Evaluation of the MTP-Argon2 PoW Scheme

Fabien Coelho
firtname.lastname@mines-paristech.fr
MINES ParisTech, PSL Research University,
with Doubloon Skunkworks support

Version 1.07 on September 28, 2017

Abstract

This report provides a critical and argumented evaluation of the MTP-Argon2 PoW scheme presented in Section 4.5 of [7], loosely based on the Argon2 [3] memory-hard password hashing function, including the Dinur-Nadler [11] and Bevand [6] attacks. It also provides new proposals which attempt to fix the known deficiencies and improve the memory hardness of the scheme.

Contents

1 Introduction

Proof-of-Work (PoW) functions [13] are designed to be hard to compute but easy to check. There are challenge-response (interactive) and solution-verification (no direct interaction between prover/miner and verifier) protocol variants. Their hardness may be based on computations, memory accesses [1, 2, 12, 9] or possibly other criteria. Memory hardness claims have been made about latency, bandwith or amount of memory required for a task. In [17], memory hard is defined as a function which requires memory proportionally to its computation cost. In the context of crypto-currencies, the solution-verification PoW scheme is an essential building block which allows to distribute the rewards between miners which maintain the distributed ledger, aka blockchain. Finding schemes for which ASIC or FPGA hardware implementations do not show an undue advantage over CPU or GPU software implementations is key to avoid the kind of mining power concentration that has occurred with Bitcoin. When ASICs are considered, requiring significant memory is seen as a good deterrent.

2 MTP-Argon2 PoW

The *Egalitarian Computing* paper [7] presents various applications loosely based on the Argon2 [3] memory-hard password hashing function for Proof-of-Work. The presented scheme includes instantiation settings for crypto-currency applications, time-lock puzzles and disk encryption. Although the Argon2 function could be used directly for such purpose, its very high single execution cost is somehow prohibitive for the verifier, thus schemes displaying smaller verifier cost but yet requiring large memory are sought. We focus here on the PoW scheme introduced in the paper and the proposed crypto-currency instantiation.

The proposed PoW construction initial step is similar to [10]: it builds a Merkle-Tree over an array and pseudo-randomly selects a subset of leaves based on the root hash of the tree as proof of computation. The feedback loop means that changing inconvenient leaves require recomputing the root hash, thus would change the leaf selection function. If enough leaves are provided, it ensures that most of the Merkle tree has been computed. However, unlike [10], the array is memory intensive and the PoW does not end at the Merkle tree proof, which is rather intended as a proof of the fixed-cost array computation. It adds a more classical iterative hash partial inversion search on top of the construction, which also depends on the contents of the large array. The idea is to provide some memory-hardness property by showing that the large array was computed and stored, and also that a probabilistic iterative search was performed. The naming of the scheme (MTP-Argon2 PoW – Merkle Tree Proof Argon2 Proof-of-Work) seems unfortunate, as the Merkle tree aspect of the proof is unrelated to the actual PoW scheme and it does not actually use the Argon2 function.

2.1 Initial Description

The following notations are taken for the paper and used:

I challenge identifier, *e.g.* a 32 or 64 bytes hash of something...

T memory size in KiB, a power of 2 *(why T?)*.

L length of one search, which induces the proof size and verifier cost.

d strength of PoW, number of expected zeros in final hash value.

$H_s(x)$ a variable-size (s bytes) cryptographic hash function. The paper uses BLAKE2 [4] ($1 \leq s \leq 64$) with an extension to larger sizes which costs about one call every 32 bytes (Section 3.2 of [3]). For complexity purposes with BLAKE2, we count a unit of cost per 128-bytes block of input data, which encompasses 12 calls to the underlying compression function.

$x' = P(x)$ Permutation P performs 8 calls to compression function G loosely based on BLAKE2 internal round function (Annexe A of [3]). It takes 128 bytes at a time and uses 64-bits add ($+_{64}$) and xor ($\oplus_{64}$), 32-bits input 64-bit result multiplication ($*_{32}$ – deemed expensive on ASIC), circular 64-bits shifts ($\gg_x$) and left 64-bits shifts (aka multiply by 2).

$B' = F(B_0, B_1)$ a compression function which takes two 1 KiB inputs, produces one 1 KiB output taken from Argon2, and which relies on 16 calls to Permutation P internally (Section 3.4 of [3]). Total computation cost is about 128 compression function calls, or about $c_F = 11$ BLAKE2 128-byte processing calls.

$\phi(i)$ a contextual indexing function which depend on the current iteration, on the phase of the computation, on the data dependent (d) or independent (i) variant, on the degree of parallelism... The data-dependent variant relies on the previous element contents and implies an integer modulo operation to select an element among the already computed ones, *i.e.* we have something like: $\phi(i) = \phi_d(X[i-1]) \bmod (i-1)$.

The initial description [7] of the PoW search algorithm (prover) is, from I, L and d:

1. Build memory $X[1 \dots T]$:
   (a) $X[1] = H_{1024}(I)$
   (b) $X[i] = F(X[i-1], X[\phi(i)])$ for $i > 1$
2. Compute Merkle-tree root Φ of X with H_{16}
3. Choose Nonce N
4. $Y_0 = H_?(\Phi || N)$
5. For $1 \leq j \leq L$ compute:
   (a) $i_j = Y_{j-1} \bmod T$
   (b) $Y_j = H_?(Y_{j-1}, X[i_j])$
6. If Y_L has d trailing zeros, the PoW search ends, otherwise go to Step 3
7. Final output is $(\Phi, N, \mathcal{Z})$ where $\mathcal{Z}$ is the *opening* (Merkle tree proof) of all $X[i_j]$ memory antecedents, namely the $2L$ elements $X[i_j - 1]$ and $X[\phi(i_j)]$.

Step 1 and 2 constitute the initialization phase. The actual search loop is between Step 3 and 6. Final Step 7 returns the PoW. The PoW verification part is then:

1. Check Merkle tree proofs $\mathcal{Z}$
2. Compute $X[i_j]$ with F and provided X values
3. Compute Y_L from N, Φ and computed X values and checks that it ends with d zeros

Memory accesses on Array X occur while building the array on Step 1 and at each iteration of the search in Step 5b.

This algorithm description from the paper include some deficiencies, most of which probably due to over-simplification and lack of proof-reading:

- Indexing function ϕ must access existing blocks in Array X, thus $1 \leq \phi(i) < i$, so we must have $\phi(2) = 1$, hence from Step 1b it follows that $X[2] = F(X[1], X[1])$.

  Compression Function F is taken from Argon2, and is such that $F(B_0, B_1) = F'(B_0 \oplus B_1)$, thus $X[2] = F'(X[1] \oplus X[1]) = F'(0)$ is a constant independent of I.

Function F is built on top of P which repeatedly combines integers without using any constant. Adding, xoring, multiplying and shifting 0 leads to 0, so $P(0) = 0$, thus $X[2] = F'(0) = 0$.

Indexing function $\phi(i)$ in the Argon2d variant depends on the first bytes of $X[i-1]$, so that $\phi(3) = 2$, thus $X[3] = F(X[2], X[2]) = 0$ (?)

By induction, we then have $\forall i, 1 < i \leq T, X[i] = 0$.

- Step 5a computes $i_j \in [0, T)$, but the index is used to access Array X which is defined on $[1, T]$.
- Step 5b uses $H(Y, X)$ which is not fully defined and could be a weakness, see Section 3.7.
- Step 7 does not specify what to do if $i_j = 1$ and there are no antecedents.
- Step 7 provision of Φ is redundant with the Merkle tree proofs and verification from Nonce N which allows to recompute and check it.
- As noted in [11], the verification does not check that provided proofs are linked to Challenge I, unless $X[1]$ happens to be included in the solution, and even that is not enough.
- Also, the verification should check that all provided X values are used.
- The precise form of the Merkle tree proof and its associated verification for a set of leaves is not specified, as also noted as Attack 2 and 3 in [6]. We will assume that this verification is complete, that is the location, value and usage of all leaves is actually checked.
- The size of the intermediate hashes is not clearly specified, we assume that it is 16 bytes.

2.2 Amended Description

In order to fix the various issues raised above, we suggest the following PoW search algorithm, which is an attempt at clarifying the initial description. The array and variable indexing starts from 0 following usual conventions. From I, L and d, we compute:

1. Build memory $X[0 \ldots T-1]$:
   (a) $X[0 \ldots 1] = H_{2048}(I)$
   (b) $X[i] = F(X[i-1], X[\phi(i)])$ for $i \geq 2$ and with $0 \leq \phi(i) < i - 1$
2. Compute Merkle-tree root Φ of X with H_{16}
3. Choose nonce N
4. $Y_0 = H_{16}(\Phi || N)$
5. For $1 \leq j \leq L$ compute:
   (a) $i_{j-1} = Y_{j-1} \bmod T$
   (b) $Y_j = H_{16}(Y_{j-1} || X[i_{j-1}])$
6. If Y_L has d trailing zeros, the PoW search ends, otherwise go to Step 3
7. Final output is $(N, \mathcal{Z})$ where $\mathcal{Z}$ is the *opening* (Merkle Tree Proof) of the L $X[i_j]$ memory antecedents namely $X[i_j - 1]$ and $X[\phi(i_j)]$ if $i_j \geq 2$, or $X[i_j]$ if $i_j < 2$.

