Tradeoff Cryptanalysis of Memory-Hard Functions

Alex Biryukov and Dmitry Khovratovich

University of Luxembourg
alex.biryukov@uni.lu, khovratovich@gmail.com

Abstract. We explore time-memory and other tradeoffs for memory-hard functions, which are supposed to impose significant computational and time penalties if less memory is used than intended. We analyze two schemes: Catena, which has been presented at Asiacrypt 2014, and Lyra2, the fastest finalist of the Password Hashing Competition (PHC).

We demonstrate that Catena's proof of tradeoff resilience is flawed, and attack it with a novel *precomputation tradeoff*. We show that using $M^{2/3}$ memory instead of M we may have no time penalties. We further generalize our method for a wide class of schemes with predictable memory access. For Lyra2, which addresses memory unpredictability (depending on the input), we develop a novel *ranking tradeoff* and show how to decrease the time-memory and the time-area product by significant factors. We also generalize the ranking method for a wide class of schemes with unpredictable memory access[1].

Keywords: password hashing, memory-hard, Catena, tradeoff, cryptocurrency, proof-of-work.

1 Introduction

Memory-hard functions are a fast emerging trend which has become a popular remedy to the hardware-equipped adversaries in various applications: cryptocurrencies, password hashing, key derivation, and more generic Proof-of-Work constructions. It was motivated by the rise of various attack techniques, which can be commonly described as optimized exhaustive search. In cryptocurrencies, the hardware arms race made the Bitcoin mining [28] on regular desktops tremendously inefficient, as the best mining rigs spend 30,000 times less energy per hash than x86-desktops/laptops[2]. This causes major centralization of the mining efforts which goes against the democratic philosophy behind the Bitcoin design. This in turn prevents wide adoption and use of such cryptocurrency in economy, limiting the current activities in this area to mining and hoarding, whith negative effects on the price. Restoring the ability of CPU or GPU mining by the use of memory-hard proof-of-work functions may have dramatic effect on cryptocurrency adoption and use in economy, for example as a form of decentralized micropayments [13]. In password hashing, numerous leaks of hash databases triggered the wide use of GPUs [3,35], FPGAs [26] for password cracking with a dictionary. In this context, constructions that intensively use a lot of memory seem to be a countermeasure. The reasons are that memory operations have very high latency on GPU and that the memory chips are quite large and thus expensive on FPGA and ASIC environments compared to a logic core, which computes, e.g. a regular hash function.

Memory-intensive schemes, which bound the memory bandwidth only, were suggested earlier by Burrows et al. [8] and Dwork et al. [16] in the context of spam countermeasures. It was quickly realized that to be a real countermeasure, the amount of memory shall also be bounded [17], so that memory must not be easily traded for computations, time, or other resources that are cheaper on certain architecture. Schemes that are resilient to such tradeoffs are called *memory-hard* [20,31].

Attack costs. The total attack cost, however, is quite difficult to estimate. Development, production, and running costs may vary significantly depending on the architecture and scale. It is widely suggested that the costs can be approximated by the time-area product [9, 11, 27, 36]. The latter notion is still, however, architecture-dependent, as it requires to know the area ratio between memory and computational logic, or the *core-memory ratio*. What can be estimated independently

[1] Both Catena and Lyra2 were significantly changed after we have shared our analysis with the designers.

[2] The estimate comes from the numbers given in [6]: the best ASICs make 2^{32} hashes per joule, whereas the most efficient laptops can do 2^{17} hashes per joule

of the architecture are the computational penalty given certain memory reduction (*computation-memory tradeoff*) and the time penalty given the memory reduction and unlimited parallelism (*time-memory tradeoff*). These two tradeoffs, the memory bandwidth limit (which may increase time), and the core-memory ratio (which affects area) can then be combined to figure out the actual time-area product and thus approximate the attack costs.

Computation-memory tradeoffs and pebbling games. Constructions that put certain bounds on computation-memory tradeoffs are well known since 1970s, where they appeared as time-space tradeoffs in the context of pebbling games on graphs [23,30]. In this framework we consider iterative functions H such that each intermediate output is a function of some previously computed values:

$$X_0 = Input, \quad X_j = F(X_{\phi_1(j)}, X_{\phi_2(j)}, \ldots, X_{\phi_k(j)}), \quad 1 < j < T, \ H(X_0) = X_T.$$

where ϕ_i are some indexing functions and F is some internal compression function. The entire function H is then represented as a directed acyclic graph on T vertices. Computing H with $S < T$ memory is then equivalent to running the pebbling game with S pebbles and following rules:

- the input vertex X_0 can be pebbled at any time;
- any vertex can be unpebbled at any time;
- a non-input vertex can be pebbled if there is a free pebble and all its predecessors have been pebbled.

Seminal results in this area include [23], which shows that all graphs of size T with maximum in-degree k can be pebbled with $c_k \frac{T}{\log T}$ pebbles, where c_k is a constant for fixed k, though the result is non-constructive. There exist graphs that can not be pebbled with fewer pebbles, and there are graphs that exhibit dramatic computation-memory tradeoffs when pebbled with this number of pebbles, i.e. the computational penalties grow exponentially as memory is reduced. The so-called superconcentrators [33] are prominent examples of such graphs. These constructions have been partially used by Dwork et al. [17] to ensure harsh computation-memory tradeoffs, and by Dziembowski et al. in the proof-of-space protocols [18].

Disadvantage of classical constructions and new schemes. The provably tradeoff-resilient constructions from [33] and their applications in [17, 18] have serious performance problems. They are terribly slow for modern memory sizes. A superconcentrator requiring N blocks of memory makes $O(N \log N)$ calls to F. As a result, filling, e.g., 1 GB of RAM with 256-bit blocks would require dozens of calls to F per block ($C \log N$ calls for some constant C). This would take several minutes even with lightweight F and is thus intolerable for most applications like web authentication or cryptocurrencies. Using less memory, e.g., several megabytes, does not effectively prohibit hardware adversaries.

