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 three finalists of the Password Hashing Competition: Catena, which was presented at Asiacrypt 2014, yescrypt and Lyra2.

We demonstrate that Catena's proof of tradeoff resilience is flawed, and attack it with a novel *precomputation tradeoff*. We show that using $M^{4/5}$ memory instead of M we have no time penalties and reduce the AT cost by the factor of 25. We further generalize our method for a wide class of schemes with predictable memory access. For a wide class of data-dependent schemes, which addresses memory unpredictably, we develop a novel *ranking tradeoff* and show how to decrease the time-memory and the time-area product by significant factors. We then apply our method to yescrypt and Lyra2 also exploiting the iterative structure of their internal compression functions.

The designers confirmed our attacks and responded by adding a new mode for Catena and tweaking Lyra2.

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 [29] on regular desktops tremendously inefficient, as the best mining rigs spend 30,000 times less energy per hash than x86-desktops/laptops[1]. 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 [15]. In password hashing, numerous leaks of hash databases triggered the wide use of GPUs [3,34], FPGAs [27] 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. [17] in the context of spam countermeasures. It was quickly realized that to be a real countermeasure, the amount of memory shall also be bounded [18], 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* [21,30]. In fact, the constructions in [18] are so strong that even tiny memory reduction results in a huge computational penalty.

Disadvantage of classical constructions and new schemes. The provably tradeoff-resilient superconcentrators [32] and their applications in [18, 19] 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. [18] the first important step was made by Percival, who suggested scrypt [30].

[1] 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.

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] with 128 KB memory parameter and is now adapted as an IETF standard for key-derivation [5]. scrypt is a notable example of *data-dependent* schemes where the memory access pattern depends on the input, and this property enabled Percival to prove some lower bound on adversary's costs. However, the performance and/or the tradeoff resilience of scrypt are apparently not sufficient to discourage hardware mining: the Litecoin ASIC miners are more efficient than CPU miners by the factor of 100 [1].

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 entries are Catena [20], just presented at Asiacrypt 2014 with a security proof based on [26], and yescrypt and Lyra2 [25], which both claim performance up to 1 GB/sec and which were quickly adapted within a cryptocurrency proof-of-work [7]. The tradeoff resilience of these constructions has not been challenged so far. It is also unclear how possible tradeoffs would translate to the cost

Our contributions. We present a rigorous approach and a reference model to estimate the amortized costs of password brute-force on special hardware using full-memory algorithms or time-space tradeoffs. We show how to evaluate the adversary's gains in terms of area-time and time-memory products via computational complexity and latency of the algorithm.

Then we present our tradeoff attacks on the last versions of Catena and yescrypt, and the original version of Lyra2. Then we generalize them to wide classes of data-dependent and data-independent schemes. For Catena we analyze the faster Dragonfly mode and show that the original security proof for it is flawed and the computation-memory product can be kept constant while reducing the memory. For ASIC-equipped adversaries we show how to reduce the area-time product (abbreviated further by AT) by the factor of 25 under reasonable assumptions on the architecture. The attack algorithm is then generalized for a wide class of data-independent schemes as a *precomputation method.*

Then we consider data-dependent schemes and present the first generic tradeoff strategy for them, which we call the *ranking method.* Our method easily applies to yescrypt and then to the second phase of Lyra2, both taken with minimally secure time parameters. We further exploit the incomplete diffusion in the core primitives of these designs, which reduces the time-memory and time-area products for both designs.

Altogether, we show how to decrease the time-memory product by the factor of 2 for yescrypt and the factor of 8 for Lyra2. Our results are summarized in Table 1. To the best of our knowledge, our methods are the first generic attacks so far on data-dependent or data-independent schemes[2].

	Catena-Dragonfly	Generic 1-pass	yescrypt	Lyra2 v1
Time	$T = 3M$	$T = M$	$T = 4/3M$	$T = 2M$
	Sec. 3	Sec. 5	Sec. 6	App. A
TM loss	200	1.28	2.1	8
AT loss	25	1.28	2.1	3
TM compactness	64	4	5.8	16
AT compactness	64	4	4.5	5

Table 1. Our tradeoff gains on Catena, yescrypt and Lyra2 with minimal secure parameters, 2^{30} memory bytes and reference hardware implementations (Section 2). TM loss is the maximal factor by which we can reduce the time-memory product compared to the full-memory implementation. AT loss is the maximal factor for time-area product reduction. Compactness of TM and AT is the maximal memory reduction factor which does not increase the TM or AT, resp., compared to the default implementation.

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 [20]. 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 [20, 25]. Simple analysis of Catena has been made in [16].

[2] The full version of this paper is available at [14]

Paper outline We introduce necessary definitions and metrics in Section 2. We attack Catena-Dragonfly in Section 3 and generalize this method in Section 4. Then we present a generic ranking algorithm for data-dependent schemes in Section 5 and attack yescrypt with this method in Section 6. The attack on Lyra2 is quite sophisticated and we leave it for Appendix A.