Computation Complexity The computation complexity in hash-block calls (with F counted as 11 calls) is $20 \cdot T + 41 + 2^d(9 \cdot L + 1) \propto (2 \cdot T + 2^{d-1} L)$ (if we admit that $10 \approx 9$ for the nonce).

Memory Access Complexity The memory accesses on Array X occur at Step 1, 2 and 5b. In Step 1, there are 2 accesses for each built element, one being the element just built $i-1$, and the other pseudo randomly through $\phi(i)$. It must be noted that Function ϕ (Section 3.2 of [3]) is not uniform: recent indexes are chosen more often with a quadratic probability, leading to improved cache effects. The rational for this is not clearly expressed, but it may be to try to counter the fact that the initial sweep for building X would use more often indexes at the beginning of the array. Hash of leaves needed by Step 2 can be computed on the fly while building Array X. In Step 5b accesses are pseudo-randomly uniform over all Array X. On average, the total number of array accesses during a search is thus about $3T + 2^d L$, which is comparable to the computation complexity.

Partial Dependency on Challenge How to check that the proofs is indeed linked to Challenge I is unclear, as noted in [11]. Including $X[0]$ in $\mathcal{Z}$ is not enough if it is not shown that other array values are linked to it as well. This issue will be addressed in Section 4.2 by using the challenge at every step of the computation.

Crypto-currency Instantiation The suggested instantiation for crypto-currency, called MTP-Argon2 in Section 4.5 of [7], is to use the Argon2d (data-dependent) variant with 4 parallel lanes (point discussed in Section 3.8), use hash function BLAKE2 [4] for H, use H_{16} for the Merkle tree, $T = 2^{21}$ so that Array X is 2 GiB, and $L = 70$, which leads to $2^{25.3} + 2^{d+9.3}$ hash-block operations and $2^{22} + 2^{d+6.1}$ memory accesses on X.

Choice of BLAKE2 The choice of hash function BLAKE2 for a memory-hard PoW is reasonable. It requires a small 336 bytes of RAM, which is not an argument when building a memory-hard PoW scheme. However, BLAKE2 uses about half cycle-per-byte compared to SHA-3, which for memory-hard PoW purpose where the expectation is to emphasize memory accesses seems a reasonable choice.

Choice of Length The proposed choice of Lengh L looks somehow arbitrary. In particular, there is no clear indication on how it should change depending on other parameters (*e.g.* memory size T, possibly PoW strength d). The paper includes a proof that this choice induces a limited $\frac{1}{12}$ advantage for ASIC implementations with these particular parameters (although the proof seems to suggest that a cheating test failure cost as much as non cheating test failures, which may not strictly the case as a cheating test failure can be detected early when an inconsistent block is required?). Other security criteria push towards a reasonably large L, but for the resulting proof size and verification costs. First, it should be large enough so that computing only a fraction of X impairs the search algorithm significantly. Equation (2) in [10], for a related problem, provides a relative cost lower bound is for a provable Merkle tree computation which depends on the required number of proofs $2L$ and number of leaves T. With these notations, the formula translates to:

$$f \geq T^{\frac{-1}{2L+1}} \frac{2L}{2L+1}$$

thus $L \approx \frac{-1}{2 \cdot \log_2 f} \cdot \log_2 T$. Choosing $f = 0.9$ as [10] leads to $L = 3.3 \cdot \log_2 T = 69$, which is consistent with [7] choice, but also provides an interesting guideline if T changes. Second, it should ensure that Step 1 initialization phase is negligeable compared to the search phase so that the search is *progress free*, thus we want $2T \ll 2^d L$. This is also consistent with having a search cover most elements $T \ll 2^d L$.

Constraints on Strength As discussed in the previous paragraph, the consistency of the scheme with respect to its objectives is that the search should cost more than the array initialization. Given the other parameters, this leads to $d \geq 16$. A trivial constraint is that Strength d is smaller than the hash size it targets, the limit of the partial inversion being a full inversion of the hash function. With a $S = 16$ bytes hash size, we understand that $d \leq 8 \cdot S = 128$. The actual choice for d in $[16, 128]$

depends on the implementation speed, available computation power, and desired frequency of finding a solution. For reference, bitcoin in 2017 typically requires under 2^{70} relatively inexpensive SHA-256 operations for each block. A working memory hard scheme would be expected to be significantly harder, although the reality of this is yet to be determined. BLAKE2 hardware efficiency is similar to SHA-256 [15], and one candidate hash with defaults settings requires 631 invocations, so the relative computation ratio over Bitcoin's SHA-256 is about $2^{9.3}$. A somehow large mining power operation where dozen millions of cores would compute one billion candidate hash per second and find a hit every few minutes leads to $d \leq 70$. In the following, we will assume $d \leq 70$ as large but still realistic and $d \leq 100$ as conservative.

Size of Proof The size of the resulting PoW is large: it involves about $2L$ 1 KiB blocks and their associated Merkle tree proofs. Note that the Merkle tree proofs of a set of leaves of the tree is smaller that the cumulated proofs for each leaf: assuming 16 byte hashes, it is about (some leaves may occasionally overlap) $2 \cdot L \cdot 16(\log_2 T - \log_2 2L) + L\frac{1}{8}\log_2 T$ bytes (first part for the needed intermediate hashes, second part for half the leaf positions as the others can be recomputed with ϕ). The overall size is about 171 KiB (lower than the 180 KiB evaluation in [7]) and induces anyway a significant burden on a crypto-currency chain.

The intention of the Merkle Tree Proof requirement is that it allows to check that the prover did indeed build the array. However the attack paper [11] shows that this is not the case, as discussed in Section 3.4. It mostly really shows that some Merkle tree computation did take place.

It is interesting to note the search state size, which is the current values of Nonce N (11 bytes?), Iteration j (1 byte?) and current Hash Y (16 bytes?), that is about 28 bytes, which is quite small. This will be discussed further in Section 3.5.

Number of Antecedants The proposed schemes suggests to include one level antecedents of the needed array elements. The scheme could have considered going multiple levels, *e.g.* two levels and recomputing both levels:

$$X[i] = F(F(X[i-2], X[\phi(i-2)]), F(X[\phi(i)-1], X[\phi(\phi(i)-1)]))$$

Alternately, the array element could depend on more previous array elements with limited additional computing costs:

$$X[i] = F(X[i-1], X[\phi_1(i)], X[\phi_2(i)], \ldots) \tag{1}$$

However, such changes would reduce search Length L for a constant number of proofs, which does not seem desirable as it would reduce array accesses. A positive impact of adding dependencies is that the Dinur-Nadler attack precomputations costs would be significantly enlarged. At the minimum, the impact of such possible changes to the design should be evaluated carefully. We investgate such options in our proposal Section 4.3.

3 Discussion

We discuss various issues about memory-hard schemes and their applicability as PoW schemes. We measure the benefit of an attack as a couple (α, σ) with α the memory saving as a fraction of the standard memory and σ the associated computation cost as a multiplier of the standard. We propose that for a memory-hard PoW function, any scheme saving half the memory or above $\alpha \geq \frac{1}{2}$ should induce a significant $\sigma \geq 64$ cost multiplier up to conservative Strength $d \leq 100$.

3.1 MTP Proof of What?

The Merkle tree proof part of the PoW in [7] shows that most Array X elements at the leaf are so that $X[i] = F(X[i-1], X[\phi(i)])$ holds. It does not follow that these elements where computed from the

provided challenge, nor that they were stored in memory instead of being possibly recomputed when doing the iterative search. At the minimum, this means that computing Array X and the Merkle tree root hash are not necessary, as reusing a previous instance would work as well, as taken advantage by the Dinur-Nadler attack discussed in Section 3.4.

3.2 Memory Saving Attack

Section 4.2 in [7] also presents a *memory saving* attack and argue that the resulting space-time complexity is prohibitive for significant compressions.

Let us consider $\alpha = \frac{1}{2}$ compression of Array X where every two elements are discarded, the other one being kept. When running the search loop the prover has 50% probability that a needed element is available and can be processed directly, and 50% that is must be recomputed. The cost of the recomputation it to apply F on the previous available element and the $\phi(i)$ element which may in turn be available or not, and so on recursively. The average recomputation cost in F calls is: $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 1 + \ldots \approx 1$. The average number of X accesses is: $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 2 + \ldots \approx 2.5$. This show that for a limited recomputation cost the memory requirement for an ASIC implementation can be halved, and under some cache size this might be interesting even for CPU implementations. This attack benefit is $(0.5, \approx \frac{c_X + c_F}{c_X} \approx 2.2)$: half memory is saved but accessing an array element costs c_F on average instead of 0, which impact the overall search cost.

The good news is that this cost increases very quickly when the memory saving is increased further (Section 5.1 in [3]). With $\alpha = \frac{2}{3}$ on average the recomputations have to randomly walk back to the first blocks of Array X inducing about $\log_2(T)$ F computations per element. With $\alpha = \frac{3}{4}$, and $T = 2^{21}$, average cost per element is over 100,000 F computations (per a numerical simulation), as most elements and their dependencies must be recomputed. These data are consistent with some memory hardness claims of [7] and [3], although this does not prevent the Dinur-Nadler pre-computation attack presented in Section 3.4.

