Fast and Tradeoff-Resilient Memory-Hard Functions for Cryptocurrencies and Password Hashing

Alex Biryukov
University of Luxembourg
alex.biryukov@uni.lu

Daniel Dinu
University of Luxembourg
dumitru-daniel.dinu@uni.lu

Dmitry Khovratovich
University of Luxembourg
dmitry.khovratovich@uni.lu

Abstract

Memory-hard functions are becoming an important tool in the design of password hashing schemes, cryptocurrencies, and more generic proof-of-work primitives that are x86-oriented and can not be computed on dedicated hardware more efficiently.

We develop a simple and cryptographically secure approach to the design of such functions and show how to exploit the architecture of modern CPUs and memory chips to make faster and more secure schemes compared to existing alternatives such as scrypt. We also propose cryptographic criteria for the components, that prevent cost reductions using time-memory tradeoffs and side-channel leaks. The concrete proof-of-work instantiation, which we call Argon2, can fill GBytes of RAM within a second, is resilient to various tradeoffs, and is suitable for a wide range of applications, which aim to bind a computation to a certain architecture. Concerning potential DoS attacks, our scheme is lightweight enough to offset the bottleneck from the CPU to the memory bus thus leaving sufficient computing power for other tasks. We also propose parameters for which our scheme is botnet resistant. As an application, we suggest a cryptocurrency design with fast and memory-hard proof-of-work, which allows memoryless verification.

1 Introduction

Adversaries, equipped with custom-based devices are now able to compute proofs-of-work of various kinds by a great factor more efficient than regular desktops or laptops. A prominent example is the Bitcoin hardware arms race, where the best x86-machine spends 30,000 times more energy to find a Bitcoin block than the best ASIC mining rigs [6]. As a result, the originally egalitarian concept of a decentralized cryptocurrency fell apart with almost 50% of computing power concentrated in a few mining factories and periodically held by a single mining pool. Moreover, the centralized mining puts security at risk, as many attacks become possible when a single entity controls more than about 50% of the network power. Similar situation takes place in the password hashing and password-based key derivation, where database leaks are frequent [2] and GPU and FPGA crackers [24, 32] are popular tools. Thus the need for GPU/FPGA/ASIC-resistant schemes becomes evident.

Memory-intensive computations were proposed in 2003 as a remedy for the increasing computational power of GPU and specialized hardware [7, 14]. The motivation behind large memory requirements is twofold. First, the memory latency is similar across different platforms, as long as the same technology is used. Secondly, the memory chips are rather large and thus are relatively expensive in production. For example, an energy-efficient DRAM chip [20] occupies 550 mm^2 per GB, which is equivalent to 10,000 Bitcoin hash cores.

It is evident that a memory-intensive computation is a countermeasure as long as the memory can not be traded for time or extra computations with the reduction in the total attack cost. Such tradeoffs were explored in the 1970s [21, 23]. For some functions it was proven that even small memory reduction causes very high, exponential in terms of memory fraction increase in time complexity [30]. All such designs are based on the iterative application of some internal hash function F to memory blocks.

Even though such constructions have been adapted in several proofs-of-work [14,15,19], they are not practical enough. The problem is that they are based on the superconcentrators [30] and are usually very slow. For instance, schemes from [30] have to hash the entire memory dozens of times, which is equivalent to about 5-10 MBytes/sec. The time constraints in web authentication and block/transaction verification in cryptocurrencies would reduce the memory to a few MBytes, which is a weak countermeasure.

A much faster alternative was suggested by Percival as scrypt in 2009 [28], which fills the memory at 2-5 cycles per byte depending on the CPU and underlying parameters (which is up to 700 MB/sec on modern CPU). However, scrypt has its own problems too. First, it combines several unrelated cryptographic primitives, tunable under many parameters, and thus is difficult to analyze. Secondly, it admits a simple time-memory tradeoff, where memory reduction by q increases the time less than by q (and linear in q). Thirdly, scrypt accesses memory in the data-dependent pattern, which makes it vulnerable to side-channel timing attacks [10,31], which

are quite relevant in password cracking. Though scrypt became popular enough to become an IETF draft [5], its adaptation in the Litecoin cryptocurrency [4] is not ASIC-resistant, as the ASIC mining rigs still are hundreds of times more efficient [3].

Percival called scrypt *memory-hard* for its tradeoff, which he proved asymptotically under certain assumptions on the underlying compression function. There were attempts to construct *memory-hard* functions that access memory in the data-independent way [13, 18, 22], thus protecting from potential timing attacks. However, the time-memory tradeoffs for the data-independent functions are much more dangerous, as an ASIC-equipped adversary can create a custom chip with multiple cores, and precompute all the memory blocks he does not store – by the time they are needed. It was demonstrated that the memory access patterns employed in [18, 22] are weak [12] and do not allow for tradeoff resistance stronger than in scrypt.

The function scrypt and later designs [22, 29] incorporated a number of tunable parameters, such as parallelism, block size, the internal hash function, the number of passes over the memory and so on. No solid theory of the design of memory-hard functions has yet appeared, so the parameters are chosen and implemented ad-hoc. Quite many parameters are left to the users with an unknown effect on security.

Also, the advantages of running the scheme on other platforms, such as GPUs and ASIC, have not been properly formalized. If we consider ASIC-based Bitcoin mining, the main efficiency parameter is the energy spent per computed hash. However, the total energy consumption is difficult to estimate for memory-equipped designs, as various memory types are quite different in static and active power consumption. Following [11, 25, 33], the authors of [12] suggest approximating the total cost with the time-area product. However, this parameter depends on the architecture as well, as we have to know the memory latency and the ratio between the memory area and the computational core area. Still, one could estimate the computational penalties due to the memory reduction (*computation-memory tradeoff*) and time penalties due to the memory reduction (*time-memory tradeoff*). For both estimates we assume unlimited parallelism (see [8] for a more rigorous treatment of the parallel complexity). For each particular platform we can figure out the area requirements and parallelism restrictions, obtain the time-area product and estimate the attack costs.

Our contributions. We propose a rigorous approach to the design of memory-hard functions and demonstrate how to construct a scheme that maximizes the adversaries' work on other platforms such as GPU and ASIC in the cryptocurrency and password hashing settings. We demonstrate how to parallelize the computation in a tradeoff-resilient way, and prove the absence of internal collisions for a permutation-based compression function. We thoroughly discuss the need for multiple passes over the memory and figure out an optimal number of passes for both data-dependent and data-independent instances of our proposal Argon2. Finally, we offer a concrete proof-of-work function called Argon2 based on the data-dependent version Argon2d, which is offered for ASIC- and botnet-resistant cryptocurrencies. It requires 2GB of memory for generation but only 1390 KB for verification, and can be computed in 2 seconds on a 2 GHz PC, while its verification takes milliseconds.