2 Preliminaries

2.1 Syntax

Let $\mathcal{G}$ be a hash function that takes a fixed-length string I as input and outputs tag H. We consider functions that iteratively fill and overwrite memory blocks $X[1], X[2], \ldots, X[M]$ using a compression function F:

$$X[i_j] = f_j(I),\ 1 \leq j \leq s; \tag{1}$$

$$X[i_j] = F(X[\phi_1(j)], X[\phi_2(j)], \ldots, X[\phi_k(j)]),\ s < j \leq T, \tag{2}$$

where ϕ_i are some indexing functions referring to some already filled blocks and f_j are auxiliary hash functions (similar to F) filling the initial s blocks for some positive s.

We say that the function *makes p passes* over the memory, if $T = pM$. Usually p and M are tunable parameters which are responsible for the total running time and the memory requirements, respectively.

2.2 Time-space tradeoff

Let $\mathcal{A}$ be an algorithm that computes $\mathcal{G}$. The *computational complexity* $C(\mathcal{A})$ is the total number of calls to F and f_i by $\mathcal{A}$, averaged over all inputs to $\mathcal{G}$. We do not consider possible complexity amortization over successive calls to $\mathcal{A}$. The *space complexity* $S(\mathcal{A})$ is the peak number of blocks (or their equivalents) stored by $\mathcal{A}$, again averaged over all inputs to $\mathcal{G}$. Suppose that $\mathcal{A}$ can be represented as a directed acyclic graph with vertices being calls to F. Then the *latency* $L(\mathcal{A})$ is the length of the longest chain the graph from the input to the output. Therefore, $L(\mathcal{A})$ is the minimum time needed to run $\mathcal{A}$ assuming unlimited parallelism and instant memory access.

A straightforward implementation of the scheme (1) results in an algorithm with computational complexity T and latency $L = T$ and space complexity M. However, it might be possible to compute $\mathcal{G}$ using less memory. According to [24], any function, that is described by Equation (1) and whose reference block indices $\phi_j(i)$ are known in advance, can be computed using $c_k \frac{T}{\log T}$ memory blocks for some constant c_k depending on the number k of input blocks for F. Therefore, any p-pass function can be computed using less than $M = T/p$ memory for sufficiently large M.

Let us fix some *default algorithm* $\mathcal{A}$ of $\mathcal{G}$ with (C_1, M_1, L_1) being computational and space complexities and latency of $\mathcal{A}$, respectively. Suppose that there is a *time-space tradeoff* given by the family of algorithms[3] $\mathcal{B} = \{B_q\}$ that compute $\mathcal{G}$ using $\frac{M_1}{q}$ space for different q. The idea is to store only one of q memory blocks on average and recompute the missing blocks whenever they are needed. Then we define the *computational penalty* $C_{\mathcal{B}}(q)$ as

$$CP_{\mathcal{B}}(q) = \frac{C(B_q)}{C_1}$$

and *latency penalty* $L_{\mathcal{B}}(q)$.

$$LP_{\mathcal{B}}(q) = \frac{L(B_q)}{L_1},$$

2.3 Attackers and cost estimates

We consider the following attack. Suppose that $\mathcal{G}$ with time and memory parameters (T, M) is used as a password hashing function with $I = (P, S)$, where P is a secret password and S is a public salt. An attacker gets H and S (e.g., from a database leak) and tries to recover P. He attempts a dictionary attack: given a list L of most probable passwords, he runs $\mathcal{G}$ on every $P \in L$ and checks the output.

Definition 1. *Let Φ be a cost function defined over a space of algorithms. Let also $\mathcal{G}_{T,M}$ be a hash function with fixed algorithm $\mathcal{A}_0$ (default algorithm). Then $\mathcal{G}_{T,M}$ is called (α, Φ)-secure if for every algorithm $\mathcal{B}$ for $\mathcal{G}_{T,M}$*

$$\Phi(\mathcal{B}) > \alpha\Phi(\mathcal{A}).$$

[3] As well as $\mathcal{A}$, the family $\mathcal{B}$ admits parallel implementations.

In other words, $\mathcal{G}_{T,M}$ can not be computed cheaper than by the factor of $\frac{1}{\alpha}$.

The cost function is more difficult to determine. We suggest evaluating amortized computing costs for a single password trial. Depending on the architecture, the costs vary significantly for the same algorithm $\mathcal{A}$. For the ASIC-equipped attackers, who can use parallel computing cores, it is widely suggested that the costs can be approximated by the *time-area product* AT [9,11,28,35]. Here T is the time complexity of the used algorithm and A is the sum of areas needed to implement the memory cells and the area needed to implement the cores. Let the area needed to implement one block of memory be the unit of area measurement. Then in order to know the total area, we need *core-memory ratio* R_c, which is how many memory blocks we can place on the area taken by one core.

Suppose that the adversary runs algorithm B_q using M/q memory and l computing cores, thus having computational complexity $C_q = C(B_q)$. The running time is lower bounded by the latency $L_q = L(B_q)$ of the algorithm. If $L_q < C_q/l$, i.e. the computing cores can not finish the work in minimum time, then the time T can be approximated by C_q/l, and the costs are estimated as follows:

$$\mathrm{AT}_{B_q}(l) = \left(lR_c + \frac{M}{q}\right)\frac{C_q}{l} = C_q(R_c + \frac{M}{ql})$$

We see that the costs drop as l increases. Therefore, the adversary would be motivated to push it to the maximum limit C_q/L_q. Thus we obtain the final approximation of costs:

$$\mathrm{AT}_{B_q} = C_qR_c + L_q\frac{M}{q}. \tag{3}$$

Here we assume unlimited memory bandwidth. Taking the bandwidth restrictions into account is even more difficult, as they depends on the relative frequency of the computing core and the memory as well as on the architecture of the memory bus. Moreover, the memory bandwidth of the algorithm depends on the implementation and is not easy to evaluate. We leave rigorous memory bandwidth evaluation and restrictions for the future work.

