *SIAM journal on computing*, 26(5):1474–1483, 1997.
26. Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Security evaluations beyond computing power. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 126–141. Springer, Heidelberg, May 2013.

A Additional Algorithms

For completeness, in this appendix we give any additional algorithms required for implementation of the key search algorithms.

Algorithm 4 The initialise algorithm to generate **b**

function Initialise($\boldsymbol{w}, W_1, W_2$)
  $\boldsymbol{b} \leftarrow [[0]^{W_2}]^m$
  for $w = 0$ to $W_2 - 1$ **do**
    for $j = 1$ to n **do**
      $b_{m,w} \leftarrow b_{m,w} + \mathbf{1}\{W_1 - w \leq w_{j,m} < W_2 - w\}$
    end for
  end for
  for $i = m - 1$ down to 1 **do**
    for $w = 0$ to $W_2 - 1$ **do**
      for $j = 1$ to n **do**
        if $w + w_{j,i} < W_2$ **then**
          $b_{i,w} \leftarrow b_{i,w} + b_{i+1,w+w_{j,i}}$
        end if
      end for
    end for
  end for
  return b
end function

Algorithm 5 A new enumeration algorithm.

function keyEnumerate($\boldsymbol{w}, W_1, W_2$)
  $\boldsymbol{K} \leftarrow \{\}$
  $\boldsymbol{b} \leftarrow$ Initialise($\boldsymbol{w}, W_1, W_2$)
  $\boldsymbol{k} \leftarrow \emptyset$
  $r \leftarrow 1$
  while True **do**
    $\boldsymbol{k} \leftarrow$ getKey($\boldsymbol{b}, \boldsymbol{w}, W_1, W_2, r$)
    if $\boldsymbol{k} = \perp$ **then break end if**
    $\boldsymbol{K} \leftarrow \boldsymbol{K} \cup \{\boldsymbol{k}\}$
    $r \leftarrow r + 1$
  end while
  return $\boldsymbol{K}$
end function

Algorithm 6 The key rank algorithm

function Rank($\boldsymbol{w}, W_1, W_2$)
  $\boldsymbol{b} \leftarrow$ Initialise($\boldsymbol{w}, W_1, W_2$)
  return $b_{1,0}$
end function

B Time Complexity Calculations

The time complexity of the classical key search algorithm was derived using the following calculations:

$$
\begin{aligned}
& m^2 \cdot n \cdot W_{max} \cdot \log n + \log W_{max} \\
& + \sum_{s=0}^{\lfloor \log_a e+1 \rfloor} (m^2 \cdot n \cdot W_2 \cdot \log n + a^s \cdot m^2 \cdot n \cdot \log n + \log W_e) \\
= \; & m^2 \cdot n \cdot W_{max} \cdot \log n + \log W_{max} \\
& + e \cdot m^2 \cdot n \cdot \log n + \sum_{s=0}^{\lfloor \log_a e+1 \rfloor} (m^2 \cdot n \cdot W_2 \cdot \log n + \log W_e) \\
\leq \; & m^2 \cdot n \cdot W_{max} \cdot \log n + \log W_{max} \\
& + e \cdot m^2 \cdot n \cdot \log n + (\log_a e + 2)(m^2 \cdot n \cdot W_e \cdot \log n + \log W_e) \\
= \; & m^2 \cdot n \cdot \log n (W_{max} + e + (\log_a e + 2) W_e) + (\log_a e + 2) \log W_e + \log W_{max} \\
= \; & \mathcal{O}(m^2 \cdot n \cdot \log n (W_{max} + e + W_e \cdot \log e))
\end{aligned}
$$

Where the classical algorithm made a^s calls to getKey for iteration s of the loop, Grover's algorithm makes $\lceil \frac{\pi}{4} \cdot a^{\frac{s}{2}} \rceil + 1$ calls [10,11,5].The time complexity of total calls to getKey, made by Grover's algorithm, can be calculated as follows:

$$
\begin{aligned}
&\sum_{s=0}^{\lfloor \log_a e+1 \rfloor} (\lceil \frac{\pi}{4} \cdot a^{\frac{s}{2}} \rceil + 1) \cdot m^2 \cdot n \cdot \log n \\
&= m^2 \cdot n \cdot \log n \cdot (\sum_{s=0}^{\lfloor \log_a e+1 \rfloor} (\lceil \frac{\pi}{4} \cdot a^{\frac{s}{2}} \rceil + 1)) \\
&\leq m^2 \cdot n \cdot \log n \cdot (2 \log_a e + 4 + \frac{\pi}{4} \cdot \sum_{s=0}^{\lfloor \log_a e+1 \rfloor} a^{\frac{s}{2}}) \\
&\approx m^2 \cdot n \cdot \log n \cdot (2 \log_a e + 4 + \frac{\pi}{4} \cdot \int_{s=0}^{\lfloor \log_a e+1 \rfloor} a^{\frac{s}{2}}) \\
&= 2 \cdot m^2 \cdot n \cdot \log n \cdot (\log_a e + 2 + \frac{\pi}{4} + \frac{\pi \cdot a}{4 \ln a} \cdot \sqrt{e}) \\
&= \mathcal{O}(\sqrt{e} \cdot m^2 \cdot n \cdot \log n)
\end{aligned}
$$