2 Definitions and concepts

We consider functions H that can be represented as directed acyclic graphs (DAGs), where each node has in-degree at maximum k. With each node we associate a t-bit value M and a call to some function F which takes the values of k other nodes as inputs:

$$M_{k+1} \leftarrow F(M_1, M_2, \ldots, M_k). \tag{1}$$

Let the graph have T nodes, then we say that it can be computed with computational complexity T and memory complexity T, where the computational unit is the call to F and the memory unit is the output length of F. If the computation is serial, then the time complexity equals the computational complexity. However, if some additional computing units are available, the total time might be reduced.

We are interested to know how the the minimal computational and time complexities of computing H change if only T/q memory can be used. We denote the penalty coefficient on the computational complexity by $C(q)$, and on the time complexity by $T(q)$ so that the total computation needs $C(q)T$ calls to F and $T(q)T$ time. Any concrete algorithm using T/q memory provides an upper bound on $C(q)$ and $T(q)$, whereas lower bounds are difficult to obtain. In practice, lower bounds on $C(q)$ and $T(q)$ are a conjecture and its validity comes from public scrutiny. Following [28], we call a function H *memory-hard* if both $C(q)$ and $T(q)$ grow at least linearly with q assuming unlimited parallelism. Since we do not specify the adversaries, our definition is not rigorous, but is apparently sufficient for the design and analysis purposes.

The vast majority of schemes that claim memory-hardness [13, 19, 22, 28, 29] conform to Equation 1 with

minor corrections. They fill the memory with T blocks $M_1, M_2, \ldots, M_T$ as follows

$$\begin{aligned} M_1 &\leftarrow H(\text{Input}), \\ M_j &= F(M_{\phi_1(j)}, M_{\phi_2(j)}, \ldots, M_{\phi_k(j)}), 1 < j \le T, \\ \mathit{Output} &\leftarrow H(M_T), \end{aligned} \tag{2}$$

where ϕ_i are some indexing functions and H is a cryptographic (memoryless) hash function. There could also be several passes over memory, and different F can be used.

The indexing functions crucially affect the security. We distinguish two classes of them. If ϕ does not depend on the input, we call the scheme *data-independent*. The memory addresses can be calculated by the adversaries. Therefore, if the dedicated hardware can handle parallel memory access, the adversary can prefetch the data from the memory. Moreover, if he implements a time-space tradeoff, then the missing blocks can be also precomputed without losing time. In order to maximize the computational penalties, the designers proposed various formulas for indexing functions [18, 22], but several of them were found weak and admitting easy tradeoffs [12].

In the second class of schemes ϕ depends on the previously computed blocks. We call such schemes *data-dependent*. It is popular just to use some bits of the previous block: $\phi(j) = \text{Truncate}(M_{j-1})$. This prevents the adversary from prefetching and precomputing missing data. The adversary figures out what he has to recompute only at the time the element is needed. If we reduce the memory by q so that a missing block is recomputed as a tree of calls to F of average depth D_q, then

$$T(q) = D_q + 1.$$

However, these schemes are vulnerable to side-channel attacks, as timing information may help to filter out password guesses at an early stage.

3 Mode of operation

Inputs to F. Let us further define the generic scheme (2). For the moment, our scheme should not allow parallel computation, so that the total computation time even in the presence of multiple cores should be T. Therefore, there must be a computation chain of length T, which is equivalent to setting

$$\phi_k(j) = j - 1.$$

Then we discuss the value of k. As k increases, we have to load more blocks from the memory so the scheme becomes slower as F has to process more blocks. Our experiments show that in the simplest case when F simply xors all inputs before passing them to the t-bit bijective function (permutation), the performance decreases by 30% when going from $k = 2$ to $k = 3$. The benefit would be that $C(q)$ and $D(q)$ also grow with k. However, we note that on the custom hardware the adversary would not experience any time penalty if he uses the full memory. Indeed, the adversary would just modify the memory bus to be able to read twice more and thus reduce the time-area product. We conclude that $k = 2$ is optimal for full-memory adversaries, so we restrict the further text to the following class of schemes:

$$\begin{aligned} M_1 &\leftarrow H(\text{Input}), \\ M_j &= F(M_{\phi(j)}, M_{j-1}), 1 < j \le T, \\ \mathit{Output} &\leftarrow H(M_T). \end{aligned} \tag{3}$$

Indexing functions. For the data-dependent addressing we set $\phi(j) = \text{Truncate}(M_{j-1})$, where the result is taken modulo j. We considered taking the address not from the block M_{j-1} but from the block M_{j-2}, which should have allowed a prefetch of the block earlier. However, not only is the gain in our implementations limited, but also this benefit can be exploited by the adversary. Indeed, the recomputation depth $D(q)$ is now reduced to $D(q) - 1$, since the adversary has one extra timeslot, and thus the total time penalty decreases by 1. Later we'll see (Table 1) that this is a major change, as the adversary would be able to reduce the memory by the factor of 5 without increasing the time-memory product.

For one instance of our scheme we use data-independent addressing. From recent research [12] we know that some variants of addressing are vulnerable to tradeoff attacks. In short, the generic tradeoff attack from [12] works as follows:

- The memory is partitioned into segments of length q; only the first block of every segment is stored.
- For each segment I_0 that should be computed we compute the blocks $\phi(I_0)$ referenced by the function ϕ from this segment. Let I_1 be the union of segments that contain $\phi(I_0)$.

- We apply ϕ to I_1 in the same way until we reach the set I_k of segments such that

$$\phi(I_k) \subset I_k$$

- Then to compute I_0 we first recompute I_k, then I_{k-1} and so on up to I_0.

This method works well if k is small and I_k is not too large compared to I_0. We conjecture that the method does not work well if ϕ works like a random function, so that its images are uniformly distributed. As our scheme should be deterministic, we make ϕ pseudo-random. A simplest choice would be $\phi(i) = H(i) \pmod i$ for some cryptographic hash function H. However, this is overkill for a single index. Instead we propose to iterate some reduced-round hash function G in the counter mode and split its output into multiple indices. For example, we could take $G(i) = F(F(i))$, where F is the block generation function, which produces hundreds of addresses at once.This approach does not give provable tradeoff bounds, but instead allows the analysis with the tradeoff algorithms suited for data-dependent addressing.

4 Compression function F

In contrast to attacks on regular hash functions, the adversary does not control inputs to the compression function F in our scheme. Intuitively, this should relax the cryptographic properties required from the compression function and allow for a faster primitive.