We recall that the value R_c is depends on the architecture, the function F, and the block size. To give a concrete example, suppose that the block is 64 bytes and F is the Blake-512 hash function. We use the following reference implementations[4]:

- The 50-nm DRAM [22], which takes 550 mm^2 per GByte;
- The 65-nm Blake-512 [23], which takes about 0.1 mm^2.

Then the core-memory ratio is $\frac{2^2 4\cdot 0.1}{550} \approx 3000$. For more lightweight hash functions this ratio will be smaller.

The actual functions F in the designs that we attack are often ad-hoc and have not implemented yet in hardware. Moreover, the numbers may change when going to smaller feature size. To make our estimates of the attack costs architecture-independent, we introduce a simpler metric — the *time-memory product* TM:

$$\mathrm{TM}_{B_q} = L_q\frac{M}{q}, \tag{4}$$

which for not so high computational penalties gives a good approximation of AT.

In our tradeoff attacks, we are mainly interested to compare the AT and TM costs of B_q with that of the default algorithm $\mathcal{A}$. Thus we define the *AT ratio* of B_q as $\frac{\mathrm{AT}_{B_q}}{\mathrm{AT}_{\mathcal{A}}}$, and the *TM ratio* of B_q as $\frac{\mathrm{TM}_{B_q}}{\mathrm{TM}_{\mathcal{A}}}$

We note that for the same TM value the implementation with less memory is preferable, as its design and production will be cheaper. Thus we explore how much the memory can be reduced keeping the AT or TM costs below those of the default algorithm.

Definition 2. *Tradeoff algorithms $\mathcal{B}$ have* AT compactness *q if it is the maximal q such that*

$$\mathrm{AT}_{B_q} \leq \mathrm{AT}_{\mathcal{A}}.$$

Tradeoff algorithms $\mathcal{B}$ have TM compactness *q if it is the maximal q such that*

$$\mathrm{TM}_{B_q} \leq \mathrm{TM}_{\mathcal{A}}.$$

For the concrete schemes we take "minimally secure" values of T, i.e. those that supposed to have (α, Φ)-security for reasonably high α. Unfortunately, no explicit security claim of this kind is present in the design documents of the functions we consider.

[4] We take low-area implementations, as possible parallelism is already taken into account.

Data-dependent and data-independent schemes. The existing schemes can be categorized according to the way they access memory. The *data-independent schemes* Catena [20], Pomelo [36], Argon2i [13] computes $\phi(j)$ independently of the actual password in order to avoid timing attacks like in [33]. Then the algorithm $\mathcal{B}$ that uses less memory can recompute the missing blocks just by the time they are requested. Therefore, it has the same latency as the full-memory algorithm, i.e. $L(\mathcal{B}) = L_0$. For these algorithms the time-memory product can be arbitrarily small, and the minimum AT value is determined by the core-memory ratio.

The *data-dependent* schemes scrypt [30] yescrypt [31], Argon2d [13] compute $\phi(j)$ using the just computed block: $\phi(j) = \phi(j, X_{i_{j-1}})$. Then precomputation is impossible, and for each recomputing block the latency is increased by the latency of the recomputation algorithm, so $L_q > L_0$. There exist hybrid schemes [25], which first run a data-independent phase and then a data-dependent one.

3 Cryptanalysis of Catena-Dragonfly

3.1 Description

Short history. Catena was first published on ePrint [20] and then submitted to the Password Hashing Competition. Eventually the paper was accepted to Asiacrypt 2014 [21]. In the middle of the reviewing process, we discovered and communicated the first attack on Catena to the authors. The authors have introduced a new mode for Catena in the camera-ready version of the Asiacrypt paper, which is resistant to the first attack. The final version of Catena, which is the finalist of the Password Hashing Competition, contains two modes: Catena-Dragonfly (which we abbreviate to Catena-D), which is an extension to the original Catena, and Catena-Butterfly, which is a new mode advertised as tradeoff-resistant. In this paper we present the attack on Catena-Dragonfly, which is very similar to the first attack on Catena.

Specification. Catena-D is essentially a mode of operation over the hash function F, which is be instantiated by Blake2b [10] in the full or reduced-round version. The functional graph of Catena-D is determined by the time parameter λ (values $\lambda = 1, 2$ 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-D-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 F 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 has 512-bit output, so the total memory requirements are 2^{n+6} bytes.

First layer is filled as follows

- $[0]^0 = G_1(P, S)$, where G_1 invokes 3 calls to F;
- $[1]^0 = G_2(P, S)$, where G_2 invokes 3 calls to F
- $[i]^0 \leftarrow F([i-1]^0, [i-2]^0),\ 2 \leq i \leq 2^n - 1.$

Then $2^{3n/4}$ nodes of the first layer are modified by function Γ. The details of Γ are irrelevant to our attack.

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

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

The layers are then computed as

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

Thus to compute $[X]^l$ we need $[\nu(X)^{l-1}]$. The latter can be then overwritten[5]. An example of Catena-D 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.