One way to improve the resistance to memory compression could be to make F rely on more previous elements, as outlined by Equation 1. For $\alpha = \frac{1}{2}$ and with 3 dependencies instead of 2, the average recomputation cost in F calls would then be: $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 + 2 \cdot \frac{1}{4} \cdot 1 + 4 \cdot \frac{1}{8} \cdot 1 + \ldots$. This construction diverges slowly along the hops, similarily to the one-in-three compression and two dependent elements, thus induces significant recomputation costs. The exact average cost depends on the various parameters (array size, number of dependencies, which elements are available) and on the biased pseudo-random index functions. It can be computed with simulations.

3.3 Raw Recomputation Attack

A baseline attack which breaks the memory requirement is simply to recompute each value when needed, recursively following dependencies. This seems to requires about $\frac{\log_2 T}{\log_2 1.5}$ elements space. The cost of computing the ith element is about $c_i \approx 1 + c_{i-1} + c_{\frac{2}{3}(i-1)}$, as it depends on its previous element and on a pseudo-randomly chosen one with a biased selector, and starting from $c_0 = 0, c_1 = 0, c_2 = 1$. This sequence growths steadily, the average cost per element with the default parameters is $2^{263.1}$ calls to F (per a numerical simulation). The attack benefit is about $(\approx 1.0, c_F \cdot 2^{263.1})$: most memory is saved, at the price of many computations. The Dinur-Nadler attack presented next improves substantially over this baseline.

3.4 Dinur-Nadler Attack

A detailed and theoretically convincing *cheating* attack is presented in [11], which is somehow a clever application of the *block modification* attack (Section 4.2 in [7]) with an added twist to allow recomputing many blocks instead of storing them, at the price of some precomputations.

The paper shows that MTP PoW memory requirements claims [7] can be avoided by precomputing a special Array X which mostly respects the F computations but the iterative construction is broken

at regular points which allow to reconstruct missing X values from a few kept values (named *control blocks* in the paper), thus avoid storing the whole array. The attack is based on control blocks which are kept stored, with a parametric t compression, so that T/t regularly spaced blocks are kept. These control blocks could be pseudo-randomly generated from a specific nonce, lowering further the memory requirements, although at the price of additional computations to rebuild X values when needed. The reconstruction cost from control block is bounded by carefully choosing those so that $\phi(i)$ functions happen to use control blocks as well, avoiding a deep recursion to rebuild missing values. The authors also discuss a trade-off where the precomputation condition is relaxed, lowering the precomputation cost, at the price of a more expensive search (Section 5.1 of [11]). We will not consider this trade-off as our analysis is mostly driven by the search phase cost.

The special array X construction involves pre-computing suitable control blocks so that all following $(t-1)$ blocks depend on the preceding block or another available control block. In [11] the cost of this precomputation for each T/t sequence is evaluated to t^{t-2} calls to F, that is an overall chain precomputation cost $T \cdot t^{t-3}$ (Equation (7) in [11]). We refine this evaluation to the following:

$$c_F \cdot \left(\sum_{i=1}^{t-2} i(t-1)t^{-i} + (t-2)t^{(2-t)} \right) \cdot t^{t-2} \cdot \frac{T}{t} \approx c_F \cdot \alpha(t) \cdot T \cdot t^{t-3} \quad \text{with} \quad 1 \leq \alpha(t) \leq 1.25 \tag{2}$$

The cost expression counts F costs. The summation computes the cost of failing to find a sequence after i iterations starting from a chosen control block, weighted by their probability. The second is the cost is the one which succeeds. The last block does not need to be computed, it is sufficient to know that its dependencies are available. The following term is the number of expected iterations to find one sequence and the final one the number of sequences to compute. For instance choosing $t = 8$, this is equivalent to $1.14 \cdot 2^{21} \cdot 2^{3 \cdot 5} \cdot 11 \approx 2^{39.7}$ BLAKE2 calls.

However, when considering the search algorithm, the probability of finding available blocks on a partial Array X is reduced in the iterative search to $(\frac{t-1}{t})$ at each stage, which ends up being quite costly if t is small as only $(\frac{t-1}{t})^L$ iterations succeed in computing a final hash. The F calls cost to compute one Ω is evaluated to $c_F \cdot t \cdot \frac{t}{2} \cdot (\frac{t}{t-1})^L$ (Equation (9) in [11], with t the average number of iterations along L, $\frac{t}{2}$ the cost of recomputing a block, and the last term is the expected number of attempts to compute one Ω hash). This search cost can be refined, based on c_F and c_X costs:

$$\left(\sum_{i=0}^{L-1} \left(1 + c_X i + c_F \frac{t}{2} i \right) \left(1 - \frac{1}{t} \right)^i \frac{1}{t} + \left(1 + c_X L + \frac{t}{2} c_F L \right) \left(1 - \frac{1}{t} \right)^L \right) \cdot \left(1 - \frac{1}{t} \right)^{-L} \tag{3}$$

The summation computes the cost of stopping when computing Y_j because it finds a control block for which the attacker does not have suitable predecessors to produce as proof. The second part is the cost for the computation which got through the whole loop. The final term is the expected number of attempts to get one final value.

Table 1a shows numerical evaluation of the precomputation and search phase $\log_2$ costs of the Dinur-Nadler attack, to be compared to 24.46 and $d + 9.3$ without cheating. Whether the resulting scheme would really be beneficial depends on the specifics of the implementation and the expected usage pattern. For the proposed parameters, the best achievable search cost is at least $d + 15.4$, that is about $2^{15.4-9.3} = 2^{6.1} \approx 68.4$ more BLAKE2 calls[1] compared to the non cheating search. The attack benefit, without taking into account precomputation costs which are assumed amortizable over many challenges, is $(\approx 1.0, 68.4)$: most memory is saved but at a significant cost. Table 1b shows the same conservative evaluation for the scheme presented in Section 4.2, including the back sweep, to be compared to 25.0 and $d + 6.99$ without cheating. However, unlike the preceding case the precomputation is challenge-specific, thus must be included in the attack cost. The analysis yields a 218.0 multiplier for $t = 17$ with a realistic $d = 70$ and a 98.0 multiplier for $t = 22$ with a conservative $d = 100$. The attack benefit is thus at least $(\approx 1.0, 98.0)$ for this design, and shows a larger margin for more realistic settings. We show in Section 5.5 that such multipliers make the attack unbeneficial even with favorable assumptions.

[1] Given the high cost of a precomputation for those settings, a more realistic assumption leads to a multiplier over 90.

t	$\log_2$ Eq (2)	$d + \log_2$ Eq (3)
4	26.78	$d + 35.61$
6	32.48	$d + 26.13$
8	39.65	$d + 22.02$
10	47.86	$d + 19.81$
12	56.85	$d + 18.47$
20	98.01	$d + 16.28$
25	126.68	$d + 15.82$
30	156.99	$d + 15.58$
35	188.64	$d + 15.47$
40	221.41	$d + 15.41$
45	255.15	$d + 15.40$
50	289.75	$d + 15.40$

(a) $T = 2^{21}, L = 70, c_F = 11, c_X = 9$, no back sweep

t	$\log_2$ Eq (2)	$d + \log_2$ Eq (3)
4	27.32	$d + 38.19$
6	33.02	$d + 26.49$
8	40.19	$d + 21.35$
10	48.41	$d + 18.55$
12	57.39	$d + 16.83$
20	98.55	$d + 13.91$
25	127.22	$d + 13.25$
30	157.53	$d + 12.89$
35	189.18	$d + 12.69$
40	221.95	$d + 12.57$
45	255.69	$d + 12.51$
50	290.29	$d + 12.48$

(b) $T = 2^{25}, L = 84, c_F = c_X = 1$, with back sweep

Table 1: Attack Cost Numerical Evaluation in BLAKE2 Calls

Key enablers of this attack include that (1) ϕ is inexpensive and only depends on the first a few bytes of an element, (2) F is independent of the challenge which allows costly pre-computations and induces that the verifier does not really check that the computation was specific to the challenge. Although the attack pre-computation cost seems prohibitive, in the context of a crypto-currency the reward could also be large, especially as it only needs to be done once and can be reused afterwards indefinitely. In order to thwart this attack, a simple counter measure, not mentioned in [11], is to make F depend on the challenge so that the precomputation would have to be performed for each challenge. We do so in Sections 4.2 and 4.3.

Moreover, as noted in [11], finding a cheating proof could make a blockchain unstable, as a full recomputation of the proof from the parameters would not yield the validated PoW, thus some verifiers may decide not to accept the proof and create a fork at such juncture while others would accept it. We discuss this further in Section 3.8.