This has been an open challenge to construct a reasonably fast and tradeoff-resilient scheme. Since the seminal paper by Dwork et al. [17] the first important step was made by Percival, who suggested scrypt [31]. The idea of scrypt was quite simple: fill the memory by an iterative hash function and then make a pseudo-random walk on the blocks using the block value as an address for the next step. However, the entire design is somewhat sophisticated, as it employs a stack of subfunctions and a number of different crypto primitives. Under certain assumptions, Percival proved that the time-memory product is lower bounded by some constant. The scrypt function is used inside cryptocurrency Litecoin [4] and is now adapted as an IETF standard for key-derivation [5]. However, the tradeoff resilience of scrypt might not be sufficient, as the Litecoin ASIC miners are more efficient than CPU miners by the factor of 1000 [1], possibly due to not so high performance of scrypt.

The need for even faster, simpler, and possibly more tradeoff-resilient constructions was further emphasized by the ongoing Password Hashing Competition [2], which has recently selected 9 finalists out of the 24 original submissions. Notable examples are Catena [19], just presented at Asiacrypt 2014 with a security proof based on [25], and Lyra2 [24], which claims performance up to 1 GB/sec and which was quickly adapted as a cryptocurrency hash function [7]. The tradeoff resilience of both constructions has not been challenged so far.

Data-dependent and data-independent schemes. The existing schemes can be categorized according to the way they access memory. The *data-independent schemes* [14, 19, 37] accesses the memory blocks according to some predefined pattern, independently of the inputs. This allows the adversary to precompute a missing block just by the time it is requested, thus saving memory with no time penalty if extra computing power is available.

The *data-dependent* schemes [31, 32] calculate the block addresses on-the-fly, which prohibits such precomputation and allows to bound the time-memory tradeoffs. However, the data-dependent schemes are vulnerable to cache-timing attacks, as the memory access pattern reveals some information on the inputs. For applications like password hashing and key derivation this might be a crucial vulnerability, as an adversary can run his own memory-intensive process alongside the hashing scheme. Then the cache misses yield information about r_i and thus shorten the password testing. Such assumptions, being unlikely for authentication services, were shown practical in the key recovery side-channel attacks in the cloud settings [34].

There exist hybrid schemes [24], which first run a data-independent phase and then a data-dependent one, so that a cache-timing information reduces the exhaustive search workload just by a small factor.

Our contributions. We present tradeoff attacks on Catena and Lyra2 and generalize them to wide classes of data-dependent and data-independent schemes. We show that the original security proof for Catena is flawed and the computation-memory product can be kept constant while reducing the memory. Together with the data-independent memory access pattern of Catena, this allows to reduce the time-memory and the time-area products by a large factor, up to running the scheme with $M^{0.66}$ memory instead of M with no time penalty. The attack algorithm is then generalized for a wide class of data-independent schemes as a *precomputation method.*

We also attack both phases of the hybrid scheme Lyra2. For the data-independent phase we apply a modification of the attack on Catena, which works even better. For the data-dependent phase we develop a generic randomized algorithm, which additionally exploits the iterative structure of function F within Lyra2. Altogether, we show how to decrease the time-memory product by the factor of 8 for the full Lyra2. The attack algorithm for the data-dependent part is then generalized as a *ranking method.*

To the best of our knowledge, both methods are the first generic attacks so far on data-dependent or data-independent schemes.

Related work. So far there have been only a few attempts to develop tradeoff attacks on memory-hard functions. A simple tradeoff for scrypt has been known in folklore and was recently formalized in [19]. Alwen and Serbinenko analyzed a simplified version of Catena in [9]. Designers of Lyra2 and Catena attempted to attack their own designs in the original submissions [19, 24]. Simple analysis of Catena has been made in [15].

2 Cryptanalysis of Catena

2.1 Description

Short history. Catena was first published on ePrint [19] and then submitted to the Password Hashing Competition. Eventually the paper was accepted to Asiacrypt 2014 [20]. In the middle of the reviewing process, we discovered and communicated our attack on Catena to the authors. The authors have changed Catena for the camera-ready version of the Asiacrypt paper in order to mitigate our attack. In this paper we consider the original submission to PHC, Catena v.1.

Specification. Catena is essentially a mode of operation over the hash function F, which can be instantiated by Blake2b [10] or SHA-2 [29]. The functional graph of Catena is determined by the time parameter λ (values $\lambda = 3, 4$ are recommended) and the memory parameter n, and can be viewed as $(\lambda + 1)$-layer graph with 2^n vertices in each layer (denoted by Catena-λ). We denote the

X-th vertex in layer l (both count from 0) by $[X]^l$. With each vertex we associate the corresponding output of the hash function H and denote it by $[X^l]$ as well. The outputs are stored in the memory, and due to the memory access pattern it is sufficient to store only 2^n blocks at each moment. The hash function H should have 512-bit output, so the total memory requirements are 2^{n+6} bytes.

First layer is filled as follows

- $[0]^0 = G(P, S)$, where G invokes 3 calls to F;
- $[i]^0 \leftarrow F([i-1]^0),\ 1 \leq i \leq 2^n - 1$.

The memory access pattern at the next layers is determined by the *bit-reversal permutation* ϕ. Each index is viewed as an n-bit string and is transformed as follows:

$$\phi(x_1 x_2 \ldots x_n) = x_n x_{n-1} \ldots x_1, \text{ where } x_i \in \{0, 1\}.$$

The layers are then computed as

- $[0]^j = F([0]^{j-1} \,||\, [2^n - 1]^{j-1})$;
- $[i]^j = F([i-1]^j \,||\, [\phi(i)]^{j-1})$.

Thus to compute $[X]^l$ we need $[\phi(X)^{l-1}]$. The latter can be then overwritten. An example of Catena with $\lambda = 2$ and $n = 3$ is shown at Figure 1.

The bit-reversal permutation is supposed to provide memory-hardness. The intuition is that it maps any segment to a set of blocks that are evenly distributed at the upper layer.

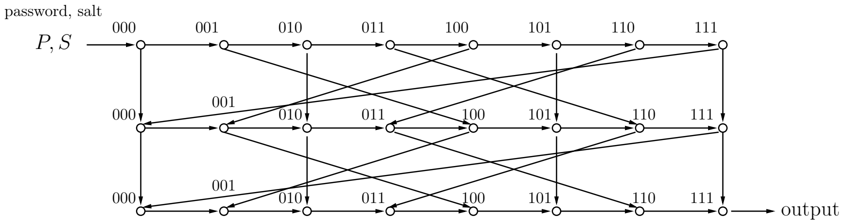

Fig. 1. Catena-2 with $n = 3$. 3 layers, 8 vertices per layer.