[5] In terms of Equation (1) we could enumerate all blocks as $[i]^j = j|| \underbrace{i}_{n \text{ bits}}$ so that $\phi(j||i) = (j-1)||\nu(i)$.

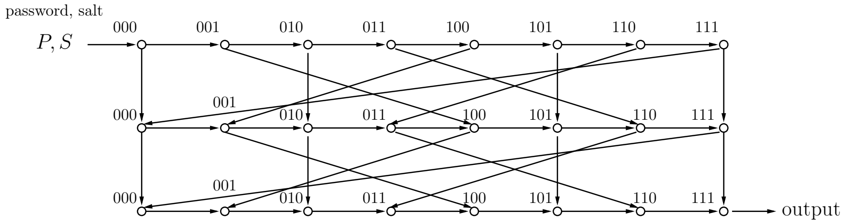

Fig. 1. Catena-D-2 with $n = 3$. 3 layers, 8 vertices per layer.

Original tradeoff analysis. The authors of Catena-D originally provided two types of security bounds against tradeoff attacks. Recall that Catena-D-λ can be computed with $\lambda 2^n$ calls to F using 2^n memory blocks. The Catena-D designers demonstrated that Catena-D-λ can be computed using λS memory blocks with time complexity[6]

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

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:

$$P_\lambda(q) \approx \left(\frac{q}{2}\right)^{\lambda}. \tag{5}$$

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

$$P_\lambda(q) \geq \Omega\left(q^{\lambda}\right). \tag{6}$$

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. (5) and thus asymptotically violating Eq. (6).

3.2 Our tradeoff attack on Catena-D

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

$$\nu(\nu(X)) = X,$$

where X can be a single index or a set of indices. We exploit it as follows. We partition layers into *segments* of length 2^k for some integer k, and store the first block of every segment (first two blocks at layer 0). As the index of such a block ends with k zeros, we denote the set of these blocks as $[*^{n-k}0^k]$. We also store all $2^{3n/4}$ blocks modified by Γ, which we denote by $[\Gamma]$.

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

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

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

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

and

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

Therefore, when we iterate the permutation ν, we are always within some 2^k segments. We suggest the computing strategy in Algorithm 1. At layer t we recompute 2^k full segments from layers 0 to $t-2$ and 2^k subsegments of length $\nu(A)$ (interpreted as a number in the binary form) at layer $t-1$. Therefore, the total cost of computing layer t is

$$C(t) = \sum_A \sum_B ((t-1)2^{2k} + \nu(A)2^k + 2^k) =$$
$$= \sum_A ((t-1)2^n + \nu(A)2^{n-k} + 2^{n-k}) = (t-1)2^{n+k} + 2^{n+k-1} + 2^n = (t-\frac{1}{2})2^{n+k} + 2^n. \tag{7}$$

[6] This result is a part of Theorem 6.3 in [20].

Algorithm 1 Tradeoff for Catena-Dragonfly.

```
1. Compute layer 0 storing $[*^{n-k}0^k]^0$ and $[*^{n-k}0^{k-1}1]^0$, i.e. the first two blocks of every segment.
2. Compute $\Gamma$ and store all the updated blocks $[\Gamma]$ in the memory.
3. Compute layer 1 segmentwise: for each segment $[AB*^k]^1$ recompute blocks $[*^k\nu(B)\,\nu(A)]^0$ using stored blocks
   from layer 0 and $[\Gamma]$. Store blocks $[*^{n-k}0^k]^1$.
4. Compute layer 2 segmentwise: for each segment $[AB*^k]^2$ recompute $2^k$ segments $[*^kB^k]^0$ using stored blocks
   from layer 0, then use them to recompute blocks $[*^k\nu(B)\,\nu(A)]^1$ using $[\Gamma]$, then compute $[AB*^k]^2$. Store blocks
   $[*^{n-k}0^k]^2$.
5. Compute layer 3 segmentwise: for each segment $[AB*^k]^3$ recompute $2^k$ segments $[*^k\nu(B)^k]^0$ using stored blocks
   from layer 0, then recompute $2^k$ segments $[*^kB^k]^1$ using stored blocks from layer 1 and $[\Gamma]$, then recompute
   blocks $[*^k\nu(B)\,\nu(A)]^2$, then compute $[AB*^k]^3$. Store blocks $[*^{n-k}0^k]^3$.
6. Compute other layers in the similar fashion.
```

The total cost of computing Catena-D-λ is

$$2^n\left(\frac{\lambda^2}{2}2^k+\lambda+1\right).$$

We store $(t+1)2^{n-k}$ blocks as segment starting points, $2^{3n/4}$ blocks $[\Gamma]$ and 2^{2k} blocks for intermediate computations. For $k=\log q+\log(\lambda+1)$ and $q<2^{n/4}$ we store about $2^n/q$ blocks, so the memory is reduced by the factor of q. This value of k yields the total computational complexity of

$$C_q = 2^n\left(\frac{q\lambda^2(\lambda+1)}{2}+\lambda+1\right) \tag{8}$$

Since the computational complexity of the memory-full algorithm is $(\lambda+1)2^n$, our tradeoff method gives the computational penalty

$$\frac{q\lambda^2}{2}+1.$$

Since Catena is a data-independent scheme, the latency of our method does not increase. Therefore, the time-memory product (Equation (4)) can be reduced by the factor of $2^{n/4}$. We can estimate how AT costs evolves assuming the reference implementation in Section 2.3:

$$\mathrm{AT}_{B_q} = 2^n\left(\frac{q\lambda^2(\lambda+1)}{2}+\lambda+1\right)\cdot 3000+(\lambda+1)2^n\frac{2^n}{q}.$$