4.1 Block size

When we request a block from a random location in the memory, we most likely get a cache miss. The first bytes would arrive at the CPU from modern RAM (DDR3 and better) within at best 10 ns, which accounts for 20-30 cycles. In practice, however, a single load instruction may take 100 cycles and more. To amortize this number, we exploit the fact that not only the requested bytes but also the following bytes are loaded into cache, automatically or using the `prefetch` instruction. The data from L1 cache theoretically may be loaded at 64 bytes per cycle on the Haswell architecture. Even though we have not reached this performance in our experiments, the speed-up for large blocks is significant. The exact numbers depend highly on the compression function. Our tests for a weak compression function (that just XORs the inputs) show that the performance reaches 0.8 cycles per byte for the 1024-bit block and then improves only a bit with 8192-bit block giving around 0.7 cycles per byte. Taking into account that larger blocks require slower compression function (more rounds), we decided to work with 8192-bit blocks (1 KB), though 4096-bit and 16384-bit blocks should be about that fast.

4.2 Insecurity of an iterative compression function

Now we have to design a compression function

$$Z = F(X, Y),$$

where X, Y, Z and Y are 8192-bit blocks. It appears that collision/preimage resistance and their weak variants are an overkill as a design criteria for the compression function F. Our main requirement is the tradeoff-resilience: there should be no way to produce the output block using less than 8192 bits of memory.

A naive approach would be an iterative compression function like the following:

- The input blocks of size t are divided into s shorter subblocks (columns) $X_1, X_2, \ldots, X_s$ and $Y_1, Y_2, \ldots, Y_s$.
- The output block Z is computed columnwise:

$$\begin{aligned} Z_1 &= G(X_1, Y_1); \\ Z_i &= G(X_i, Y_i, Z_{i-1}),\ i > 0. \end{aligned}$$

A similar compression function is used in the hashing scheme Lyra [22] and was found vulnerable to tradeoff attacks in [12]. The idea is to recompute the block columnwise as follows. Suppose that to recompute Z we need to compute a tree of depth D. We compute only Z_1, which is also a tree of depth D, but with smaller function G. The recomputation of Z_2 may start just after the first columns are computed in the recomputation of Z_2 so that the next columns are produced with latency of 1 call to G. In total, the recomputation takes $(D + s)$ calls to G instead of D calls to F. As each call to F is s calls to G, we obtain that

$$T(q) = \frac{D + s}{s} = 1 + D/s.$$

Therefore, the time penalty is efficiently divided by s, which reduces the time-memory product significantly.

scrypt partially handles this problem by interleaving columns from the first and the second half of Z, though we believe that a slightly different attack would still apply. To completely solve the problem, we have proposed the design criteria for such compression function.

4.3 Design criteria

We suggest the following principles:

- *The compression function must require about t bits of storage (excluding inputs) to compute any output bit.*
- *Each output byte of F must be a nonlinear function of all input bytes, so that the function has differential probability below a certain value, for example $\frac{1}{4}$.*

These criteria ensure that the attacker is unable to compute an output bit using only a few input bits or a few stored bits. Moreover, the output bits should not be (almost) linear functions of input bits, as otherwise the function tree would collapse.

We have not found any generic design strategy for such large-block compression functions. It is difficult to maintain diffusion on large memory blocks due to the lack of CPU instructions that interleave many registers at once. A naive approach would be to apply a linear transformation with guaranteed diffusion. However, even if we operate on 16-byte registers, a 1024-byte block would consist of 64 elements. A 64×64-matrix would require 32 XORs per register to implement, which gives a penalty about 2 cycles per byte.

Instead, we propose to build the compression function on the top of some smaller transformation P. We apply P in parallel (having a P-box), then shuffle the output registers (similarly to the ShiftRows transformation of AES [27]) and apply it again. If P handles p registers, then the compression function may transform a block of p^2 registers with 2 rounds of P-boxes. We do not have to manually shuffle the data, we just change the inputs to P-boxes. As an example, an implementation of the Blake2b [9] permutation processes 8 128-bit registers, so with 2 rounds of Blake2b we can design a compression function that mixes the 8192-bit block. We stress that this approach is not possible with dedicated AES instructions. Even though they are very fast, they apply only to the 128-bit block, and we still have to diffuse its content across other blocks.

As a result, we get an invertible transformation $Q()$ that applies to 8192-bit blocks. To get a compression function, we apply a simple feedback:

$$F(X, Y) = Q(X \oplus Y) \oplus X \oplus Y.$$

The resulting function is not collision-resistant, but apparently we do not need this property.

5 Parallelism

As modern CPUs have several cores possibly available for hashing, it is tempting to use these cores to increase the bandwidth, the amount of filled memory, and the CPU load.

The simplest way to use p parallel cores (used by scrypt) is to compute and hash together p independent calls to a memory-hard function MF:

$$H'(X) = H(MF(X, 0)||MF(X, 1)|| \cdots ||MF(X, p)),$$

where H is a cryptographic hash function. If a single call uses m memory units, then p calls use pm units. However, this method admits a trivial tradeoff: an adversary just makes p sequential calls to H using only m memory in total, which keeps the time-area product constant.

We suggest the following solution for p cores. Consider the class of schemes given by Equation (3). We view the memory as p lanes of T blocks each and operate as follows:

$$\begin{aligned} M_{i,1} &\leftarrow H(\text{Input}||i||1),\ 1 \le i \le p \\ M_{i,2} &\leftarrow H(\text{Input}||i||2),\ 1 \le i \le p \\ M_{i,j} &= F(M_{\phi(i,j)}, M_{i,j-1}),\ 1 \le i \le p, 1 < j \le T, \\ Output &\leftarrow H(M_{1,T}||M_{2,T}|| \dots ||M_{p,T}). \end{aligned} \tag{4}$$

To ensure the dependency between lanes, we allow the indexing function $\phi(i, j)$ to reference blocks from other lanes. We split each lane into l segments $I_1, I_2, \dots, I_l$, and use the following rules for $\phi(i, j)$:

1. $\phi(i, j)$ does not refer to $M_{i,j-1}$.

2. In I_1 of each lane $\phi(i,j)$ refers only to the same segment.

3. In I_k, $k > 1$, $\phi(i,j)$ may refer to I_k of the same lane and to I_j of all lanes for all $j < k$.

This idea is easily implemented in software with p threads and l synchronisation points. It is easy to see that the adversary can use less memory when computing I_l, for instance by computing I_l sequentially from lane 1 to lane p. Then his time is multiplied by $(1 + \frac{p-1}{l})$, whereas the memory use is multiplied by $(1 - \frac{p-1}{pl})$, so the time-memory product is modified as

$$TM_{new} = TM \left(1 - \frac{p-1}{pl}\right) \left(1 + \frac{p-1}{l}\right).$$

For $2 \leq p, l \leq 10$ this value is always between 1.05 and 3. We have selected $l = 4$ as this value gives low synchronisation overhead while imposing time-area penalties on the adversary who reduces the memory even by the factor 3/4. We note that values $l = 8$ or $l = 16$ could be chosen.