Original tradeoff analysis. The authors of Catena provide two types of security bounds against tradeoff attacks. Recall that Catena-λ can be computed with $\lambda 2^n$ calls to F using 2^n memory blocks. The Catena designers demonstrated that Catena-λ can be computed using λS memory blocks with time complexity[3]

$$T \leq 2^n + 2^n \left(\frac{2^n}{2S}\right)^{\lambda-1} + 2^n \left(\frac{2^n}{2S}\right)^{\lambda}$$

Therefore, if we reduce the memory by the factor of q, i.e. use only $\frac{2^n}{q}$ blocks, we get the following penalty:

$$P_\lambda(q) \approx \left(\frac{q}{2}\right)^{\lambda}. \tag{1}$$

The second result is the lower bound for tradeoff attacks with memory reduction by q:

$$P_\lambda(q) \geq \Omega\left(q^\lambda\right). \tag{2}$$

However the constant in $\Omega()$ is too small (2^{-18} for $\lambda = 3$) to be helpful in bounding tradeoff attacks for small q. More importantly, the proof is flawed: the result for $\lambda = 1$ is incorrectly generalized for larger λ. The reason seems to be that the authors assumed some independence between the layers, which is apparently not the case (and is somewhat exploited in our attack).

In the further text we demonstrate a tradeoff attack yielding much smaller penalties than Eq. (1) and thus asymptotically violating Eq. (2).

[3] This result is a part of Theorem 6.3 in [19].

2.2 Our tradeoff attack on Catena

The idea of our method is based on the simple fact that

$$\phi(\phi(X)) = X,$$

where X can be a single index or a set of indices. We exploit it as follows. We partition the memory into *segments* of length 2^k for some integer k, and store the first block of every segment. As the index of such a block ends with k zeros, we denote the set of these blocks as $[*^{n-k}0^k]$.

Consider a single segment $[AB*^k]$, where A is a k-bit constant, B is a $n-2k$-bit constant. Then

$$\phi([AB*^k]) = [*^k\phi(B)\phi(A)].$$

Blocks $[*^k\phi(B)\phi(A)]$ belong to 2^k segments that have $\phi(B)$ in the middle of the index. Denote the union of these segments by $[*^k\phi(B)*^k]$. Now note that

$$\phi([*^k\phi(B)*^k]) = [*^kB*^k],$$

and

$$\phi(\phi([*^kB*^k])) = [*^kB*^k].$$

Therefore, when we iterate the permutation ϕ, we are always within some 2^k segments. Now it should be clear how we should compute a segment having only blocks $[*^{n-k}0^k]$ stored.

Consider, for instance, segment $[AB*^k]^4$, i.e. blocks with indices starting with AB at layer 4. It can be computed as follows:

- Recompute blocks $[*^kB*^k]^0$ (2^k segments) using stored $[*^{n-k}0^k]^0$, which requires 2^{2k} calls to F;
- Use $[*^kB*^k]^0$ and stored $[*^{n-k}0^k]^1$ to compute $[*^k\phi(B)*^k]^1$ (2^{2k} calls to F, as all inputs are known);
- Use $[*^k\phi(B)*^k]^1$ to compute $[*^kB*^k]^2$ (2^{2k} calls to F);
- Use $[*^kB*^k]^2$ to compute $[*^k\phi(B)\,\phi(A)]^3$ ($\phi(A)2^k$ calls to F);
- Use $[*^k\phi(B)\,\phi(A)]^3$ to compute $[AB*^k]^4$ (2^k calls to F).

If we generalize it to a segment at level t, we obtain that

$$(t-1)2^{2k} + \phi(A)2^k + 2^k \tag{3}$$

hash function calls are needed to compute the segment $[AB*^k]^t$. Therefore, the total cost of computing layer t is

$$\begin{aligned} C(t) &= \sum_A\sum_B((t-1)2^{2k} + \phi(A)2^k + 2^k) = \\ &= \sum_A((t-1)2^n + \phi(A)2^{n-k} + 2^{n-k}) = (t-1)2^{n+k} + 2^{n+k-1} + 2^n = (t-\frac{1}{2})2^{n+k} + 2^n. \end{aligned} \tag{4}$$

As a result, computing layer t costs $((t-0.5)2^k+1)$ times as much as computing with 2^n memory blocks. Our methods requires to store $t2^{n-k}$ blocks as segment beginnings and 2^{2k} blocks for intermediate computations.

We can calculate the total complexity of computing Catena-λ for some k. For example, Catena-4 with $k=3$ can be computed with

$$(1+5+13+21+29)2^n = 69\cdot 2^n$$

calls to F using $4\cdot 2^{-3} = \frac{1}{2}$ of memory. As the full-memory execution requires 52^n calls to F, the *computational penalty* is $\frac{69}{5} = 13.8$.

Final tradeoffs for Catena. Using Equation (4) for different k, we get computational-memory tradeoffs for Catena-3 and Catena-4, which are summarized in Table 1. For Catena-3 we have to distribute $2^n/2^l$ blocks to 3 layers, so we use segments of different lengths (2^k and 2^{k+1}, more details in Appendix A.1).

The *time-memory tradeoff*, which assumes unlimited parallelism, for Catena is even more dramatic due to its predictable addressing. Our method allows any $k < n/3$, thus the time can be kept the same $\lambda 2^n$ block computations using only $2^{2n/3}$ memory blocks instead of 2^n.

Estimates for the time-area product. For a certain architecture we can speculate how the actual time-area product changes when we apply our tradeoff. Assume that we use Dynamic RAM and Blake2b as the underlying hash function and the following reference implementations:

- The 50-nm DRAM implementation [21], which takes 550 mm^2 per GByte;
- The 65-nm Blake-512 implementation [22], which takes about 0.1 mm^2.

Suppose that the implementations are taken as is, without scaling to the same feature size. Then the core-memory ratio is about 2^{12}. Therefore if the computational penalty reaches 2^{12}, the total area consumed by Blake2b cores would be about the same as the original memory area. If we reduce the memory by the factor of 2^5, then the computational penalty would be 2^7 for Catena-3, so the memory area would be approximately equal to the Blake2b area. Therefore, the time-area product can be reduced by the factor of 30.