For $q=2^{n/5}$ and $\lambda=2$ we get

$$\mathrm{AT}_{B_{2^{n/5}}} = 2^n\left(6\cdot 2^{n/5}\right)\cdot 2^{11.5}+3\cdot 2^{9n/5}.$$

For $n=24$ (1GB of RAM) we get

$$\mathrm{AT}_{B_{24.8}} \approx 2^{24+2.5+4.8+11.5}+2^{43.2+1.5}\approx 2^{44}.$$

whereas

$$\mathrm{AT}_{B_1} = 2^{49.5}.$$

Therefore, we expect the time-area product dropped by the factor of about 25 if the memory is reduced by the factor of 30. In the terms of Definition 1, Catena-D-2 is not $(1/25,\mathrm{AT})$-secure. Our tradeoff method also have AT and TM compactness at least $2^{n/5}=64$.

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

Violation of Catena-D lower bound Our method shows that the Catena-D 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:

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

which is asymptotically smaller than the lower bound $\Omega(q^\lambda)$ (Equation (6)) from the original Catena submission [20].

3.3 Other results for Catena

Our attack on Catena can be further scrutinized and generalized to non-even segments. More details are provided in [14] with the summary given in Table 2.

4 Generic precomputation tradeoff attack

Now we try to generalize the tradeoff method used in the attack on Catena for a class of data-independent schemes. We consider schemes $\mathcal{G}$ where each memory block is a function of the previous block and some earlier block:

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

where ϕ is a deterministic function such that $\phi(i) < i$. A group of existing password hashing schemes falls into this category: Catena [20], Pomelo [36], Lyra2 [25] (first phase). Multiple iterations of such a scheme are equivalent to a single iteration with larger T and an additional restriction

$$x - M \leq \phi(x),$$

so that the memory requirements are M blocks.

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 $U_k = U(I)$:

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

The scheme $\mathcal{G}$ is then computed according to Algorithm 2.

Algorithm 2 Precomputation method

```
– For all segments precompute the block indices in the union chains $I \to U(I)$
– Compute $\mathcal{G}$ by segment. For each segment $I$:
    1. Compute blocks $U(I) = U_k$;
    2. Compute blocks in $U_{k-1}$, then in $U_{k-2}$, up to $U_0 = I$.
    3. Store the first block of $I$ in the memory.
```

The total amount of calls to F is $\sum_{i \geq 0} |U_i|$, and the penalty to compute I is

$$CP(I) = \frac{\sum_{i \geq 0} |U_i|}{|I|}.$$

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

Memory fraction	Catena-D-3		Catena-D-4	
	Computational penalty			
	Our	[20]	Our	[20]
$\frac{1}{2}$	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 2. Computation-memory tradeoff for Catena-D-3 and Catena-D-4.

function or a bit permutation [20, 25]. 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 $\phi(i) \in \mathcal{L}$.

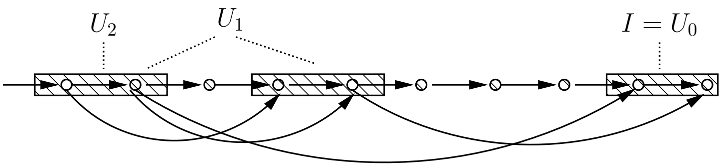

Fig. 2. Segment unions in the precomputation method.

5 Generic ranking tradeoff attack

Now we present a generic attack on a wide class of schemes with data-dependent memory addressing. Such schemes include scrypt [30] and the PHC finalists yescrypt [31], Argon2d [13], and Lyra2 [25]. We consider the schemes described by Equation (1) with $k = 2$ and the following addressing (cf. also Figure 3):

$$\begin{aligned} &X[1] = f(I) \\ &\text{for } 1 < i < T \\ &\qquad r_i = g(X[i-1]); \\ &\qquad X[i] = F(X[i-1], X[r_i]). \end{aligned} \tag{9}$$

Here g is some indexing 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.

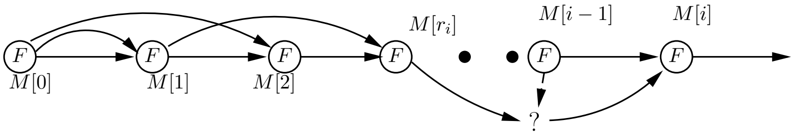

Fig. 3. Data-dependent schemes.

Our tradeoff strategy is following: we compute the blocks sequentially and for each block $X[i]$ decide if we store it or not. If we do not store it, we calculate its *access complexity* $A(i)$ – the number of calls needed to recompute it as a sum of access complexities of $X[i-1]$ and $X[r_i]$ plus one. If we store $X[i]$, its access complexity is 0.

The storing heuristic rule is the crucial element of our strategy. The idea is to store the block if $A(r_i)$ is too high.