3.5 Parallel Searches

In the context of a probabilistic PoW the bottleneck of which is memory bandwidth, the prover is interested in performing more computation per loaded data, or with data already available in cache. In order to do that, the search paradigm can be inverted so that instead of fetching Array X elements needed for updating the Y value for nonces, it rather fetches the Y values for which the array elements are available, and scans the array in a round-robin fashion so as to make all parallel searches progress. This algorithmic change is significantly profitable if the search state for a nonce is small. The proposals below add a step to enlarge this search state beyond the (possibly) 28 bytes in [7]. This does not impact Array X memory requirement, but reduces the efficiency of a memory bandwith limited implementation, and makes it harder to amortize the 2 GiB memory cost over several solvers. A possible hardware implementation of such a solver is outlined in Section 5.6.

3.6 Memory-Hardness

Memory-hardness has been sought for based on latency, bandwidth and size.

Latency is usually significant when computing just one memory hard function. However, when doing an extensive search, such as in password enumeration or PoW, several passwords or nonces can be evaluated together and the latency of one is masked by the computation of others, so that the key limiting factor is really the memory bandwidth and not its per access latency.

So bandwith and possibly cache hit ratio is the relevant factor for computing directly the function values on a set of nonces. However, as discussed in Section 3.5, computing these functions with a

enumerative progressive approach can reduce the bandwith requirement per function computation by reusing data already in cache. If such implementation is possible for a given function, the required memory for search states becomes the limiting factor, and once available the computations can proceed at full (ASIC) speed. A key factor is the computation state size per nonce which also requires some memory and limits the applicability of the approach. This point induces the added Step 6 in our proposals (Section 4.

Another point is that if the ASIC profitability is very sensitive to memory, and as requiring a significant memory size for the auxiliary array would push ASIC designers to compute more nonces in parallel in order to amortize the array cost, then enlarging the search state memory requirement would also help limit the benefit of such designs.

3.7 Hash Composability

The composability property of hash functions may change the memory requirements significantly. Hash functions use an internal state which is updated as block of data are put in to be hashed, and the actual hash extract part or all of this state. The initial state is H_0, the final extraction is H_ω, and the H_π(block) is the state updating function, so that for instance $H(B_1||B_2) = H_\omega \circ H_\pi(B_2) \circ H_\pi(B_1) \circ H_0$.

When using such a composable hash function for memory hardness, a potential issue is that initial blocks may be preprocessed so that only the hash function state needs to be kept and the actual block content can be discarded, replacing a large memory block by a smaller hash state. In order to avoid this issue, it is important that $H(x, y)$ in [7] is really implemented as $H(x||y)$ and that on each call parameters are sorted so that the most recently known value is processed first by the hash function.

3.8 Parallel Function

The Argon2 [3] specification emphasizes the internal parallelism of the memory-hard hash function with a parametric loosely interdependent number of lanes which induce some issues of its own (Attack 1 and 4 in [6]). Using parallelism makes sense for a password checking algorithm, as most devices are now multi-core and expected to grow. On the verifier side, as it must fully recompute the value to check for the password, taking advantage of parallelism provides both speed and yet consummes resources. On the password cracking attacker side, parallelism will be used anyway to enumerate passwords. Whether this property is desirable in a crypto-currency memory-hard PoW context is at the least debatable. It could be chosen to make it highly parallel or not parallel at all.

The PoW scheme prover part first builds a large array, roughly with 4 degree of parallelism (4 lanes suggested in [7]). This is a fixed cost (and time) to be incurred by each prover group before starting the classical partial hash inversion search. It makes the search less *progress free* unless this phase is negligeable compared to the search itself and can be amortized. Whether this is the case depends on the detailed settings and implementations.

On the verifier side, the array construction is partially verified thanks to the MTP part of the proof. However, as noted the verification is only partial: for instance the Dinur-Nadler attack does not need to rely on computing the array memory and can be run in parallel to build the tweaked array.

So the alternative is:

1. Make the array construction fully sequential (*i.e.* 1 lane), which reduces some advantage of an honest ASIC prover which will have to wait for the whole array to be available before starting the partial hash inversion search, but at the price of making it harder to have a progress free setting, and to keep only a partial check on the array construction.

2. make the array construction highly parallel, for instance with independent sequences of array elements of some length, provided that it proovably does not reduce the memory requirements of the scheme, which gives the ASIC prover some opportunity to take advantage of its hardware in the array construction phase, but could also provide the verifier with the opportunity to actually check that some sequences where computed by recomputing them.

Having a fully verifiable proof seems an attractive property, so we would recommend to allow very large paralleism for building T even if this also somehow favors large miners. Also, as this obviously breaks the memory requirement property of the scheme, the idea is that it should be in the performance interest of the solver to keep this memory over recomputations: it could be enough that fetching a precomputed value, induring some storage cost and latency penalty doing so, significantly outweight the cost of recomputing the value.

4 Proposals

We describe three memory-hard PoW schemes inspired by but trying to fix issues found in [7].

4.1 Constant Array

A key enabler of the attack [11] on MTP PoW is that it attempts – and fails – to check that Array X was indeed computed without actually recomputing it.

In the context of crypto-currencies PoW, does it matter if the array depends on Challenge I? Not necessarily... If not, then Array X may be fixed once and for all (*i.e.* constant), built from some expensive scheme so as to deter on-the-fly ASIC derivations. This cost would be amortized over the whole life of the currency and thus be negligeable. If no verification of the memory is necessary, the whole Merkle-tree phase of the PoW is not needed, nor sending array elements, which removes most of the weight of the PoW.

However, the search itself now needs to depend on Challenge I, which is achieved in the following by using a Challenge I dependent hash function:

1. Let Array $X[0 \dots T-1]$ with elements of size x be
2. No-op, so that step numbering is compatible between versions
3. Choose Nonce N
4. $Y_0 = H_S(N||I)$
5. For $1 \leq j \leq L$ compute:
   (a) $i_{j-1} = Y_{j-1} \bmod T$
   (b) $Y_j = H_S(Y_{j-1}||X[i_{j-1}] \oplus I)$
6. $\Omega = H_S(Y_L|| \dots ||Y_{1-L \bmod 2})$
7. If Ω has d trailing zeros, the PoW search ends, otherwise go to Step 3
8. Final output is N

The verification simply consists in computing and checking Hash Ω zeros from Challenge I and Nonce N using a pseudo-random walk on constant Array X.

Complexity The computation complexity in hash-block calls is $2^d \left(c_X \cdot L + \lceil \frac{S \cdot L}{128} \rceil + 1\right) \propto 2^d L$, where $c_X \approx \lceil \frac{x+S}{128} \rceil$ is the cost for hashing one Array X element at Step 5b. With $x = 1024$ and $S = 64$ this is $2^d(10.5 \cdot L + 1)$. With $x = 64$ and $S = 64$ this is $2^d(1.5 \cdot L + 1)$. The memory access complexity on Array X is $2^d L$.

Length L must be large enough so as to deter storing only part of Array X and finding a sequence of array elements which happen to be in this part. If only Fraction f of X is available, we want $f^L \ll 1$. On the other hand L is the verification cost which is desired as small as possible. Choosing $L = 32$ or $L = 64$ looks like a reasonable option, provided that $T \ll 2^d L$ so that most array elements are accessed in a search.

Size S We suggest $S = 64$ which is the maximum for one invocation of BLAKE2. Back sweep Step 6 makes one search state size at least LS bytes and adds a significant memory requirement (a few KB per nonce) for parallel searches. It somehow ensures that the memory requirement is proportional to the computation, although some trade-off is always possible: this memory is not strictly necessary, as the Y_j values could still be (re)computed from available data, but as keeping these values takes less space than their dependencies, recomputations are not worth it. If some parallel pipelining takes place, the average number of values needed is $\frac{L}{2} + 1$ per search and should be considered to dimension the memory requirements.

Array X Ideally, its contents should be fully incompressible random data. A possible construction scheme for building a large constant Array X from entropy rich Constant C could be to use the Argon2 scheme [3] using at least two sweeps ($t > 1$) over C as a password.

4.2 Fully I-dependent Array, Tree and Search

Another key anabler of the attack [11] on MTP PoW is that compression Function F is fixed, thus expensive precomputations can be done to build a special array independent of Challenge I and reuse it for each PoW. In order to avoid this issue, a challenge-dependent function can be sought instead: $B_3 = F_I(B_1, B_2)$, so that pre-computations needed for an attack would have to be specific to each challenge. This could also provides an indirect way to actually check that the computed Merkle tree is fully specific to the challenge, as well as the underlying array, as checked F_I element computations and the full Merkle tree would now depend on I.

The updated search algorithm would be, from I, L and d:

1. Build challenge dependent memory $X_I[0 \dots T-1]$ with elements of size x:
   (a) $X_I[0 \dots 1] = H_{2x}(I)$
   (b) $X_I[i] = F_I(X_I[i-1], X_I[\phi(i)])$ with $2 \leq i < T$ and $0 \leq \phi(i) < i - 1$
2. Compute Merkle-tree root Φ of X with $H_M^I(x) = H_M(x||I)$
3. Choose Nonce N
4. $Y_0 = H_S(N||\Phi)$
5. For $1 \leq j \leq L$ compute:
   (a) $i_{j-1} = Y_{j-1} \bmod T$
   (b) $Y_j = H_S(Y_{j-1}||X_I[i_{j-1}] \oplus I)$
6. $\Omega = H_S(Y_L|| \dots ||Y_{1-L \bmod 2})$
7. If Ω has d trailing zeros, the PoW search ends, otherwise go to Step 3
8. Final output is $(N, \mathcal{Z})$ where $\mathcal{Z}$ is the *opening* (Merkle Tree Proof) of the $X_I[i_j]$ memory antecedents namely $2L$ elements $X_I[i_j - 1]$ and $X_I[\phi(i_j)]$ if $i_j \geq 2$, or $X_I[i_j]$ if $i_j < 2$.