On other architectures the results can be quite different, and we expect that an adversary would choose one that maximizes the tradeoff effect, so the actual impact of our attack can be even higher.

Violation of Catena lower bound Our method shows that the Catena lower bound is wrong. If we summarize the computational costs for λ layers, we obtain the following computational penalty for the memory reduction by the factor of q:

$$C_\lambda(q) = O(\lambda^3 q),$$

which is asymptotically smaller than the lower bound $\Omega(q^\lambda)$ (Equation (2)) from the original Catena paper [19].

3 Cryptanalysis of Lyra2

3.1 Description of Lyra2

Lyra2 [24] is a PHC finalist, notable for its high memory filling rate (up to 1 GB/sec). Very recently, Lyra2 has been significantly changed for the second round of the competition. This section describes the original submission to PHC [24].

Lyra2 is a hybrid hashing scheme, which uses data-independent addressing in the first phase and data-dependent addressing in the second phase. Lyra2 operates on blocks of 768 bits (96 bytes)

Memory fraction	Catena-3		Catena-4	
	Computational penalty			
	Our	[19]	Our	[19]
$\frac{1}{2}$	7.4	36.2	13.8	512
$\frac{1}{4}$	15.5	252	26.6	7373
$\frac{1}{8}$	30.1	1872	52	2^{17}
$\frac{1}{2^l}$	$2^{l+1.9}$	2^{3l}	$2^{l+2.8}$	$2^{4l+1.5}$

Table 1. Computation-memory tradeoff for Catena-3 and Catena-4.

each, and fills the memory with $2^n \cdot C$ such blocks, where n and C are parameters, and C is by default set to 128 [24, p.39]. In this paper we use $C = 128$. The entire memory is represented as a $(2^n \times C)$-matrix M, and we refer to its components as rows and columns. Rows are denoted by $M[i]$.

Lyra2 has two main phases: the single-iteration **Setup phase**, where the memory is addressed data-independently, and the multiple-iteration **Wandering phase**, where the memory is addressed data-dependently. The number T of iterations in the Wandering phase can be as low as 1, and we take this value in our analysis.

Setup phase. The first phase fills rows sequentially from $M[0]$ to $M[2^n - 1]$ as follows:

```
$M[0], M[1] \leftarrow G(\text{Password,Salt});$
for $i$ from 2 to $2^n - 1$
        $M[i] \leftarrow F(M[i-1], M[\phi(i)]);$
        $M[\phi(i)] \leftarrow M[\phi(i)] \oplus \overleftarrow{M[i]}.$
```

Here

$$\phi(i) = 2^k - i,$$

where 2^k is the smallest power of 2 that is not smaller than i, $\overleftarrow{M[]}$ stands for the left rotation of each 768-bit word by 32 bits, and G is a cryptographic hash function.

The following details of F are relevant to our attack:

- Function F is stateful: it operates on the 1024-bit state S, which is preserved between rows.
- Function $F(X, Y)$ processes columns X_i, Y_i of X and Y sequentially. The internal state undergoes C rounds (similarly to the duplex-sponge construction [12]), where in round i column Z_i of the output Z is produced as follows:

$$\begin{aligned} S &\leftarrow S \oplus X_i \oplus Y_i; \\ S &\leftarrow P(S); \\ Z_i &\leftarrow 768 \text{ least sign. bits of } (S). \end{aligned}$$

  Here P is a single round of the Blake2b internal permutation [10]. We do not exploit any specific property of P. Thus F can be seen as a duplex-sponge instantiated with a Blake2b round function.

We remind the reader that Z is used not only to produce a new row $M[i]$ but also to overwrite the row $M[2^k - i]$.

Wandering phase. The Wandering phase transforms the blocks produced in the Setup phase. First, it reverses the ordering. Then it operates similarly to the Setup phase, but the second input block to F is taken pseudo-randomly:

```
for $i$ from 1 to $2^n - 1$
        $r_i \leftarrow g(M[i-1], i-1);$
        $M[i] \leftarrow M[i] \oplus F(M[i-1], M[r_i]);$
        $M[r_i] \leftarrow M[r_i] \oplus \overleftarrow{F(M[i-1], M[r_i])}.$
```

Here g merely truncates the first input to the least significant 32 bits and xores with the second input. All indices are computed modulo 2^n.

3.2 Tradeoff attack on the Setup phase of Lyra2

Our strategy for the Setup phase is similar to the one for Catena. Again, we exploit the properties of the indexing function ϕ.

Let us denote a segment $\{M[i], M[i+1], \ldots, M[j]\}$ by $M[i:j]$. Consider a, b such that $2^{k-1} < a < b < 2^k$. Then

$$\phi([a:b]) = [(2^k - b) : (2^k - a)].$$

Thus to construct a single segment we need another segment of the same length. This suggests the following strategy for computing 2^n rows in the Setup phase. We store:

- First 2^{n-l} rows $M[0], \ldots, M[2^{n-l} - 1]$ for some $l > 0$ (parameter of the attack).
- We split rows from $M[2^{n-l}]$ to $M[2^n - 1]$ into segments of length q for some $q > 0$. Store the entire state S at the start of each segment.

Then to compute segment $M[a : a+q-1], 2^{k-1} < a < 2^k$ we have to compute $M[\phi(a : a+q-1)]$, which has been updated when computing segments between 2^{k-2} and 2^{k-1}. Eventually we reach the stored 2^{n-l} rows. To compute $M[a : a+q-1], 2^{k-1} < a < 2^k$ we need to compute a segment in the interval $[2^i : 2^{i+1}]$ for each $n - l < i < k$ (Figure 2).

Let us figure out the memory reduction and the computational overhead of this procedure. We store 2^{n-l} first rows and then $\frac{2^n}{96q}$ to store the starting states. For segments between rows $M[2^{n-l}]$ and $M[2^{n-l+1}]$ we need 1 call to F per row, as there is no recomputation. For segments between rows $M[2^{n-l+1}]$ and $M[2^{n-l+2}]$ we need 2 calls to F per row, and so on. In general, we make

$$(k - n + l)q \tag{5}$$

calls to F to compute a segment of length q between row indices 2^k and 2^{k+1}.In total we make

$$\underbrace{2^{n-l}}_{M[0:2^{n-l}-1]} + \underbrace{2^{n-l}}_{M[2^{n-l}:2^{n-l+1}-1]} + 2 \cdot 2^{n-l+1} + 3 \cdot 2^{n-l+2} + \cdots + l2^{n-1} \leq (l - 0.5)2^n$$

calls to F. Therefore, the computational penalty is $(l - 0.5)$ and is independent of q. The optimal q is then equal to $2^{n/2}$ and the memory requirements are $2^{n-l} + 2^{n/2+1}$ blocks.