If the compression function is collision-resistant, then one may easily prove that block collisions are highly unlikely. However, we employ a permutation-based compression function, which has simple invariants. We avoid block collisions by introducing additional rule:

4. First block of I_k can not refer to the last block of I_{k-1} in any lane.

For the following theorem we need some consecutive enumeration of blocks. We use the following: first we enumerate blocks of I_1 of lane 1, then of I_1 of lane 2, and so on, then I_2 of all lanes, etc.

Theorem 1. *Let Π be a hash function, which operates according to Eq. (4) and uses the indexing function under the rules 1-4 (above). Let $F(X,Y) = P(X \oplus Y) \oplus X \oplus Y$ be its internal compression function such that*

- *$P(Z) \oplus Z$ is collision-resistant, i.e. it is hard to find a, b such that $P(a) \oplus a = P(b) \oplus b$.*
- *$P(Z) \oplus Z$ is 4-generalized-birthday-resistant, i.e. it is hard to find distinct a, b, c, d such that $P(a) \oplus P(b) \oplus P(c) \oplus P(d) = a \oplus b \oplus c \oplus d$.*

Then all the blocks M_i are different.

Proof. As H is assumed collision-resistant, the first two blocks of each lane of I_1 are all distinct. Consider the other blocks.

Suppose the proposition is wrong, and let (M_a, M_b) be a block collision such that $x < y$ and y is the smallest among all such collisions. As $P(Z) \oplus Z$ is collision resistant, the collision occurs in Z, i.e.

$$Z_x = Z_y.$$

Let $r_x = \phi(x), r_y = \phi(y)$, and let p_x, p_y be previous block (second input to F) indices for M_x, M_y. Then we get

$$M_{r_x} \oplus M_{p_x} = M_{r_y} \oplus M_{p_y}.$$

As we assume 4-generalized-birthday-resistance, some arguments are equal. Consider three cases:

- $r_x = p_x$. This is forbidden by rule 1.
- $r_x = r_y$. We get $M_{p_x} = M_{p_y}$. As $p_x, p_y < y$, and y is the smallest yielding such a collision, we get $p_x = p_y$. However, by construction $p_x \neq p_y$ for $x \neq y$.
- $r_x = p_y$. Then we get $M_{r_y} = M_{p_x}$. As $r_y < y$ and $p_x < x < y$, we obtain $r_y = p_x$. Since $p_y = r_x < x < y$, we get that x and y are in the same segment, we have two options:
  - p_y is the last block of a segment. Then y is the first block of the next segment in this lane. Since r_x is the last block of a segment, and $x < y$, x must be in the same slice as y, and x can not be the first block in a segment by the rule 4. Therefore, $r_y = p_x = x - 1$. However, this is impossible, as r_y can not belong to the same segment as y.
  - p_y is not the last block of a segment. Then $r_x = p_y = y - 1$, which implies that $r_x \geq x$. The latter is forbidden.

Thus we get a contradiction in all cases. This ends the proof. □

We note that the proof easily generalizes to multi-pass schemes, where the same rules apply to the second and later passes (except for the first segment rules, which do not apply).

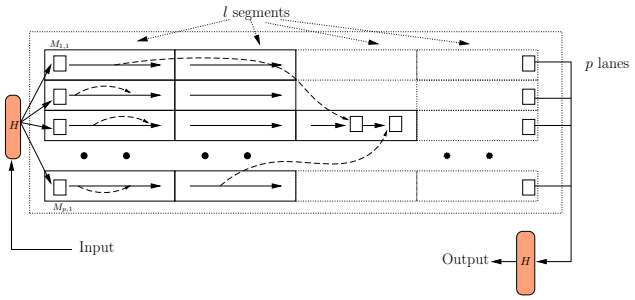

Figure 1: Our idea for tradeoff-resistant parallelism: p lanes and l synchronisation points.

6 Optimizing tradeoff resilience

6.1 Tradeoff basics

In the case of *data-dependent* schemes the adversary can reduce the time-memory product if the time penalty due to the recomputation is smaller than the memory reduction factor. The time penalty is determined by the depth D of the recomputation tree, so the adversary wins as long as

$$D(q) + 1 \leq q.$$

In contrast, the time penalty grows much slower the *data-independent* schemes in our model, as the missing blocks can always be precomputed. Regarding the time-area product, the total area decreases until the area needed to host multiple cores for recomputation matches the memory area. Suppose that the logic for G takes A_{core} of area (measured, say, in mm^2), and the memory amount that we consider (say, 1 GB), takes A_{memory} of area. The adversary reduces the total area as long as:

$$C(q)A_{core} + A_{memory}/q \leq A_{memory}.$$

The maximum memory bandwidth Bw_{max} may be an additional constraint on the scheme performance in tradeoffs. We discuss this in more details in Appendix A.

6.2 Tradeoffs for multi-pass schemes

So far the only generic tradeoff attacks on schemes in Equation (3) were reported in [12]. Using the so called *ranking method* (for the sake of completeness we describe it in Appendix C), the authors obtained the results on $C(q)$ and $D(q)$, which are given in Table 1.

We made some modifications to the algorithm given in [12] to cover multi-pass schemes, since we are interested in whether these schemes offer better tradeoff-resilience under the same time constraints.

Memory	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{8}$	$\frac{1}{10}$
C(q)	1.71	2.95	6.3	16.6	55	877	$2^{14.2}$
T(q)	2.7	3.5	4.8	6.7	9.2	16.7	27.6

Table 1: Computation penalties ($C(q)$) and time penalties ($T(q) = D(q)+1$) for the ranking tradeoff attack [12].

Suppose we make k passes with T blocks each following the scheme (3), so that after the first pass any address in the memory may be used. Then this is equivalent to running a single pass with kT iterations such that $\phi(j) \geq j - T$. The time-space tradeoff would be the same as in a single pass with T iterations and additional condition

$$\phi(j) \geq j - \frac{T}{k}.$$

We have modified the function ϕ in the ranking algorithm (Appendix C) and obtained the results in Tables 2,3. We conclude that for the data-dependent schemes several passes do increase the time-area product for the adversary who uses tradeoffs. Indeed, suppose we run a scheme with memory A with one pass for time T, or on $A/2$ with 2 passes. If the adversary reduces the memory to $A/6$ GB (i.e. by the factor of 6) for the first case, the time grows by the factor of 9.2, so that the time-area product is $1.55AT$. However, if in the second setting the memory is reduced to $A/6$ GB (i.e. by the factor of 3), the time grows by the factor of 15.3, so that the time-area product is $2.3AT$. For other reduction factors the ratio between the two products remains around 2.

Memory fraction	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$
1 pass	1.7	3	6.3	16.6	55
2 passes	15	410	19300	2^{20}	2^{25}
3 passes	3423	2^{22}	2^{32}		

Table 2: Computation/read penalties ($C(q)$ for the ranking tradeoff attack.