Preferred Parameters Our preferred parameters, discussed below, are: $T = 2^{25}$, $M = \lceil \frac{d+13}{8} \rceil$, $S = x = 64$, $L = 84$, $F_I = H^I$.

Complexity Let $c_X \approx \lceil \frac{x+S}{128} \rceil$ the cost for computing one Array X element hash at Step 5b or in the Merkle tree, and c_F the cost of calling F_I once. The computation complexity in hash-block calls is about $(c_F + c_X + 1) \cdot T + 2^d \cdot \left((c_X + \frac{S}{128}) \cdot L + 1 \right) \propto 2^d L$ (nearly). The memory access complexity on Array X is about $3T + 2^d L$. With $S = 16$, $x = 1024$, $c_F = 11$, $c_X = 9$, $T = 2^{21}$, $L = 70$ we have about $2^{25.4} + 2^{d+9.3}$. With $S = 64$, $x = 64$, $c_F = 1$, $c_X = 1$, $T = 2^{25}$, $L = 84$: we have about $2^{26.6} + 2^{d+7}$.

Length L In first analysis, choose $L = 3.3 \cdot \log_2 T$ by applying the formula in [10], and only lower this length with clear cryptographic arguments. Note that it may make sense to look for such arguments because of the implied PoW size cost induced by a larger L.

Sizes T **and** x Changing Array X number of elements for a constant overall size could have interesting effects. Consider doubling $T' = 2T$ while symmetrically halving the element size. With the proposed link between the number of proofs and the array number of elements, and assuming that the ASIC resistance of the scheme stays the same, we have new Length $L' \propto \log_2 T' \propto 1 + \log_2 T \propto L + 1$, which reduces significantly the MTP proof size which is essentially the contents of $2L$ elements. However to keep the constraint that most array elements are accessed in a search and that the initial phase is small compared to the search itself, in array accesses $2T' \ll 2^{d'} L'$, that is $4T \ll 2^{d'}(L+1)$ which constraints the parameters harder. Also, the computation costs of hash operations is reduced on smaller elements as it is proportional to data sizes. Choosing the best balance depends on implementation parameters (desired size, implementation speed, desired proof effort...) and building an updated compression function for the new size. A limit case could be $T = 2^{25}$ with 64 bytes array elements, where BLAKE2 could be used directly as the element compression function, as discussed below. Switching from F_I to H has the benefit of using a standard hash function, but looses the detailed tweaking of F to favor cpu-intensive operations such as large integer multiplications. Building 2 GiB Array X would then be slightly more expensive (16 hash calls per KiB instead of 11), Length L would be slightly larger (84 instead of 70), and the search constraint would lead to $d > 20$ which seems reasonable. The resulting proof size could be reduced under 50 KiB, most of which dedicated to *opening* (Merkle tree proof) hashes.

Function F_I A way to build F_I would be to use a P_I variant which would involve the challenge, say by updating G to G_I by adding some dependency on I. Another advantage would be to avoid $F'(0) = 0$ identity which tends to propagate. The limit case with element size $x = 64$ is to simply use the hash function as F, for instance with $F_I(B_1, B_2) = H_{64}(B_1, B_2 \oplus I)$. An additional benefit is that F is then a full blown cryptographic hash function, not a water down version.

Size M If the PoW size is an issue, and following [10], the Merkle-tree computation can use a smaller hash size without ampering the overall security. With a simple criterion that one inversion to build an array element should cost more than the whole PoW, for our preferred parameters this leads to $2^{8 \cdot M} \geq 2^{26.6} + 2^{d+7}$, which is roughly satisfied including a $2^6 = 64$ margin if $M = \lceil \frac{d+7+6}{8} \rceil$ and if the initial phase is negligeable as expected. For realistic $d \leq 70$ this gives $M \leq 11$, and for conservative $d \leq 100$, $M \leq 15$. Note that collisions on Φ do not constitute a replay, as computations both before and after depend on Challenge I.

Size S Again choose $S = 64$ which is the maximum for one invocation of BLAKE2. Step 6 makes one search state size at least $L \cdot S$ bytes and adds a significant memory requirement (a few KiB per nonce) for parallel searches. Note that this memory is not strictly necessary, as the Y_i values could still be (re)computed from available data, but at the price of L^2 recomputations and Array X accesses. As Y_i values are smaller than their dependencies, their is no reason not to keep them (in register or cache) in place of these and incur a recomputation cost.

4.3 MTP PoW Scheme Redesign: Itsuku

In this section, we propose a new design, named Itsuku, closely inspired by the MTP-Argon2 proposals, but based on a different memory-hardness security principle and allowing a large parallelism for building the array, allowing to lower the overall elapsed time cost on a parallel system. The key design principle we follow is that the best attack benefit (α, σ) should have $\alpha \geq \frac{1}{2} \Rightarrow \sigma \geq 64$, that is halving the memory should induce a 64 or more search cost multiplier.

The Array X building phase of Section 4.2 is replaced by a parallel generation which builds P independent sequences of length $\ell = \frac{T}{P}$.

1. Build challenge dependent memory $X_I[0 \dots T-1]$ as :
   (a) $X_I[p\ell \dots p\ell + n - 1] = H_{nx}(p||I)$ for $0 \leq p < P$
   (b) $X_I[p\ell + i] = F_x^I(X_I[p\ell + \phi_0(i)], \dots, X_I[p\ell + \phi_{n-1}(i)])$
   for $0 \leq p < P$, $n \leq i < \ell$ and assuming $\forall k, 0 \leq k < n, 0 \leq \phi_k(i) < i$

The last phase is also updated to return the element themselves when they have no predecessors. The choices involved, which determine whether the security constraint is met, are: the level of Parallelism P, the number of dependencies n, the precise structure of Function F_x^I depending on the number of dependencies, the ϕ_k indexing functions, including how biased they could be. The cost multiplier when halving memory is very sensitive to detailed changes: biaising more or less the ϕ functions results in longer pseudo-random walks for recomputing values; the number of dependencies induces – or not – a dense requirement on preceding elements, possibly triggering more recomputations; the pattern of discarded elements profoundly influences the cost. In order to evaluate this elusive multiplier we have relied on a numerical simulations and taken a large margin, whereas actually providing some proof would be desirable.

Array Size T and Element Size x are still assumed as $T = 2^{25}$ and $x = 64$ to keep a 2 GiB memory footprint. However, as discussed in Section 5.7, we advise to consider extending T to 2^{26} or 2^{27} to avoid efficient implementations allowed by improved technologies in the mid term.

Number of Dependencies n is a key parameter to trigger a dense recomputation when the array is only partially available. The initial scheme chose $n = 2$, which allows a small multiplier for $\alpha = \frac{1}{2}$. We investigate $n \geq 3$ in order to enlarge the multiplier in this case, so as to fulfill our security objective. The larger the number of parameters, the more array elements are included in the PoW when providing predecessors, thus the smaller the length parameter to achieve a given number of leaves.

Dinur-Nadler attack A side effect of added dependencies is that the Dinur-Nadler attack precomputations are much more costly, if at all possible. Its detailed evaluation depends on the actual index functions. For instance, data independent ϕ_3 suggested below makes it impossible to fall on a control block for all i. If we assume independent pseudo-random $\phi_{k,k>0}$ index functions, the probability that the precomputation is stopped at each stage in Equation 2 is $\left(1 - \frac{1}{t}\right)^{n-1}$, and the probability to be successful up to the final stage is $\frac{1}{t}^{(n-1)(t-2)}$, which makes the precomputation overly prohibitive at $\approx c_F \cdot T \cdot t^{(n-1)(t-2)-1}$. On the other hand, the search phase cost is reduced significantly as Length L is smaller for a comparable number of proofs. The smallest cost multiplier for $L = 28$, $n = 6$, $d = 70$ is 80.8 with $t = 6$. For conservative $d = 100$ and $n = 6$, we need a larger $L = 31$ to reach a 69.9 cost multiplier with $t = 7$.

Indexing Functions ϕ_k and Bias The multiplier simulations are very sensitive to the choice of indexing functions ϕ_k with $0 \leq k < n$. We have considered the following debatable mixture of data dependent and independent functions: $\phi_0(i) = i - 1$, $\phi_1(i) = \phi(i)$, $\phi_2(i) = \frac{\phi(i)}{2}$, $\phi_3(i) = \frac{i-1}{2}$ $\phi_4(i) = \frac{\phi(i)+i}{2}$ and $\phi_5(i) = \frac{3 \cdot \phi(i)}{4}$, where $\phi(i)$ is the quadratic-biased function defined in the Argon2 scheme, or possibly a cubic-biased version. Choosing $n = 2$ and $P = 1$ results in the Section 4.2 scheme.