Our *ranking tradeoff method* works according to Algorithm 3 (Figure 4).

Here w, s and l are parameters, and we usually set $l = 3s$. The computational complexity is computed as

$$C = \sum_i A(r_i).$$

We also compute the latency $L(i)$ of each block as $L(i) = \max(L(r_i), L(i-1)) + 1$ if we do not store $X[i]$ and $L(i) = 0$ if we store it. Then the total latency is

$$L = \sum_i L_i.$$

Algorithm 3 Ranking method

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

We implemented our attack and tested it on the class of functions described by Equation (9). For fixed w and s the total number of calls to F and the number of stored blocks is entirely determined by indices $\{r_i\}$. Thus we do not have to implement a real hash function, and it is sufficient to generate r_i according to some distribution, model the computation as a directed acyclic graph, and compute C and L for this graph. We made a number of tests with uniformly random r_i (within the segment $[0; i]$ and $T = 2^{12}$) and different values of w and s. Then we grouped C and L values by the memory complexity and figured the lowest complexities for each memory reduction factor. These values are given in Table 3.

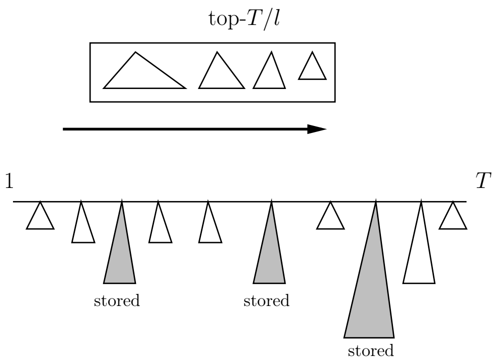

Fig. 4. Outline of the ranking tradeoff method.

Memory fraction $(1/q)$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$	$\frac{1}{9}$	$\frac{1}{10}$
Computation penalty $CP(q)$	1.59	2.98	7.3	16.6	57.5	180	635	3340	$2^{13.2}$
Latency penalty $LP(q)$	1.56	2.55	4	5.8	8.7	11.6	15.4	21.1	24.8
AT ratio	0.78	0.85	1.02	1.16	1.45	1.69	2.04	2.97	4.24
TM ratio	0.78	0.85	1.02	1.16	1.45	1.65	1.9	2.34	2.48
Segment length s	3	5	8	10	13	16	18	21	23
Window size $\frac{w}{M}$	0.06	0.01	0.01	0	0	0	0	0	0

Table 3. Computational, latency, AT (for $R_c = 3000$ and $M = 2^{24}$), and TM penalties for the ranking tradeoff attack on generic data-dependent schemes.

We conclude that generic 1-pass data-dependent schemes with random addressing are $(0.75, AT)$- and $(0.75, TM)$-secure using our ranking method. Both AT and TM ratios exceed 1 when $q \geq 4$, so both the AT- and the TM-compactness is about 4.

6 Cryptanalysis of yescrypt

6.1 Description

yescrypt [31] is another PHC finalist, which is built upon scrypt and is notable for its high memory filling rate (up to 2 GB/sec) and a number of features, which includes custom S-boxes to thwart exhaustive search on GPU, multiplicative chains to increase the ASIC latency, and some others. yescrypt is essentially a family of functions, each member activated by a combination of flags. Due to the page limits, we consider only one function of the family.

Here we consider the yescrypt setting where flag yescrypt_RW is set, there is no parallelism, and no ROM (in the further text – just yescrypt). It operates on 1024-byte memory blocks $X[1], X[2], \ldots, X[M]$. The scheme works as follows:

$$\begin{aligned} X[1] &\leftarrow F'(I);\\ X[i] &\leftarrow F(X[i-1] \oplus X[\phi(i)]),\ 1 < i \leq M;\\ Y &\leftarrow X[M];\\ Y &\leftarrow X[Y \bmod M]) \leftarrow F(Y \oplus X[Y \pmod M)]), M < i \leq T. \end{aligned}$$

Here F and F' are compression functions (the details of F' are irrelevant for the attack). Therefore, the memory is filled in the first M steps and then $(T - M)$ blocks are updated using the state variable Y. Here $\phi(i)$ is the data-dependent indexing function: it takes 32 bits of $X[i-1]$ and interprets it as a random block index among the last 2^k blocks, where 2^k is the largest power of 2 that is smaller than i.

Transformation F operates on 1024-byte blocks as follows:

- Blocks are partitioned into 16 64-byte subblocks $B_0, B_1, \ldots, B_{15}$.
- New blocks are produced sequentially:

$$\begin{aligned} B_0^{new} &\leftarrow f(B_0^{old} \oplus B_{15}^{old});\\ B_i^{new} &\leftarrow f(B_{i-1}^{new} \oplus B_i^{old}),\ 0 < i < 16. \end{aligned}$$

The details of f are irrelevant to our attack.

6.2 Tradeoff attack on yescrypt

Our crucial observation is that there is no diffusion from the last subblocks to the first ones. Thus if we store all B_0, we break the dependencies between consecutive blocks and the subblocks can be recomputed from B_1 to B_{15} with pipelining (Figure 5). Suppose that the block $X[i]$ is computed with latency $L(i)$, i.e. its computation tree has $L(i)$ levels if measured in F. However, if we consider the tree of f, then the actual latency of $X[i]$ is $L(i) + 15$ instead of expected $16L(i)$ if measured in calls to f.