Memory fraction	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$
1 pass	2.7	3.5	4.8	6.7	9.2
2 passes	6.7	13.3	27.8	48	74
3 passes	21.7	57	104	–	–

Table 3: Time penalties ($T(q) = D(q) + 1$) for the ranking tradeoff attack.

7 Our proposal Argon2

Now we summarize our developments in a new design Argon2.

7.1 Specification

We suggest two schemes: Argon2d with data-dependent addressing and Argon2i with data-independent addressing. They are based on Eq.(4) with the following corrections:

- The block size is 8192 bits (see Section 4.1);
- The compression function is based on the Blake2b permutation (see Section 4.3);
- Argon2d makes one pass over memory;
- Argon2i makes three passes over memory;
- Both variants use 2, 4, or 8 lanes depending on the available number of CPU cores (1, 2, and 4, respectively) and 4 synchronisation points following the indexing rules in Section 5.
- Indices for Argon2i are produced by running $F(F())$ in the counter mode, so that each call to double-F produces 256 32-bit indices.

Argon2d is optimized for settings where the adversary does not get regular access to system memory or CPU, i.e. he can not run side-channel attacks based on the timing information, nor can he recover the password much faster using garbage collection [17]. These settings are more typical for backend servers and cryptocurrency minings. For practice we suggest the following settings:

- Cryptocurrency mining, that takes 0.1 seconds on a 2 Ghz CPU using 1 core — Argon2d with 2 lanes and 250 MB of RAM;
- Backend server authentication, that takes 0.5 seconds on a 2 GHz CPU using 4 cores — Argon2d with 8 lanes and 4 GB of RAM.

Argon2i is optimized for more dangerous settings, where the adversary possibly can access the same machine, use its CPU or mount cold-boot attacks. We use three passes to get entirely rid of the password in the memory. We suggest the following settings:

- Key derivation for hard-drive encryption, that takes 3 seconds on a 2 GHz CPU using 2 cores — Argon2i with 4 lanes and 6 GB of RAM;
- Frontend server authentication, that takes 0.5 seconds on a 2 GHz CPU using 2 cores — Argon2i with 4 lanes and 1 GB of RAM.

7.2 Compression function

Compression function F is built upon the Blake2b round function P (fully defined in [9]). P operates on the 128-byte input, which can be viewed as 8 16-byte registers (see details below):

$$P(A_0, A_1, \ldots, A_7) = (B_0, B_1, \ldots, B_7).$$

Compression function $F(X, Y)$ operates on two 1024-byte blocks X and Y. It first computes $R = X \oplus Y$. Then R is viewed as a 8×8-matrix of 16-byte registers $R_0, R_1, \ldots, R_{63}$. Then P is first applied rowwise, and then columnwise to get Z:

$$\begin{gathered}
(Q_0, Q_1, \ldots, Q_7) \leftarrow P(R_0, R_1, \ldots, R_7); \\
(Q_8, Q_9, \ldots, Q_{15}) \leftarrow P(R_8, R_9, \ldots, R_{15}); \\
\ldots \\
(Q_{56}, Q_{57}, \ldots, Q_{63}) \leftarrow P(R_{56}, R_{57}, \ldots, R_{63}); \\
\\
(Z_0, Z_8, Z_{16}, \ldots, Z_{56}) \leftarrow P(Q_0, Q_8, Q_{16}, \ldots, Q_{56}); \\
(Z_1, Z_9, Z_{17}, \ldots, Z_{57}) \leftarrow P(Q_1, Q_9, Q_{17}, \ldots, Q_{57}); \\
\ldots \\
(Z_7, Z_{15}, Z_{23}, \ldots, Z_{63}) \leftarrow P(Q_7, Q_{15}, Q_{23}, \ldots, Q_{63}).
\end{gathered}$$

Finally, F outputs $Z \oplus R$:

$$F: \quad (X, Y) \;\rightarrow\; R = X \oplus Y \;\xrightarrow{P}\; Q \;\xrightarrow{P}\; Z \;\rightarrow\; Z \oplus R.$$

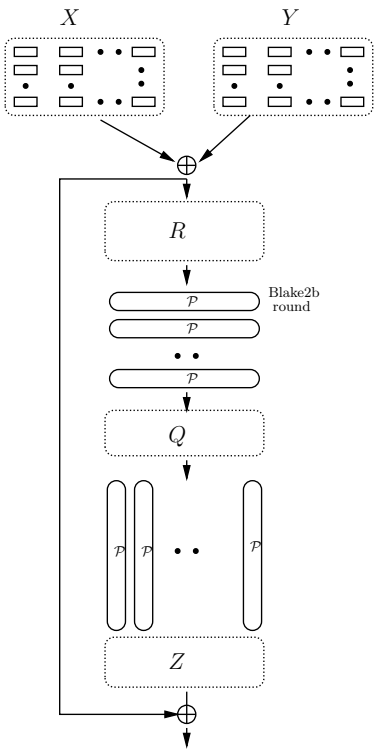

Figure 2: Compression function F in Argon2.

7.3 Argon2 vs scrypt

Here we summarize the advantages of Argon2 over scrypt:

- **Simplicity.** Argon2 uses only a single external primitive – Blake2, whereas scrypt needs PBKDF, Salsa20/8, and SHA-256 [28] (with some other alternatives often suggested). Argon2 does not involve a stack of subprocedures; on the contrary, it uses only the Blake2-based permutation as a subroutine.

Thr.	Argon2d (1 pass)		Argon2i (3 passes)	
	Perf. cpb	Bandw. (GB/s)	Perf. cpb	Bandw. (GB/s)
1	1.23	2.94	3.6	3
2	0.75	4.8	2.1	5.15
4	0.68	5.3	1.8	6
8	0.61	5.9	1.7	6.35

Table 4: Performance (in cycles per memory byte filled) and memory bandwidth of Argon2 with 1 GB of RAM on Core i7-4500U (4x1.8 GHz). Bandwidth is computed assuming that single pass load/stores 2GB.

The only sophisticated part of Argon2 is the block indexing rule for multi-threaded use; it can be further simplified if not a permutation but a cryptographic compression function is used. Most of the parameters of Argon2 are fixed, so that it can be used out-of-the-box with little tuning; only the level of parallelism and the memory size have to be specified.

- **Performance.** One-pass Argon2 is 50% faster than scrypt. Two-pass, two-thread Argon2 is as fast as scrypt.
- **Parallelism.** Argon2 parallelizes its computation in the tradeoff-resilient way, thus increasing the time-memory and the time-area products if the memory is reduced, whereas the parallelized scrypt lets the parallel computation be serialized and keep the time-memory product constant.
- **Tradeoff resilience.** Argon2d offers better tradeoff resilience compared to scrypt while also being faster. We compare the time-memory product change for two instances that run equally fast on our machine (using Eq. (6) and Table 3):

Memory	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$
2-pass 2-lane Argon2d	3.3	4.4	6.9	9.6	12.3
scrypt	0.62	0.5	0.44	0.4	0.38

The two-threaded scrypt runs faster, but its tradeoff resilience is much worse as the computation may be serialized easily (Section 5).