Parallelism P Table 2 shows the average recomputation Cost c_R in hash calls for accessing array elements when every other element is available, depending on the array size and number of dependencies. The value is the average of 5 to 100 simulations, depending on the computation time. Using a

n \ $\log_2 \ell$	11	12	13	14	15	16	17	18
2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3	3.2	3.5	3.8	4.1	4.5	4.8	5.1	5.6
4	10.0	13.0	17.1	21.8	28.1	36.2	46.1	
5	28.9	43.8	65.3	98.4	146.3	222.0		
6	41.6	66.4	105.5	166.9	265.7			

Table 2: Half Array Element Access Cost c_R with Quadratic-biased ϕ

cubic-biased ϕ function typically enlarges these costs by another 1.5 factor. Other storage strategies we have tested, such as keeping the first half and second half, resulted in much larger access costs: we conjecture that the every other approach might the best possible one, or close to. The associated cost multiplier for the search algorithm is $\frac{1+(c_R+\frac{3}{2})\cdot L}{1+\frac{3}{2}\cdot L} \approx \frac{2\cdot c_R}{3}$ if $c_R \gg \frac{3}{2}$, so we are looking for $c_R > 96$. Small number of dependencies do not allow to reach the target cost level. Without surprise, the greater the number of dependencies or the array size, the higher the cost, hence the multiplier. Chosing an even (for F_x^I symmetry) but not too large (for reducing threshold effect on the number of leaves) number of dependencies $n = 6$ and keeping a significant level of parallelism $P = 2^{10}$ or 2^{11} leads to $\ell = 2^{15}$ or 2^{14}, and a seemingly comfortable 178 to 112 cost multiplier, allowing a significant margin for better storage strategies on the half size array. If the every other strategy is indeed the best possible one, $P = 2^{12}$ would give $\ell = 2^{13}$ and a large enough 71 cost multiplier. Note that this size seems too large for the verifier to check array element values from the challenge.

Length L minimal value is derived from the expected number of leaves for the Merkle tree opening $n \cdot L \geq 168$, which leads to $L \geq 28$. Assuming that only data dependent $\phi_{k,k>0}$ functions are chosen, the Dinur-Nadler attack for $d \leq 100$ suggested a slightly larger $L = 31$. This length is enough so that even with a minor $\alpha = \frac{1}{4}$ not computed at all array saving, the search cost would be roughly multiplied by $(1-\alpha)^{-L} > 3000$.

Hash Function F_x^I is chosen so as to limit the cost of computing Array X to one hash call per element, and to avoid simplifications in case the ϕ_k functions would collide for an index. The function depends both on I and p so that each challenge and parallel segment has its own unique computations. We suggest to rely on $+_{64}$ modular addition on the array element considered as a vector of 8 unsigned 8-bytes integers to combine array elements so that only one hash call is needed:

$$X_I[pt+i] = H_x(p \oplus \sum_{k=0}^{k<\frac{n}{2}} X_I[pt+\phi_{2k}(i)], I \oplus \sum_{k=0}^{k<\frac{n-1}{2}} X_I[pt+\phi_{2k+1}(i)])$$

Preferred Parameters for this variant are $T = 2^{25}$, $S = x = 64$, $P = 2^{10}$, $n = 6$, $L = 31$, $M = \lceil \frac{d+13}{8} \rceil$ and the above $\phi_{k,0\leq k<n}$ and F_x^I functions. If the choice of ϕ_k functions makes the Dinur-Nadler attack impossible, for instance by including several data independent dependencies, $L = 28$ would reduce the proof size.

5 High-End Hardware Implementations

Instead of considering abstract implementations and time-area ratios as in [7], this section discusses actual hardware designs for MTP scheme PoW solvers, and projects the possible result with improved technology. First, we discuss an hypothetical BLAKE2 core (Section 5.1) and the reduced options for storing 2 GiB Array X (Section 5.2). Then we take as a reference for possible future specialized harware the NVIDIA Volta V100 GPU (12 nm process, June 2017) [16]: 21.1 billion transistors $\approx 5,275$ MGE

accomodating on die 5,120 CUDA cores running at 727.5 MHz and about 45 MiB of SRAM, and off die 16 GiB DRAM with 1 TB/s memory bandwidth assumed either way. We postulate that the die area can be divided freely between computing cores or up to 628.8 MiB of 4T SRAM. Based on such capabilities, we discuss the design of a dedicated special purpose processor where many BLAKE2 cores share a common memory for Array X. This hardware are evaluated for: MTP-Argon2 scheme ($c_X = 9, L = 70$) (Section 5.3); same with $\alpha = \frac{1}{2}$ array compression (Section 5.4); Dinur-Nadler scheme ($c_F = 11$) (Section 5.5); same with transposed implementation (Section 5.6); then these various implementations and their possible evolution depending on technology changes are discussed (Section 5.7); finally Section 5.8 shows expected performance for Algorithm 4.2 ($c_F = c_X = 1, L = 84$) and Algorithm 4.3 ($c_F = c_X = 1, n = 6, L = 31, P = 2^{10}$).

5.1 BLAKE2 Core Hardware

In order to evaluate the area cost for a BLAKE2 core, we consider the VLSI implementations proposed in 2011 for BLAKE-64 [15]. A high performance 8G (compression function) hardware is evaluated to 128 kGE and 15 cycles running at 298 MHz to process to process a 128 bytes input block (Table III in [15]), that is 2.37 GiB/s. Taking this as a reference, and considering that BLAKE2 differs from BLAKE-64 with less rounds (12 instead of 16), lower memory requirement and operation count per round, we will assume a BLAKE2 hardware implementation with 100 kGE can run in 10 cycles at 300 MHz, which will be considered as 1 tick, *i.e.* a tick is one input block hashing and run at 30 MHz. We consider this as representative of the time-area performance of the BLAKE2 hash function, with possible smaller area leading to more cycles and vice-versa.

5.2 Hardware Memory

We assume that the SRAM (registers, caches) implementation on die requires only 4 transistors per bit (4T SRAM), which is on the optimistic side. Storing 2 GiB of data would thus imply 17.2 GGE on die or 68.8 billion transistors, way beyond todays technology, although maybe not that far away: IBM foresees 30 billion transistor chips by 2019 [14]. As storing such an amount of data with very fast accesses on die is not a realistic option for now, we will assume that any significant memory is stored externally in DRAM, which raises the question of memory latency and bandwidth. However, search states are significantly smaller and must be immediately available to avoid delaying computations, thus would likely be stored in on die SRAM memory.

5.3 MTP-Argon2 Scheme Hardware

Let us consider a high-end fully-unrolled pipelined implementation of the initial MTP-Argon2 PoW scheme, which could produce one candidate Ω at each tick: it requires $1 + c_X \cdot L = 1 + 9 \cdot 70 = 631$ BLAKE2 hash cores, that is about 63.1 MGE. The GV100 die could thus accomodate up to 83 PoW solvers. Each solver would load $x \cdot L = 70$ KiB of memory per tick (30 MHz), thus consumming 2.15 TB/s of memory bandwidth. As the available bandwidth is only 1 TB/s, it constitutes the performance bottleneck, and most of the potential die area is not used. If the remaining area is converted to cache, its effectiveness for random accesses at about 30.3% of the target memory would help improve performance. Such a PoW solver would generate PoW candidates at about 21.5 MΩ/s.

This does not strike as an ideal design, as only 1% of the die is used for hash cores and the remainder for a large cache. Also, it does not consider the typically 80-100 cycles memory latency [20] to access DRAM for Array X. Such latency could be masked with about 8-10 threads, which would require to replicate the small search state accordingly. The 2 GiB array is not a significant burden in itself, as it is shared somehow by hundreds of mostly active BLAKE2 hash cores. The real limiting factor is that the memory bandwidth can really accomodate about 300 (without on die cache) or 427 (with large on die cache) BLAKE2 cores, although the die could host up to 52,750 of them. The area cannot be used efficiently for computing.

5.4 Array Compression Hardware

Let us consider $\alpha = \frac{1}{2}$ Array X compression. As noted in Section 3.2, on average one F call and 2.5 elements are required per accesses to X. As the memory bandwidth was already the bottleneck of the previous design, this approach will necessarily reduce performance, unless the compressed array can be kept mostly in cache. In order to sustain the same throughput as the previous implementation each PoW core would require $c_F \cdot L = 770$ additional BLAKE2 cores, the memory requirement is divided by two but the memory bandwidth requirement is 2.5 higher. We are neglecting that this design would also require implementating index Function ϕ which implies a integer modulo operator, adding some complexity and latency to the computation. Our die would accomodate up to 37 PoW solvers requiring each 4.89 TB/s, but the available bandwidth would be saturated for about 0.205 solvers. Luckily, converting the remainder area into cache can help improve throughput significantly, especially as it represents a significant part of the 1 GiB compressed memory. This overall architecture would produce 13.9 MΩ/s, about half the performance of the preceding implementation. Similarly to the preceding case, only a small part of the die is dedicated to computing.