Therefore, the memory reduction by the factor of s yields the computational penalty of about $\log s$ (Table 2). There is no time penalty as long as we afford $\log s$ computing cores.

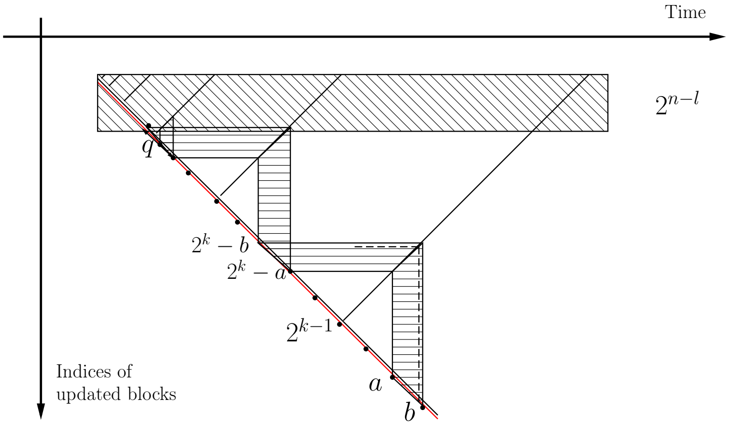

Fig. 2. Computing segment of length q with precomputation method in the Setup phase of Lyra2.

Memory fraction	Computational penalty	[24]
$1/4$	1.5	2
$1/8$	2.5	4
2^{-l}	$l - 0.5$	2^{l-1}

Table 2. Computational-memory tradeoff for the Setup phase of Lyra2: our method and designers' analysis.

Access complexity of a single row. In the next phase we will need to calculate the cost of recomputing a single row rather than a segment. To compute a single row, we need to recompute $(l-0.5)$ segments on average, so the average recomputation complexity is:

$$A = q(l - 0.5). \tag{6}$$

3.3 Tradeoff attack on the Wandering phase of Lyra2

The memory access pattern in the Wandering phase depends on the input and thus is not known to the adversary a priori. We present a method that adaptively choose the rows to store according to their potential recomputation complexities. The computational penalty of our method is also data-dependent so we made a series of experiments and calculated the average penalty values.

First, we introduce some notation. Suppose that we store a fraction of rows in the memory. If at some point we need the row $M[i]$ that is not stored, we have to recompute it. The number of calls to F needed to recompute $M[i]$ is called its *access complexity* and is denoted $A(i)$. For the Wandering phase of Lyra2 we compute $A(i)$ on-the-fly according to the following rules:

- In the beginning all $A(i)$ are equal to 0.
- At step i, if we store the output of F, then $A(i)$ and $A(r_i)$ remain unchanged.
- At step i, if we do not store the output of F, we set

$$A(i) \leftarrow A(r_i) \leftarrow A(i) + A(r_i) + A(i-1) + 1.$$

The idea of our *ranking tradeoff method* is to store the list of the largest access complexities and to store a block if the access complexity of the referral block is among the highest (Figure 6):

1. Split the memory into segments of length q rows and store the starting row of each segment (for some integer q);
2. Store all r_i;
3. Store all access complexities in the sorted list;
4. For each i, if $A(r_i)$ is among the $2^n/l$ highest access complexities, we store the output of F for this row (which is used to update rows i and r_i).

To store the access complexities, we allocate 16 bits per row, which accounts for $1/6144$ of the total memory for $C = 128$. Here q and l are parameters, and we found experimentally that the best tradeoffs are delivered by $l = 2q$.

To calculate the penalties of our tradeoff method, we made the series of experiments[4]. For each $2 \leq q \leq 80$ we took $l = 2q$, generated 100 Lyra2 executions with 4096 rows, where r_i is selected randomly, and applied our method to it. Then we calculated the fraction of rows that we stored during the computation and the total amount of recomputation costs. We obtained a set of values (α, p), where α is the memory fraction and p is the computational penalty. For each value $1/i$ (i is integer) we computed the average penalty for cases when the memory fraction $1/\alpha \in [i - 0.1; i + 0.1]$ and obtained the results in Table 3. Additionally, we computed the average depth of the recomputation tree, which directly affects the total time.

[4] We plan to make the source code for our experiments public and can provide it to interested reviewers.

Memory fraction	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$	$\frac{1}{10}$	$\frac{1}{12}$	$\frac{1}{16}$
Computation penalty	2.7	10.4	75	1071	2^{14}	2^n	2^n	2^n	2^n	2^n
Depth penalty	2.4	4.8	8.9	15.4	23.8	35.7	49	83	124	193

Table 3. Average computational and depth penalties for the ranking method on the Wandering phase of Lyra2, without exploiting the row pipelining.

3.4 Computation-memory tradeoff on the full Lyra2

To run the attack on the full Lyra2 with $1/l$ of memory, we have to split the available memory between Setup and Wandering phases. Suppose that we allocate α of memory for the Setup phase and β of memory for the Wandering phase. Let $P_S(\alpha)$ be the penalty of running the Setup phase with α of memory, $P_R(\alpha)$ be the average access complexity of a single row from the Setup phase run with α of memory (Eq. (6)), and $P_W(\beta)$ be the penalty of running the Wandering phase with β of memory (Table 3). Then the total memory reduction will be $\alpha + \beta$. To estimate the time penalty, we note that in our tradeoff for the Wandering phase, each recomputation requests as many rows from the Setup phase as many hash calls is made in the Wandering phase. Therefore, the total time penalty would be estimated as

$$P(\alpha+\beta) = \frac{P_S(\alpha)2^n + P_R(\alpha)P_W(\beta)2^n}{2^{n+1}},$$

as we construct $2 \cdot 2^n$ blocks in two phases. Taking $l = -\log_2 \alpha$ and $q = 1$ in Eq. (6), we get the results in Table 4. For reductions larger than the factor 4 we essentially have to recompute all rows, so the computational penalty would be 2^n. However, due to pipelining (see below), only 1 column of each row has to be stored in the memory, plus the internal state of F.