8 Cryptocurrency proof-of-work based on Argon2

8.1 Overview

As a concrete application, we suggest a cryptocurrency proof-of-work based on Argon2. We aim to make this PoW unattractive for botnets, so we suggest using 2 GB of RAM, which is very noticeable (and thus would likely alarm the user), while being bearable for the regular user, who consciously decided to use his desktop for mining. We further require it to occupy no more than 2 cores. Thus we can use Argon2d with 4 lanes. Here also the strong tradeoff resistance of the scheme is crucial. To strengthen the resistance to tradeoff attacks, we propose to make 3 passes over memory. On our 1.8 GHz machine this scheme runs in 2 seconds.

We can estimate the resistance to tradeoff attacks of our proposal from Tables 2 and 3. The ranking method with reduction by 2 applied to 1-lane, 3-pass Argon2d yields the average depth of the recomputation tree is 20.7, thus the time penalty is 21.7. So we conclude that the time-memory product will grow at least by the factor of 7, and the same holds for the time-area product. Therefore, tradeoffs are not helpful when implementing this Proof-of-Work on ASIC, which should reduce the relative efficiency of potential ASIC mining rigs and allow more egalitarian mining process. Even if someone decides to use large botnets (10,000 machines and more), all the botnets machines would have to use the same 2 GB of memory, otherwise they would suffer huge penalty (3423 for reducing the memory by the factor of 2).

Using the 50-nm DRAM implementation [20], an ASIC mining chip would occupy at least 1100 mm^2, which is equivalent to about 30000 SHA-256 cores in the best Bitcoin 40-nm ASICs [1]. Thus the ASIC energy advantage over CPU will decrease by the factor of 2^{15}.

Even though the amount of required RAM (2GB) is small enough for desktops and laptops, in many applications it is advisable to be able to perform verification of the proof-of-work on the lightweight mobile devices. Thus it is desirable to allow the memoryless verification of the PoW. Following [15], we suggest using Merkle hash trees for the verification. As a straightforward implementation of this idea results in an interactive protocol, we apply a well-known Fiat-Shamir heuristic to make the protocol non-interactive.

8.2 Memoryless non-interactive verification.

Hash trees. Hash trees are widely used in distributed systems, and the protocol works as follows. A prover P commits to T blocks $M[1], M[2], \ldots, M[T]$ by computing the hash tree where the blocks $M[i]$ are at leaves at depth $\log T$ and nodes compute hashes of their branches. For instance, for $T = 4$ and hash function G prover P computes and publishes

$$\Phi = G(G(M[1], M[2]), G(M[3], M[4])).$$

Prover stores all blocks and all intermediate hashes. In order to prove that he knows, say, $M[5]$ for $T = 8$, (or to *open* it) he discloses the hashes needed to reconstruct the path from $M[5]$ to Φ:

$$\text{open}(M[5]) = (M[5], M[6], g_{78} = G(M[7], M[8]),$$
$$g_{1234} = G(G(M[1], M[2]), G(M[3], M[4])), \Phi),$$

so that the verifier can make all the computations. If G is collision-resistant, it is hard to open any block in more than one possible way. We suggest using the 256-bit Blake2b for G, as it is already used in Argon2.

Interactive PoW verification. Let us denote the Argon2 hash function that we use in the protocol by H. To fill 2GB of RAM, we need $T = 2^{21}$ memory blocks, so $3T = 2^{22.5}$ blocks are generated during the 3 passes. Let us enumerate them in some way from 1 to $3T$ so that block $M[i]$ is computed using the previous block $M[\text{prev}(i)]$ and the "random" block $M[\phi(i)]$. In the first step of the protocol the prover publishes the input I, the hash digest $H(I)$ and the Merkle tree root Φ of the blocks $M[1], M[2], \ldots, M[3T]$. In order to verify that the prover has computed H, the verifier V selects D random values $i_1, i_2, \ldots, i_D < 3T$ and asks P to open $M[i_j]$ and the blocks needed to compute $M[i_j]$: $M[\text{prev}(i_j)]$ and $M[\phi(i_j)]$ for all $j \le D$. An honest prover is supposed either to store all $3T$ blocks and the entire Merkle tree (the total memory requirements are around $4T$), or to recompute them once the request is made (then T memory and $3T$ extra computations are needed). We note that the root of the tree can be computed with negligible memory (around $\log T$ hashes).

A prover may theoretically cheat by computing a different function $H' \neq H$. For example, he can produce some blocks $M[i']$ (which we call *inconsistent*) not as specified by H (e.g. by simply computing $M[i'] = G(i')$) so that the preimage of $M[i]$ is not among the blocks used to compute the hash tree. In contrast to the verifiable computation approach, our protocol allows a certain number of inconsistent blocks. Instead, we are to prove the following properties:

- If the number of incosistent blocks does not exceed ϵT for some ϵ, the new function H' remains memory-hard, and the time and computational penalties for a memory-reducing prover translate from H to H' slightly weakened.
- The probability γ that ϵT or more inconsistent blocks evade detection by the verifier can be made arbitrarily low by selecting proper D.

Thus we will demonstrate that either the prover has to compute a memory-hard function (not differing too much in penalties from H) or he will be caught with probability at least $1 - \gamma$.

Let us prove the first property. The inconsistent blocks cut the functional graph of H in ϵT points. One may expect that the resulting graph(s) can be computed using far less memory. However, this intuition is incorrect. As noticed in [15], ϵT inconsistent blocks can be modelled as giving ϵT extra memory to the prover (i.e. the prover could have calculated these blocks consistently and just stored them). The computational and time penalties imposed on the memory-reducing prover would be calculated as

$$C_{H'}(q) = C_H\left(\frac{q}{q\epsilon + 1}\right);$$
$$D_{H'}(q) = D_H\left(\frac{q}{q\epsilon + 1}\right).$$

Let us give an example. Let us set $\epsilon = \frac{1}{4}$, which is equivalent to $T/4$ memory additionally available for the prover. If he attempts to compute H using $1/4$ of total memory and also cheat by having $T/4$ inconsistent blocks, his time penalty would be $T(1/4 + 1/4) = 21.7$ (Table 3). Thus this memory reduction does not reduce the time-memory product, as the time-memory product increases by the factor of 5. For further memory reductions the win might be better (as the time penalty never exceeds $D(4)$), but the high computational penalties (2^{32} for using $1/4$ of memory, cf. Table 2) make this advantage inexploitable: there could be no such parallelism in practice.