The tradeoff strategy is given in Algorithm 4.

Algorithm 4 Tradeoff attack on yescrypt.

```
1. Start the ranking tradeoff method with some parameters $w, s$;
2. Store $B_0$ of each block;
3. If $X[i]$ needs the missing block $X[r_i]$:
   (a) Compute $B_0$ of $X[i]$ using one call to $f$, as all previous $B_0$ are stored;
   (b) Compute $B_1$.
   (c) While $B_1$ is recomputed, start recomputing $B_2$, as it needs exactly the same subblocks used in the recomputation of $B_1$. This adds latency of one call to $f$.
   (d) Compute $B_i$ for all other $i$.
```

If the missing block is recomputed by a tree of depth D, then the latency of the new block is $D + 16$ measured in calls to f, or $\frac{D}{16} + 1$ if measured in calls to F. This number should be compared to the latency $D + 1$ if we had not exploited the iterative structure of F. Thus if the ranking method gives the total latency L (measured in F), the actual latency should be $\frac{L+15}{16}$.

For the smallest secure parameter ($T = 4M/3$) we get the final computational and latency penalties as well as AT and TM penalties are given in Table 4 (1/16-th of each block is added to the attacker's memory). We conclude yescrypt is only $(0.45, AT)$- and $(0.45, TM)$-secure, whereas the AT compactness is 4 and the TM compactness is 6. Since this numbers are worse than for generic 1-pass schemes, our attack clearly signals of a vulnerability in the design of BlockMix. We expect that our attack becomes inefficient for $T = 2M$ and higher.

Memory	1	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$
Computation penalty $CP(q)$	1	2.9	26	1135	2^{19}	–	–	–
Latency penalty $LP(q)$	1	1.1	1.4	2	3.5	6.3	11.1	17.5
TM ratio	1	0.55	0.47	0.5	0.75	1.05	1.59	2.19
AT ratio	1	0.55	0.46	0.7	95	–	–	–

Table 4. Computational, latency, AT (for $R_c = 3000$ and $M = 2^{24}$), and TM penalties for the ranking tradeoff attack on yescrypt mode of operation with 4/3 passes, using the iterative structure of F.

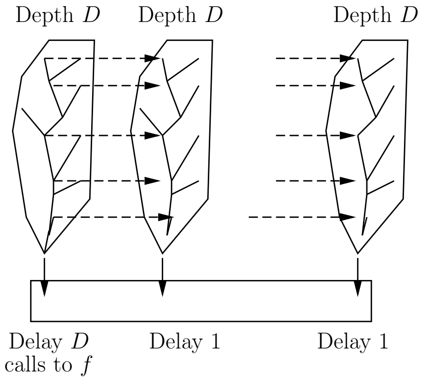

Fig. 5. Pipelining the block computation in yescrypt: only the first subblock is computed with delay D.

7 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: Pomelo [36] and the modified Lyra2.
- Set of design criteria for the indexing functions that would withstand our attacks.
- 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.

8 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-Dragonfly and yescrypt. We show that Catena-Dragonfly is not memory-hard despite original 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 yescrypt 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 the factor of 2.

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.

9 Acknowledgement

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

A Cryptanalysis of Lyra2 v1

A.1 Description of Lyra2 v1

Lyra2 [25] 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 [25], Lyra2 v1 (just Lyra2 in the further text).