5.5 Dinur-Nadler Attack Hardware

Let us now consider the Dinur-Nadler Attack. As noted on Table 1a, if the precomputation cost is fully neglected, which is not realistic, the computation cost is multiplied by at least 68.4, thus 52,750 on die BLAKE2 cores would produce at most $\frac{52750}{631 \cdot 68.4} = 1.22\ \Omega$ per tick. We neglected the index access function which would be also needed. A more realistic although still quite optimistic evaluation, which amortize the precomputation on one $d = 70$ proof produced every 10 minutes over 10 years (2^{19} proofs), yields a 116.2 multiplier for $t = 21$, leading to 0.72 Ω per tick. The memory requirement being much lower, we assume it could be stored on die in cache, thus memory (DRAM) bandwidth would not be an issue. The overall architecture would produce about 36.7 MΩ/s (68.4 multiplier limit) or 21.6 MΩ/s (116.2 multiplier). This design improves throughput by 0.5% (quite optimistic) to 71% (unrealistic) over the standard implementation. Although most of the area is dedicated to computing, the throughput improvement is limited because of the large cost multipliers. On these settings, the Dinur-Nadler attack brings few benefits in practice, even with favorable assumptions.

5.6 Transposed Search Hardware

We now investigate the transposed search algorithm outlined in Section 3.5. Its main benefit is to reduce the bandwidth requirement which is the bottleneck of both the standard and compressed implementations. As noted before, the search state size is 28 bytes, to which we add 4 bytes to manage a data structure which would allow to order searches per next array element to process, using some simple array and linked-list scheme. We will dedicate area for 576 MiB of search states on die, accomodating on average 9 search states for each 2^{21} elements, leaving 8.5% of die area available for up to 4,429 BLAKE2 cores.

When sending one array element, 9 search states on average can be incremented using 9 BLAKE2 operations each, thus with available cores we can handle a throughput of about $\frac{4429}{9 \cdot 9} \approx 54.7$ elements per tick, which translate to 1.53 TB/s bandwith to fetch them. This is yet again above the bandwidth limit, but for a much larger number of cores, which are thus active at about 65.5%. The overall throughput is about 137.8 MΩ/s, as BLAKE2 cores are quite actives and the memory bandwidth, although still the bottleneck, is much less so. Throughput could be improved a little by allocating remainder transistors to caching memory. The critical bandwidth efficiency is improved as 9 search states share loading one array element. Moreover, as the memory access pattern is known in advance, data can be prefetched and deep threading is not needed.

This design allows to feed thousands of hash cores with the available bandwidth. Although it is much more efficient than previous implementations, most of the area is still dedicated to cache, and under 10% is really used for computing.

5.7 Comparison of MTP-Argon2 Scheme Hardware Implementations

The throughputs achieved by the preceding hardware implementations of MTP-Argon2 PoW solvers are: 21.5 MΩ/s for the standard implementation, 13.9 MΩ/s for the $\alpha = \frac{1}{2}$ compressed implementation, 21.6–36.7 MΩ/s for the (quite optimistic to unrealistic) Dinur-Nadler attack implementation and 137.8 MΩ/s for the transposed implementation. The Dinur-Nadler attack is not really practical in this setting, given the very large precomputation cost needed to achieve limited benefit. For other implementations, hundreds to thousands of active BLAKE2 cores share a common memory, amortizing the burden of Array X size. Memory bandwidth is the key limiting factor, which gives a significant advantage to the transposed implementation as loading an array element is shared by several hash computations. However, these designs do not look very efficient as only 1-10% of the area is dedicated to hash computations, and the remainder for cache.

These evaluations are sensitive to hypotheses which may not be fulfilled when technology evolves: whether the array fit on die, the available transistors and the memory bandwidth constraint.

On Die Array As soon as the array can fit on die, any BLAKE2 core that can be crammed next to it could be fed most of the time (well, probably not without any limit, that we ignore to simplify our argument), improving the throughput. An hypothetical 100 billion transistor chip by 2026, about 5 times larger than the area considered by the previous design, could use about $\frac{2}{3}$ of its area for 2 GiB SRAM memory, leaving area for 300,000 BLAKE2 cores generating 14.2 GΩ/s, thus providing a handsome 100-fold throughput improvement. Even before this event, a smaller 50 billion transistor chip would bring a performance milestone by hosting an $\alpha = \frac{1}{2}$ compressed array on die, leaving area for 150,000 BLAKE2 cores generating 3.2 GΩ/s, thus providing an honest 23 speedup. To avert such implementations, a memory-hard PoW scheme design must ensure that the array, or even half of it for these settings, cannot fit on a die. As available die area grows with time, the array size should scale accordingly. Starting with a 4 or 8 GiB array could be both compatible with todays CPU and GPU hardwares and provide a larger margin against future on die storage.

More Transistors In the mean time, assuming a constant memory bandwith, transistor count improvements add more cache and cores. The transposed implementation scales roughly linearly, feeding 9 BLAKE2 cores per 64 MiB search state cache increments for each element transfered. A 30 billion transistor chip by 2019 would allow for up to 894 MiB of on die SRAM, fitting 13 states per array element (832 MiB) instead of 9 and up to 5,206 BLAKE2 cores on the remaining area. At the maximum memory bandwidth, 976 M elements can be loaded per second, feeding ($\times$13) 12.7 G memory states per second requiring ($\times$9) 114.3 G BLAKE2 hash computations per second, which needs 3,809 BLAKE2 cores running at 30 MHz. The throughput would reach 181.1 MΩ/s, a 31.4% improvement compared to our 21.1 billion transistor chip.

Memory Bandwidth is the bottleneck of all our hardware designs but the Dinur-Nadler attack implementation. Memory latency was assumed to be maskable by appropriate threading. Although cores, cache and memory improve directly from better integration, the limiting factor of memory bandwidth is less obvious, as it is mostly about fitting more memory bus lanes and handling the flow of requests. If we assume that some novel technology could break the current limits, or that a specially designed PoW solver could provide significantly better bandwidth, then the next in line bottleneck is the number of cores. Under CPU-bound conditions, the standard implementation would run at 2,508 MΩ/s, the compressed implementation at 1,129.6 MΩ/s, the Dinur-Nadler implementation would still optimistically run at 21.6–36.7 MΩ/s and the transposed implementation would raise to 210.6 MΩ/s. At such performance level, the memory size, shared between dozen thousands of cores, is not a real issue.

The MTP-Argon2 hardware implementations show threshold effects depending on the origin of the performance bottleneck. A memory bandwith bottleneck with a small search state makes the transposed approach the most effective. As available on die memory rises, the compressed implementation

where half the array fits on die becomes the leader, followed by the standard implementation once the full array can be hosted. The Dinur-Nadler implementation would only come ahead by lowering the available memory bandwidth. Performance results are similar with our proposals as shown in Section 5.8, although the transposed approach cannot be implemented and the margin against the Dinur-Nadler attack is larger.

5.8 Algorithm 4.2 and 4.3 Implementations

We now discuss the implementation of the algorithm variant presented in Section 4.2 and 4.3, with $x = S = 64$ and a back sweep. We do not take into account Array X challenge-dependent initialization, which is somehow assumed as negligeable compared to the search cost.

For Algorithm 4.2, one pipelined PoW solver ($L = 84$) would require $1 + \frac{3}{2} \cdot L = 127$ BLAKE2 cores and consume $x \cdot L = 5{,}376$ bytes per tick (161.28 GB/s) of memory bandwidth. As before, the DRAM access latency should be masked by using perhaps $\theta = 9$ threads. However with this scheme the search state memory is significantly higher, as each PoW solver requires at least $S \cdot (\frac{L}{2} + 1) \cdot L = 225.75$ KiB per thread (computed with hash size and average number of live values). From that state, only the current 64-bytes hash for each L first hash cores really needs to remain on die for most of the computation to mask the L concurrent accesses, the reminder $\frac{L}{2}$ hashes on average can be either kept on die or could be exported to RAM and brought back when the final back sweep must be computed. However, this would consume more bandwidth which is already the performance bottleneck. Thus the full state of 9 threads should be kept on die, requiring 2 MiB per solver, which is larger than the PoW solver itself. The throughput computations take into account that the significant remaining area is converted to cache, thus lowers the bandwidth requirement. The best cost multiplier of the Dinur-Nadler attack including pre-computation cost is 98.0 for conservative Strength $d = 100$. This is not enough to make the scheme advantageous compared to the standard implementation.

A transposed implementation is much more problematic than the previous case because of the large $S \cdot (\frac{L}{2} + 1) + 12 = 2764$ bytes average search state size and the larger number 2^{25} of array elements, which would imply 86.4 GiB for just one search state per element. Even if only one current hash is kept and other state hashes could be offloaded to DRAM, the search state is still $S + 12 = 76$ bytes per array element, inducing a still unpractical 2.375 GiB for one search state per element: As the search state is larger than one element, the transposed search which provided the best throughput on the MTP-Argon2 PoW standard scheme cannot yield any benefit. Assuming $S = x$ kills the possibility of a transposed search.