Now let us bound the cheating probability by the value γ. To ensure that, the random D blocks must interleave with a pre-selected ϵT blocks (out of $3T$ total blocks) with probability at least $1 - \gamma$. Thus we obtain the following condition on D:

$$(1 - \epsilon/3)^D \le \gamma \;\Leftrightarrow\; D \ge \log_{1-\epsilon/3} \gamma.$$

We suggest two parameter sets:

- Set 1: $\epsilon = 1/4, \gamma = 2^{-64}$, where cheating is infeasible.
- Set 2: $\epsilon = 1/8, \gamma = 1/100$, where cheating might work in 1% of blocks but the time and computational penalties are so high that it is worthless.

We get

$$D_1 \geq \log_{2^{-0.13}} 2^{-64} \approx 509.8;$$
$$D_2 \geq \log_{2^{-0.06}} 2^{-6.64} \approx 108.1.$$

Thus it is sufficient to request 510 blocks in the first case and 109 blocks in the second case. Each opening consists of 3 blocks and $\log 3T = 23$ 32-byte hashes per block, thus in total the verifier receives about $510*3\cdot(1+23/32) = 2630$ KBytes in Set 1 and 562 KB in set 2. The actual tag size should be even smaller as $M[i]$ and $M[\text{prev}(i)]$ usually share most of the path to the hash tree root.

Non-interactive verification. Finally, we apply the Fiat-Shamir approach [16] to convert an interactive protocol into a non-interactive one. For this we simply require that the indices $i_1, i_2, \ldots, i_D$ be taken from applying G to Φ multiple times. As we need $\log 3T = 23$ bits per index, in total we apply the hash function $(510 \cdot 23)/256 \leq 46$ times in set 1 (10 times in Set 2). The output strings join to form 510 23-bit indices, each referring to a block produced while computing H.

To summarize, together with the output hash, the prover publishes the hash tree root Φ, which determines 510 (resp., 109) memory blocks to open. These blocks are also opened and published, altogether forming a memory-hard Proof-of-Work. As noted earlier, the prover either stores all intermediate blocks and the entire hash tree (which totals to $4T$ blocks) or recomputes H and the hash tree (thus spending $3T$ extra calls to F).

We note that our proof-of-work can be easily combined with the standard concept of difficulty [26], where some fixed number of trailing zeros must be present in the hash value to become a currency winning block. In this case the natural strategy is to recompute the blocks and the tree for the verification only if a valid currency block is produced, so the total overhead is negligible.

9 Acknowledgments

We thank Brian Shaft for proofreading the paper.

10 Conclusion

In this paper we developed a rigorous approach to the design of memory-hard functions, which would maximize their running costs on custom hardware. We showed how to implement parallelism in a tradeoff-resilient way using threads and synchronisation points. We outlined design criteria for the internal compression function and the indexing functions. We designed a concrete proposal with both data-dependent and data-independent addressing, which is faster, simpler and more tradeoff-resilient than existing alternatives like scrypt. Finally, we showed how to design a memory-hard proof-of-work with efficient and memory-light verification for cryptocurrencies.

We can outline several interesting directions for future work. First, one can try to design a GPU-oriented memory-hard function following our methodology so that high memory latency is not a problem. Secondly, new and better tradeoff algorithms are needed, for multi-pass schemes in particular. Finally, the link between the time-area products, time and computational penalties, and the attack costs should be investigated in detail.

References

[1] Avalon asic's 40nm chip to bring hashing boost for less power. http://www.coindesk.com/avalon-asics-40nm-chip-bring-hashing-boost-less-power/.

[2] eBay hacked, requests all users change passwords. http://www.cnet.com/news/ebay-hacked-requests-all-users-change-passwords/.

[3] Litecoin: Mining hardware comparison. https://litecoin.info/Mining_hardware_comparison.

[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. available at https://en.bitcoin.it/wiki/Mining_hardware_comparison. We compare 2^{32} hashes per joule on the best ASICs with 2^{17} hashes per joule on the most efficient x86-laptops.

[7] Abadi, M., Burrows, M., Manasse, M. S., and Wobber, T. Moderately hard, memory-bound functions. *ACM Trans. Internet Techn. 5*, 2 (2005), 299–327.

[8] Alwen, J., and Serbinenko, V. High parallel complexity graphs and memory-hard functions. *IACR Cryptology ePrint Archive 2014/238*.

[9] Aumasson, J., Neves, S., Wilcox-O'Hearn, Z., and Winnerlein, C. BLAKE2: simpler, smaller, fast as MD5. In *ACNS'13* (2013), vol. 7954 of *Lecture Notes in Computer Science*, Springer, pp. 119–135.

[10] Bernstein, D. J. Cache-timing attacks on aes. Tech. rep., 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[11] Bernstein, D. J., and Lange, T. Non-uniform cracks in the concrete: The power of free precomputation. In *ASIACRYPT'13* (2013), vol. 8270 of *Lecture Notes in Computer Science*, Springer, pp. 321–340.

[12] Biryukov, A., and Khovratovich, D. Tradeoff cryptanalysis of memory-hard functions. Tech. rep., 2015. `https://orbilu.uni.lu/handle/10993/20043`.

[13] Chang, D., Jati, A., Mishra, S., and Sanadhya, S. Rig: A simple, secure and flexible design for password hashing. In *Inscrypt'14* (2014), Lecture Notes in Computer Science, to appear, Springer.

[14] Dwork, C., Goldberg, A., and Naor, M. On memory-bound functions for fighting spam. In *CRYPTO'03* (2003), vol. 2729 of *Lecture Notes in Computer Science*, Springer, pp. 426–444.

[15] Dziembowski, S., Faust, S., Kolmogorov, V., and Pietrzak, K. Proofs of space. *IACR Cryptology ePrint Archive 2013/796*.

[16] Fiat, A., and Shamir, A. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings* (1986), A. M. Odlyzko, Ed., vol. 263 of *Lecture Notes in Computer Science*, Springer, pp. 186–194.

[17] Forler, C., List, E., Lucks, S., and Wenzel, J. Overview of the candidates for the password hashing competition - and their resistance against garbage-collector attacks. *IACR Cryptology ePrint Archive 2014* (2014), 881.

[18] Forler, C., Lucks, S., and Wenzel, J. Catena: A memory-consuming password scrambler. *IACR Cryptology ePrint Archive, Report 2013/525*. non-tweaked version `http://eprint.iacr.org/2013/525/20140105:194859`.

[19] Forler, C., Lucks, S., and Wenzel, J. Memory-demanding password scrambling. In *ASIACRYPT'14* (2014), vol. 8874 of *Lecture Notes in Computer Science*, Springer, pp. 289–305. tweaked version of [18].

[20] Giridhar, B., Cieslak, M., Duggal, D., Dreslinski, R. G., Chen, H. M., Patti, R., Hold, B., Chakrabarti, C., Mudge, T. N., and Blaauw, D. Exploring DRAM organizations for energy-efficient and resilient exascale memories. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2013)* (2013), ACM, pp. 23–35.