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) 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 [25, 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:

```
$M[0], M[1] \leftarrow f(\text{Password,Salt})$;
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 truncates the first input to the least significant 32 bits and xores with the second input. All indices are computed modulo 2^n.

A.2 Tradeoff attack on the Setup and Wandering phases 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 of rows $\{M[i], M[i+1], \ldots, M[j]\}$ 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.

1. First 2^{n-l} rows $M[0], \ldots, M[2^{n-l}-1]$ for some $l > 0$ (parameter of the attack).
2. We split rows from $M[2^{n-l}]$ to $M[2^n - 1]$ into segments of length q for some $q < 2^{n-l}$. 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 6).

Let us figure out the memory reduction and the computational overhead of this procedure. We store 2^{n-l} first rows and $\frac{2^n}{96q}$ rows for starting state in each segment, then a segment of length q during recomputation. 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

$$(k - n + l)q \tag{10}$$

calls to F to compute a segment of length q between row indices 2^k and 2^{k+1}. For the entire Setup phase we spend

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

calls to F. The memory requirements are $2^{n-l} + q + \frac{2^n}{96q}$, which reaches the minimum of $2^{n-l} + 2^{n/2-4.5}$ for $q = 2^{n/2-5.5}$.

To summarize, our tradeoff algorithm B has computational penalty $(l - 0.5)$ if the memory is reduced by the factor of 2^l (Table 5).

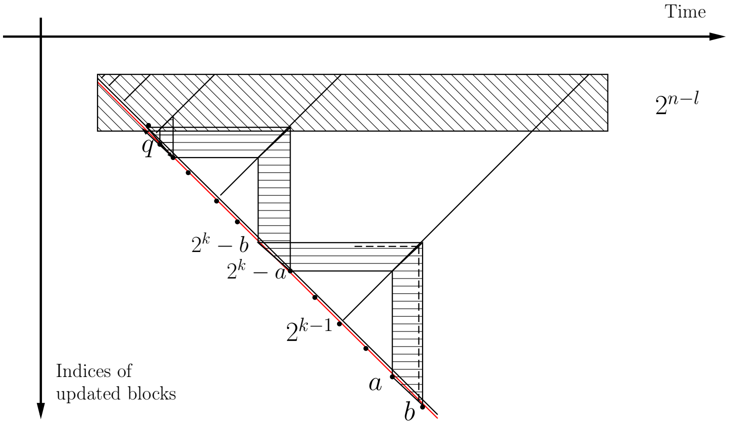

Fig. 6. Computing segment of length q with precomputation method in the Setup phase of Lyra2.

Memory fraction	Computational penalty	[25]
1/4	1.5	2
1/8	2.5	4
2^{-l}	$l - 0.5$	2^{l-1}

Table 5. 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:

$$A = q(l - 0.5). \tag{11}$$

Tradeoff attack on the Wandering phase of Lyra2 with $T = 1$.
We apply the ranking method to the Wandering phase of Lyra2. Since Lyra2 updates two rows at once, its penalties are higher than in generic data-dependent schemes and are given in Table 6.

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

A.3 Tradeoff for the full Lyra2 with $T = 1$

Memory partition. To run the attack on the full Lyra2 with fraction $1/l$ of memory, we have to split the available memory between Setup and Wandering phases. Suppose that we allocate fraction α of memory for the Setup phase and fraction β of memory for the Wandering phase. Let $P_S(\alpha)$ be the penalty of running the Setup phase with fraction α, $P_R(\alpha)$ be the average access complexity of a single row from the Setup phase run with fraction α (Eq. (11)), and $P_W(\beta)$ be the penalty of running the Wandering phase with fraction β (Table 6). 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.

Exploiting iterative compression function Similarly to the attack on yescrypt we can 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: 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 latency penalty is $1 + D/128$. Therefore, the total latency 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 latency penalty in the Setup phase, $D_R(\alpha) = -\log \alpha - 0.5$ is the average latency penalty for accessing the row from the Setup phase, and $D_W(\beta)$ is the depth penalty for the Wandering phase given in Table 6.

The results are given in Table 7. We conclude that Lyra2 is only $(0.33, AT)$-secure and $(0.1, TM)$-secure. The AT compactness is 4 and the TM compactness is 16.Thus Lyra2 v1 is more susceptible to tradeoff attacks compared to yescrypt.

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/`.

Memory Wandering+Setup	1	0.45 $1/3 + 1/8$	0.31 $1/4 + 1/16$	0.26 $1/5 + 1/16$	0.12 $1/10 + 1/64$	0.06 $1/17 + 1/256$
Comp.penalty $CP(q)$	1	14.2	133	1876	2^{19}	–
Lat. penalty $LP(q)$	1	1.03	1.05	1.08	1.37	1.93
TM penalty	1	0.47	0.33	0.28	0.16	0.12
AT penalty	1	0.47	0.35	0.63	94	–

Table 7. Computational, latency, AT (assuming $R_c = 3000$ and $M = 2^{24}$) and TM penalties for the ranking tradeoff attack on Lyra2 v1 with $T = 1$. Memory fraction is given as a sum of Wandering and Setup allocations

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, Daniel Dinu, and Dmitry Khovratovich. Argon and argon2: password hashing scheme. Technical report, 2015. `https://password-hashing.net/submissions/specs/Argon-v2.pdf`.
14. Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of memory-hard functions. Cryptology ePrint Archive, Report 2015/227, 2015. `http://eprint.iacr.org/`.
15. 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.
16. 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`.
17. 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.
18. 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.
19. Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. *IACR Cryptology ePrint Archive 2013/796, to appear at Crypto 2015.*
20. Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. *IACR Cryptology ePrint Archive, Report 2013/525. Version of 5 Jan 2014, `https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2013/525&version=20140105:194859&file=525.pdf`*, 2014.
21. 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.
22. 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.
23. 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.
24. John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. On time versus space. *J. ACM*, 24(2):332–337, 1977.
25. 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.
26. 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.
27. Katja Malvoni. Energy-efficient bcrypt cracking. Passwords'14 conference, available at `http://www.openwall.com/presentations/Passwords14-Energy-Efficient-Cracking/`.

28. 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.
29. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2009. http://www.bitcoin.org/bitcoin.pdf.
30. Colin Percival. Stronger key derivation via sequential memory-hard functions. 2009. http://www.tarsnap.com/scrypt/scrypt.pdf.
31. Alexander Peslyak. Yescrypt - a password hashing competition submission. Technical report, 2014. available at https://password-hashing.net/submissions/specs/yescrypt-v0.pdf.
32. Nicholas Pippenger. Superconcentrators. *SIAM J. Comput.*, 6(2):298–304, 1977.
33. 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.
34. Martijn Sprengers and Lejla Batina. Speeding up GPU-based password cracking. In *SHARCS'12*, 2012. available at http://2012.sharcs.org/record.pdf.
35. Clark D. Thompson. Area-time complexity for VLSI. In *STOC'79*, pages 81–88. ACM, 1979.
36. Hongjun Wu. POMELO: A password hashing algorithm. Technical report, 2014. available at https://password-hashing.net/submissions/specs/POMELO-v1.pdf.