Memory fraction (Wandering + Setup)	Computational penalty	[24]
$\frac{1}{2} = 1/3 + 1/8$	14.2	2^{14}
$\frac{1}{3} = 1/4 + 1/16$	133	$2^{14.5}$
$\frac{1}{4} = 1/5 + 1/16$	1876	2^{15}

Table 4. Comparison of computation-memory tradeoffs for the full Lyra-2: our ranking method and original estimates (taken from [24, p.24] for 2^{17} rows).

3.5 Time-memory tradeoff on the full Lyra2

Even though our tradeoff method impose quite high computational penalties for using 1/5 of memory and lower, a practical attack can still use these fractions if the computational cores are rather small. As an ASIC-equipped attacker would be able to parallelize the computations, the time complexity of the tradeoff attack is actually determined by the depth of the recomputation tree. Indeed, if a row needs 100 calls to F to recompute, but this can be done in a tree of depth 10, then this step takes only 10 times as much time (assuming sufficient memory bandwidth). The average depth values for the Wandering phase are given in Table 3 and are quite high, though lower than the computational penalties.

However, we are able to exploit the fact that Lyra2 produces blocks of a row columnwise. Therefore, we have to make D calls to P to compute the first column of the block, whereas computation of other columns can be pipelined (Figure 3): the second column of the deepest tree level can be computed simultaneously with the first column of one level higher. To compute all 128 columns, we spend time needed to compute $D + 128$ columns only, so the actual time penalty is $1 + D/128$. Therefore, the total time penalty can be calculated as follows:

$$D(\alpha+\beta) = \frac{D_S(\alpha) + \frac{D_R(\alpha)+D_W(\beta)}{128} + 1}{2},$$

where $D_S(\alpha) = 1 - (\log\alpha)/256$ is the average depth penalty in the Setup phase, $D_R(\alpha) = -\log\alpha - 0.5$ is the average depth penalty for accessing the row from the Setup phase, and $D_W(\beta)$ is the depth penalty for the Wandering phase given in Table 3.

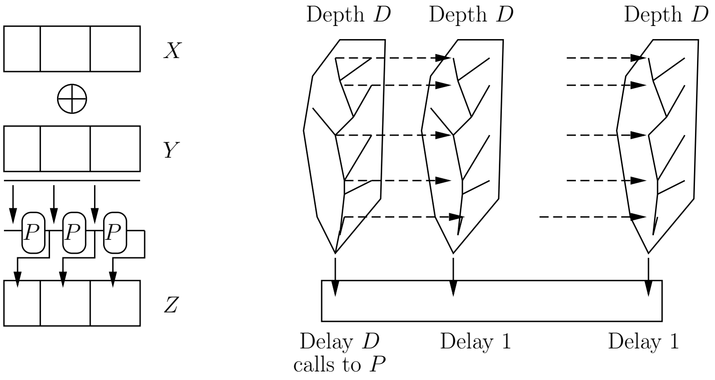

Fig. 3. Pipelining the row computation in Lyra2: only the first column is computed with delay D.

The results are given in Table 5. We conclude that the time-memory product for Lyra2 can be decreased by the factor of 8.

The *time-area product* is more difficult to estimate. If we take reference area values as in Section 2, we obtain that the time-area product can be reduced by the factor of 3 if the memory is reduced by the same factor. If the memory is reduced further, we have to place 2^n cores on the chip, which takes about the same area as the total memory.

Memory fraction (Wandering + Setup)	Time penalty	Time-memory product
$\frac{1}{2} = 1/3 + 1/8$	1.03	0.52
$\frac{1}{3} = 1/4 + 1/16$	1.05	0.35
$\frac{1}{4} = 1/5 + 1/16$	1.08	0.28
$\frac{1}{8} = 1/10 + 1/64$	1.37	0.17
$\frac{1}{16} = 1/17 + 1/256$	1.93	0.12

Table 5. Time-memory tradeoff for the full Lyra-2 with the ranking method.

4 Generic precomputation tradeoff attack

Now we try to generalize the tradeoff method used in the attacks on Catena and Lyra2 for a class of data-independent schemes. We consider schemes where each memory block is a function of the previous block and some earlier block:

$$M[i] \leftarrow F(M[i-1], M[\phi(i)]), 0 \leq i < N$$

where ϕ is a deterministic function such that $\phi(x) < x$. A group of existing password hashing schemes falls into this category: Catena [19], Pomelo [37], Lyra2 [24] (first phase), Rig [14]. Multiple iterations of such a scheme are equivalent to a single iteration with larger N and an additional restriction

$$x - N_0 \leq \phi(x) < N,$$

so that the memory requirements are $N_0 < N$.

The crucial property of the data-independent attacks is that they can be tested and tuned offline, without hashing any real password. An attacker may spend significant time to search for an optimal tradeoff strategy, since it would then apply to the whole set of passwords hashed with this scheme.

Precomputation method. Our tradeoff method generalizes as follows. We divide memory into segments and store only the first block of each segment. For every segment I we calculate its image $\phi(I)$. Let $\overline{\phi(I)}$ be the union of segments that contain $\phi(I)$. We repeat this process until we get an invariant set U_k:

$$\underbrace{I}_{U_0} \to \underbrace{\overline{\phi(I)}}_{U_1} \to \underbrace{\overline{\phi(\overline{\phi(I)})}}_{U_2} \cdots \to U_k.$$

Then we recompute the unions in this chain from U_k to U_0 and store the just derived unions in the memory. Note that U_i can be computed from U_{i+1} with no penalty. The total amount of calls to F is $\sum_{i\geq 0}|U_i|$, and the penalty to compute I is

$$P = \frac{\sum_{i\geq 0}|U_i|}{|I|}.$$

How efficient the tradeoff is depends on the properties of ϕ and the segment partition, i.e. how fast U_i expands. As we have seen, Catena uses a bit permutation for ϕ, whereas Lyra2 uses a simple arithmetic function or a bit permutation [19, 24]. In both cases U_i stabilizes in size after two iterations. If ϕ is a more sophisticated function, the following heuristics (borrowed from our attacks on data-dependent schemes) might be helpful:

- Store the first T_1 computed blocks and the last T_2 computed blocks for some T_1, T_2 (usually about N/q).
- Keep the list $\mathcal{L}$ of the most expensive blocks to recompute and store $M[i]$ if $\phi(i) \in \mathcal{L}$.