[21] Hopcroft, J. E., Paul, W. J., and Valiant, L. G. On time versus space. *J. ACM 24*, 2 (1977), 332–337.

[22] Jr, M. A. S., Almeida, L. C., Andrade, E. R., dos Santos, P. C. F., and Barreto, P. S. L. M. The Lyra2 reference guide, version 2.3.2. Tech. rep., april 2014.

[23] Lengauer, T., and Tarjan, R. E. Asymptotically tight bounds on time-space trade-offs in a pebble game. *J. ACM 29*, 4 (1982), 1087–1130.

[24] Malvoni, K. Energy-efficient bcrypt cracking. Passwords'14 conference, available at `http://www.openwall.com/presentations/Passwords14-Energy-Efficient-Cracking/`.

[25] Mukhopadhyay, S., and Sarkar, P. On the effectiveness of TMTO and exhaustive search attacks. In *IWSEC 2006* (2006), vol. 4266 of *Lecture Notes in Computer Science*, Springer, pp. 337–352.

[26] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. `http://www.bitcoin.org/bitcoin.pdf`.

[27] National Institute of Standards and Technology (NIST), available at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`. *FIPS-197: Advanced Encryption Standard*, November 2001.

[28] Percival, C. Stronger key derivation via sequential memory-hard functions. `http://www.tarsnap.com/scrypt/scrypt.pdf`.

[29] Peslyak, A. Yescrypt - a password hashing competition submission. Tech. rep., 2014. available at `https://password-hashing.net/submissions/specs/yescrypt-v0.pdf`.

[30] Pippenger, N. Superconcentrators. *SIAM J. Comput. 6*, 2 (1977), 298–304.

[31] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. 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* (2009), pp. 199–212.

[32] Sprengers, M., and Batina, L. Speeding up GPU-based password cracking. In *SHARCS'12* (2012). available at `http://2012.sharcs.org/record.pdf`.

[33] Thompson, C. D. Area-time complexity for VLSI. In *STOC'79* (1979), ACM, pp. 81–88.

A Memory bandwidth in tradeoff attacks

The maximum memory bandwidth Bw_{max} is a hypothetical upper bound on the memory bandwidth on the adversary's architecture. Suppose that for each call to G an adversary has to load $R(q)$ blocks from the memory on average, where q is the memory reduction factor. Therefore, the adversary can keep the execution time the same as long as

$$R(q)Bw \leq Bw_{max},$$

where Bw is the bandwidth achieved by a full-space implementation.

Lemma 1.

$$R(q) = C(q).$$

Proof. Let A_j be the computational complexity of recomputing $M[j]$. If $M[j]$ is stored, then $A_j = 0$. When we have to compute a new block $M[i]$, then the computational complexity C_i of computing $M[i]$ (measured in calls to F) is calculated as

$$C_i = A_{\phi_2(i)} + 1.$$

and the total computational penalty is calculated as

$$C(q) = \frac{\sum_{i<T}(A_{\phi_2(i)} + 1)}{T}.$$

Let R_j be the total number of blocks to be read from the memory in order to recompute $M[j]$. The total bandwidth penalty is calculated as

$$R(q) = \frac{\sum_{i<T} R_{\phi_2(i)}}{T}.$$

Let us prove that

$$R_j = A_j + 1. \tag{5}$$

by induction.

- We store $M[0]$, so for $j = 0$ we have $R_0 = 1$ and $A_0 = 0$.
- If $M[j]$ is stored, then we read it and make no call to F, i.e.

$$A_j = 0; \quad R_j = 1.$$

- If $M[j]$ is not stored, we have to recompute $M[j-1]$ and $M[\phi_2(j)]$:

$$A_j = A_{j-1} + A_{\phi_2(j)} + 1 = R_{j-1} - 1 + R_{\phi_2(j)} - 1 + 1 = \\ = (R_{j-1} + R_{\phi_2(j)}) - 1 = R_j - 1.$$

  The last equation follows from the fact that the total amount of reads for computing $M[j]$ is the sum of necessary reads for $M[j-1]$ and $M[\phi_2(j)]$.

Therefore, we get

$$C(q) = \frac{\sum_{i<T}(A_{\phi_2(i)} + 1)}{T} = \frac{\sum_{i<T} R_{\phi_2(i)}}{T} = R(q).$$

□

B scrypt

Here we describe a simplified version of scrypt, which is used for the time-area comparison with Argon2. One-threaded scrypt first fills the memory as

$$M[1] \leftarrow H(\text{Input});$$
$$M[i] \leftarrow G(M[i-1]),\ 1 < i < T.$$

Here H is cryptographic hash function, and G is a transformation, which is not cryptographically strong. Then we initialize $X \leftarrow M[T]$ and do a pseudo-random walk:

$$X \leftarrow G(X \oplus M[X \pmod T]).$$

Finally, we output $H(X)$.

Tradeoff for scrypt It is well known (and recently formalized in [19]) that there is a simple tradeoff for scrypt. Indeed, suppose that we store every q-th memory block. Then on each step in the second phase we have to recompute $(q-1)/2$ blocks on average. Since the total number of calls to G is $2T$, we get the following formula for penalties:

$$C(q) = T(q) = \frac{2T + T(q-1)/2}{2T} = \frac{q+3}{4}. \tag{6}$$

C Ranking tradeoff method

The idea of the ranking method [12] is as follows. When we generate a memory block M_l, we make a decision, to store it or not. If we do not store it, we calculate the access complexity of this block — the number of calls to F needed to compute the block, which is based on the access complexity of M_{l-1} and $M_{\phi(l)}$. The detailed strategy is as follows:

1. Select an integer q (for the sake of simplicity let q divide T).
2. Store M_{kq} for all k;
3. Store all r_i;
4. Store the T/q highest access complexities. If M_i refers to a block from this top list, we store M_i.

The memory reduction is a probabilistic function of q. The results are given in Table 1.

We conclude that for data-dependent one-pass schemes the adversary is always able to reduce the memory by the factor of 4 and still keep the time-area product almost the same.

D Cheat detection

In the second set of parameters cheating is rare though still possible. As the inputs to H are public, any other user can run it on his own machine if he has sufficient memory. However, even he gets a different result, which signals of cheating, he still has to communicate this news to the other users and convince them that the original block is incorrect.

We suggest a simple protocol that marks the cheated hashes invalid and does not require human interaction or any other consensus. Suppose that a user U runs H on some input I and finds out that $H(I) \neq H_0$, the published hash digest of I. We suggest U to run another iteration of the verification protocol using $2D$ block requests instead of D, and to publish the new tag, which becomes twice as large. If the tag is valid, a cryptocurrency protocol should accept it and mark the malicious block as invalid.

It is again possible that U cheats as well, but the cheating probability is changed from γ to γ^2. To detect this "second-order" cheat, yet another user should run the verification protocol with $4D$ block requests and so on.