We note that our method is much more powerful than the folklore attack on scrypt [19], as we compute the entire segment using the same precomputed values and allow for a flexible segment length.

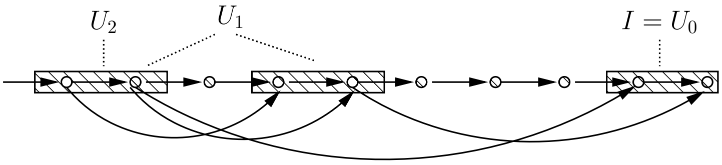

Fig. 4. Segment unions in the precomputation method.

5 Generic ranking tradeoff attack

Now we generalize the attack on the Wandering phase of Lyra2 to a wide class of schemes with data-dependent memory addressing. Such schemes include scrypt [31] and the PHC finalist yescrypt [32]. We consider the functions that are described with the following equations (cf. also Figure 5):

```
$M[0] = \text{Input}$
 for $1 < i < T$
    $r_i = g(M[i-1]);$
    $M[i] = F(M[i-1], M[r_i]).$
```

Here F is some atomic function with computational cost 1, and g is any fast function. This construction and our tradeoff method can be easily generalized to multiple functions F, to stateful functions (like in Lyra2), to multiple inputs, outputs, and passes, etc. However, for the sake of simplicity we restrict to the construction above.

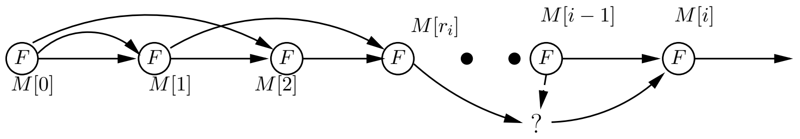

Fig. 5. Data-dependent schemes.

We follow the notation of Section 3.3 and denote the access complexity of block $M[i]$ (number of calls to F needed to recompute $M[i]$) by $A(i)$.

The generic *ranking tradeoff method* works as follows (Figure 6):

1. Split the memory into segments of length q;
2. Set $A(0) = 0$;
3. Keep the sorted list $\mathcal{L}$ of the T/l highest access complexities;
4. Compute blocks sequentially. For block $M[i]$, if $M[r_i]$ is missing, recompute it.
5. If $M[i]$ is the starting block of the segment, we store it and set $A(i) = 0$;
6. If $M[i]$ is not the starting block of the segment, but $A(r_i) \in \mathcal{L}$, we store $M[i]$ and set $A(i) = 0$;
7. If $M[i]$ is not the starting block of the segment, and $A(r_i) \notin \mathcal{L}$, we do not store $M[i]$ and set $A(i) = A(r_i) + A(i-1) + 1$.

Here q and l are parameters. The total computational penalty is computed as

$$C(l) = \frac{\sum_i A(r_i)}{T}$$

We also compute the depth of the recomputation tree for each $A(i) \neq 0$ using the formula $D(i) = \max(D(r_i), D(i-1)) + 1$. We made experiments following the methodology in Section 3.3. The results are given in Table 6.

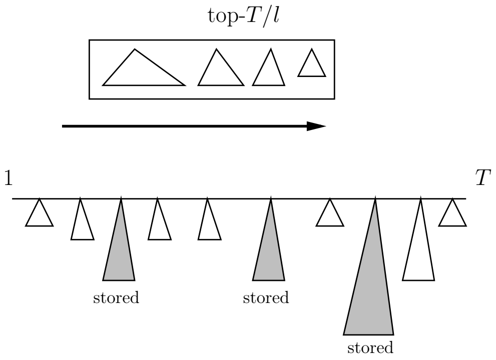

Fig. 6. Outline of the ranking tradeoff method.

Memory fraction $(1/l)$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$	$\frac{1}{9}$	$\frac{1}{10}$	$\frac{1}{11}$
Computation penalty $C(l)$	1.71	2.95	6.3	16.6	55	206	877	4423	$2^{14.2}$	$2^{16.5}$
Depth penalty $D(l)$	1.7	2.5	3.8	5.7	8.2	11.5	15.7	20.8	26.6	32.8

Table 6. Depth and computation penalties for the ranking tradeoff attack for random graphs.

6 Future work

Our tradeoff methods apply to a wide class of memory-hard functions, so our research can be continued in the following directions:

- Application of our methods to other PHC candidates and finalists: yescrypt [32], Rig [14], and the modified Catena and Lyra2.
- Set of design criteria for the indexing functions that would withstand our attacks.
- Study the relevant cost metrics for each application scenario (cryptocurrencies, password hashing, etc.) and make more precise attack cost estimates in these scenarios.
- New methods that directly target schemes that make multiple passes over memory or use parallel cores.
- Design a set of tools that helps to choose a proof-of-work instance in various applications: cryptocurrencies, proofs of space, etc.

7 Conclusion

Tradeoff cryptanalysis of memory hard functions is a young, relatively unexplored and complex area of research combining cryptanalytic techniques with understanding of implementation aspects and hardware constraints. It has direct real-world impact since its results can be immediately used in the on-going arms race of mining hardware for the cryptocurrencies.

In this paper we have analyzed memory-hard functions Catena and Lyra2. We show that Catena v.1 is not memory-hard despite claims and the security proof by the designers', since a hardware-equipped adversary can reduce the attack costs significantly using our tradeoffs. We also show that Lyra2 is more tradeoff-resilient than Catena, though we can still exploit several design decisions to reduce the time-memory and the time-area product by factors 8 and 3 respectively.

We generalize our ideas to the generic precomputation method for data-independent schemes and the generic ranking method for the data-dependent schemes. Our techniques may be used to estimate the attack cost in various applications from the fast emerging area of memory-hard cryptocurrencies to the password-based key derivation.

8 Acknowledgement

We would like to thank the authors of Catena for verifying and confirming our attack.

References

1. Litecoin: Mining hardware comparison. https://litecoin.info/Mining_hardware_comparison.
2. Password Hashing Competition. https://password-hashing.net/.
3. Software tool: John the Ripper password cracker. http://www.openwall.com/john/.
4. *Litecoin - Open source P2P digital currency*, 2011. https://litecoin.org/.
5. *IETF Draft: The scrypt Password-Based Key Derivation Function*, 2012. https://tools.ietf.org/html/draft-josefsson-scrypt-kdf-02.
6. Bitcoin: Mining hardware comparison, 2014. https://en.bitcoin.it/wiki/Mining_hardware_comparison.
7. *Vertcoin: Lyra2RE reference guide*, 2014. https://vertcoin.org/downloads/Vertcoin_Lyra2RE_Paper_11292014.pdf.
8. Martín Abadi, Michael Burrows, Mark S. Manasse, and Ted Wobber. Moderately hard, memory-bound functions. *ACM Trans. Internet Techn.*, 5(2):299–327, 2005.
9. Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. *IACR Cryptology ePrint Archive 2014/238.*
10. Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *ACNS'13*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.
11. Daniel J. Bernstein and Tanja Lange. Non-uniform cracks in the concrete: The power of free precomputation. In *ASIACRYPT'13*, volume 8270 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2013.
12. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *SAC'11,*.

13. Alex Biryukov and Ivan Pustogarov. Proof-of-work as anonymous micropayment: Rewarding a Tor relay. *IACR Cryptology ePrint Archive 2014/1011*. to appear at Financial Cryptography 2015.
14. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Sanadhya. Rig: A simple, secure and flexible design for password hashing. In *Inscrypt'14*, Lecture Notes in Computer Science, to appear. Springer, 2014.
15. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Time memory tradeoff analysis of graphs in password hashing constructions. In *Preproceedings of PASSWORDS'14*, pages 256–266, 2014. available at `http://passwords14.item.ntnu.no/Preproceedings_Passwords14.pdf`.
16. Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In *CRYPTO'03*, volume 2729 of *Lecture Notes in Computer Science*, pages 426–444. Springer, 2003.
17. Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In *CRYPTO'05*, volume 3621 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2005.
18. Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. *IACR Cryptology ePrint Archive 2013/796*.
19. Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. *IACR Cryptology ePrint Archive, Report 2013/525*.
20. Christian Forler, Stefan Lucks, and Jakob Wenzel. Memory-demanding password scrambling. In *ASIACRYPT'14*, volume 8874 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 2014.
21. Bharan Giridhar, Michael Cieslak, Deepankar Duggal, Ronald G. Dreslinski, Hsing Min Chen, Robert Patti, Betina Hold, Chaitali Chakrabarti, Trevor N. Mudge, and David Blaauw. Exploring DRAM organizations for energy-efficient and resilient exascale memories. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2013)*, pages 23–35. ACM, 2013.
22. Frank Gürkaynak, Kris Gaj, Beat Muheim, Ekawat Homsirikamol, Christoph Keller, Marcin Rogawski, Hubert Kaeslin, and Jens-Peter Kaps. Lessons learned from designing a 65nm ASIC for evaluating third round SHA-3 candidates. In *Third SHA-3 Candidate Conference*, March 2012.
23. John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. On time versus space. *J. ACM*, 24(2):332–337, 1977.
24. Marcos A. Simplicio Jr, Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, and Paulo S. L. M. Barreto. The Lyra2 reference guide, version 2.3.2. Technical report, april 2014.
25. Thomas Lengauer and Robert Endre Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *J. ACM*, 29(4):1087–1130, 1982.
26. Katja Malvoni. Energy-efficient bcrypt cracking. Passwords'14 conference, available at `http://www.openwall.com/presentations/Passwords14-Energy-Efficient-Cracking/`.
27. Sourav Mukhopadhyay and Palash Sarkar. On the effectiveness of TMTO and exhaustive search attacks. In *IWSEC 2006*, volume 4266 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2006.
28. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2009. `http://www.bitcoin.org/bitcoin.pdf`.
29. National Institute of Standards and Technology (NIST). *FIPS-180-4: Secure Hash Standard*, March 2012. available at `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`.
30. Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game on graphs. *Mathematical Systems Theory*, 10:239–251, 1977.
31. Colin Percival. Stronger key derivation via sequential memory-hard functions. 2009. `http://www.tarsnap.com/scrypt/scrypt.pdf`.
32. Alexander Peslyak. Yescrypt - a password hashing competition submission. Technical report, 2014. available at `https://password-hashing.net/submissions/specs/yescrypt-v0.pdf`.
33. Nicholas Pippenger. Superconcentrators. *SIAM J. Comput.*, 6(2):298–304, 1977.
34. Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212, 2009.
35. Martijn Sprengers and Lejla Batina. Speeding up GPU-based password cracking. In *SHARCS'12*, 2012. available at `http://2012.sharcs.org/record.pdf`.
36. Clark D. Thompson. Area-time complexity for VLSI. In *STOC'79*, pages 81–88. ACM, 1979.
37. Hongjun Wu. POMELO: A password hashing algorithm. Technical report, 2014. available at `https://password-hashing.net/submissions/specs/POMELO-v1.pdf`.

A Other results for Catena

A.1 Tradeoff with arbitrary fraction of memory

Consider a more general case, when the adversary stores $(1+\alpha)2^{n-k}$ points per layer, for some $0 \leq \alpha \leq 1$. In addition to 2^{n-k} vertices $[*0^k]$ she also stores $\alpha 2^{n-k}$ vertices of form $[*0^{k-1}]$. As a result, we have segments of both lengths 2^k and 2^{k-1}.

For shorter segments the string B is $(n-2k+2)$ bits long, instead of the previous $(n-2k)$-bits long B. First, we require that the short segments are mapped to the short segments by the bit-reversal permutation, and vice versa (this is achieved by having union $[*B*]$ of short segments).

The computation proceeds as follows. When we compute a longer segment, it is the same as with $\alpha = 0$. When we compute a shorter segment, we precompute 2^{2k-2} points at each level, so in total we make (cf. Eq. (3))

$$(t-1)2^{2k-2} + \phi(A)2^{k-1} + 2^{k-1}$$

hash function calls per 2^{k-1} elements. Therefore, Eq. (4) appears as follows:

$$C(t) = \left((1-\alpha/2)t + \frac{\alpha-1}{2}\right)2^{n+k} + 2^n. \tag{7}$$