Scheme	cores per solver	Max PoW solvers die area	Max PoW solvers mem bw	Ω/tick	MΩ/s
Algorithm 4.2 + full on die thread cache	*127*	179.6	6.2	8.9	267.7
same + partial on die thread cache	*127*	402.4	3.1	4.5	134.1
$\alpha = \frac{1}{2}$ compression and mem cache	*211*	250.0	2.5	6.4	191.6
Dinur-Nadler Attack (×98.0)	*12446*	4.2	–	4.2	127.1

Table 3: Algorithm 4.2 implementations, including cache effects

Table 3 summarizes the various implementations for Section 4.2 PoW variant. The standard implementation is the best one by a large margin, even assuming a conservative multiplier for the Dinur-Nadler attack. Similarly to the previous case, all implementations but the Dinur-Nadler attack are bounded by memory bandwidth, and most die area is used for caching rather than for computation.

Table 4 shows evaluations for Section 4.3 variant. The $\alpha = \frac{1}{2}$ Array compression is prohibitive with $n = 6$ because of the computation cost and the added bandwidth requirement.

Scheme	cores per solver	Max PoW solvers die area	Max PoW solvers mem bw	Ω/tick	MΩ/s
Algorithm 4.3 + full on die thread cache	*48*	1098.9	16.8	24.0	720.0
Dinur-Nadler Attack (×69.9)	*3355*	15.7	–	15.7	471.6

Table 4: Algorithm 4.3 implementations, including cache effects

6 Memory-Hard Crypto-Currency PoW Schemes

As noted in the introduction, crypto-currencies such as Bitcoin rely on a PoW function to randomly share the reward for the maintainance of the distributed ledger, a.k.a. blockchain. Relying on a simple computation-bound hash function has centralized the bitcoin mining market around a select group of miners who can afford the specialized mining hardware. A consequence of mining power concentration is that it can be open to cheating strategies which allow a participant to receive more rewards than their mining power should entitle them, and their relative efficiency makes competition unprofitable. Thus PoW schemes impervious to FPGA or ASIC implementations are sought, focusing in particular on memory-bound approaches. Memory bound has been defined in term of latency, bandwidth or size. The standard approach is to combine hashing and memory access in some pseudo-random way. We discuss here two particular schemes: the CryptoNight [18] and Wild Keccak [8] hash functions.

6.1 CryptoNight

The CryptoNight hash function [18] proposed for CryptoNote [19] builds a scratchpad of pseudo-random data with numerous read/write operations and hashes the result to get the final value. The first phase applies Keccak [5] (aka SHA-3) to build some pseudo-random data, which are then encrypted with repeated AES simple round iterations using various keys to fill a 2 MiB (2^{21} bytes) scratchpad. The second phase updates the scratchpad by iterating 2^{19} AES rounds using a pseudo-random walk on the scratchpad seen as 2^{17} blocks of 16 bytes, thus overwriting each block 4 times on average. The third and final phase uses XOR, AES and some part of Keccak to sweep over the scratchpad and compute a hash of the whole thing, ending with a pseudo-randomly chosen hash function among BLAKE, Groestl, JH, Skein. The overall structure is similar to the Argon2 password hashing scheme.

The provided specification lacks any discussion for the choices of various parameters (*e.g.* why 2^{19} iterations... should it rather be 2^{18} or 2^{20}?) and a justification for the apparent over-complication: no full AES is used, the Keccak permutation is used at some point, then other unrelated hash functions are invoked. No doubt the authors had some idea in mind, that they should have shared with the reader. Using only partial or modified cryptographic algorithms means that the security properties expected and studied for the full designs cannot be ensured, thus the whole security should be re-analysed very carefully. No clear complexity/cost analysis is provided, in particular an analysis of potential performance bottlenecks would be welcome. On the whole, it is plausible that the 2 MiB scratchpad is really needed to compute the final value, without shortcut. However, the same property could probably have been obtained with a simpler and more argumented approach.

From a crypto-currency perspective, we think that the approach is not ideal: The CryptoNight is certainly an expensive hash function (maybe equivalent to 2^{18} full AES calls?) which requires 2 MiB. However, verifying the results require the same amount of computation and memory, which we believe are both too expensive on this side: A memory-hard PoW scheme should not require the same memory constraint for the verifier. Moreover, even if 2 MiB is very expensive for ASIC, it is still doable, especially as technology improves: the step is higher for a miner, but once achieved benefits are ripe nevertheless.

6.2 Wild Keccak

The Wild Keccak hash function [8] has been proposed for the Boolberry project. It aims at reducing the memory-hardness on the verifier side while keeping a high requirement for the miner. For this purpose, a global scratchpad is built with data coming from the blockchain itself, and is then extended as the chain moves forward, targetting a 90 MiB per year growth. The hash function itself is a modification of Keccak [5] where internal state update operations are modified and intertwined with memory dependencies accessing the currency state. We agree with the authors who state: *It is debatable if this modification will keep all cryptographic properties of hash function...* such changes should be discussed in depth. Each hash computation involves 1100 accesses to 32-bytes block of scratchpad. The paper lacks an analysis of how much of the stratchpad would be used in a typical search and more generally an analysis of performance bottlenecks. A proof-of-concept implementation shows a $25 - 45$ μs per hash computation, depending on the scratchpad size.

In the crypto-currency context, we agree that relying on the blockchain data is potentially a good idea to build a relevant scratchpad, provided that data is pseudo-random. Another benefit is that it paves the way to a future growth of the requirement, although only a linear one. However, this also means that the verifier needs this data: the verification cannot be performed using the block itself and the light weight verifier property is only partially obtained.

6.3 PoW Schemes Comparison

We compare CryptoNight, Wild Keccak, MTP initial and present proposals.

All proposed schemes but the present MTP variants rely on weakened cryptographic primitives: CryptoNight uses AES simple rounds and the Keccak permutation, Wild Keccak modifies the internal operations and mixes them with other data, the MTP initial scheme devises a special F block combining function based on a simplified version of the BLAKE2 compression function, stripped of constants and with different operators. Weakening the cryptographic primitives without a clear analysis of the consequences does not help building trust, thus we think that such design choices should be avoided. If deemed necessary, accessing memory and using expensive operators should be performed out of the primitives so that their security properties are unaltered. More generally, CryptoNight and Wild Keccak are really specifications which lack precise and extensive cryptographic arguments and justifications. Although this does not mean that they are weak, providing such discussions and chosing genuine primitives that keep their native security properties would help build confidence in these schemes.

The PoW function computation cost, essential for verification, is very large for CryptoNight, say hundreds of thousands of calls, just one call for Wild Keccak, and a few dozens to a few hundreds for the MTP variants. We think that this number should be kept reasonably low. CryptoNight and Wild Keccak would benefit from a precise performance bottleneck analysis, taking into account potential hardware implementation, which we have provided for MTP variants.

The underlying PoW properties, inherited from key design choices, are significantly different. The verifier needs to store or rebuild the memory scratchpad with both CryptoNight and Wild Keccak, enduring significant memory costs, while the MTP variants design rely on the Merkle-tree proof to only convey part of the array, which is thus not fully needed for verification. As a consequence of this property, the proof is significantly larger with MTP variants, typically dozens of kilobytes, compared to only a few bytes for the former. Moreover, the array building cost must be amortized on a significant number of searches so as to be negligeable and have a near *progress free* scheme.

For a memory-hard scheme, a key overall design option is the memory size requirement. It is remarkable in itself that the different proposals vary so widely on that point: CryptoNight deems 2 MiB as enough, Wild Keccak starts with about 100 MiB, and the initial MTP scheme variant requires 2 GiB. We think that even more is needed to provide enduring resistance to the scheme.

7 Conclusion

This report brings new memory-hard PoW proposals based on [7] and which attempt to counter known attacks [11, 6] on the scheme. Key contributions, which are steps toward a better memory-hard MTP Argon2-based PoW scheme, include:

- to consider starting from a larger than 2 GiB array and include a way to increment its size as hardware capabilities evolve.
- to use a constant Array X if possible, allowing a small PoW as the whole MTP part is avoided.
- otherwise, a criterion to choose L depending on T.
- to make compression Function F dependent on Challenge I through F_I, with a limit case where $F_I = H^I$.

  Even if the Dinur-Nadler attack is not practical, such dependency ensures that any pre-computation-based attack would have to be specific to the challenge, which is a good property.
- to make Merkle tree computation dependent on Challenge I through H^I.
- to enlarge the search state size per nonce, and a way to do so with a back sweep hashing on intermediate search hashes.
- to consider using a larger number of smaller elements for Array X, thus reducing the proof size and making a transposed search uninteresting once $x = S$.
- a variant built around a clear security criteria: that any implementation which saves half the memory or more should endure a 64-fold computation cost up to conservative PoW Strength $d \leq 100$.
- which allows to build Array X in parallel, making the scheme more *progress free*.
- a criterion to reduce the hash size for the Merkel Tree depending on d, which helps reduce the proof size.
- particular instantiations:
  - $T = 2^{25}, x = S = 64, F_I = H^I, n = 2, L = 84, P = 1, M = \lceil \frac{d+7+6}{8} \rceil$
  - and the variant ... $P = 2^{10}, n = 6, L = 31$
- a coarse hardware evaluation loosely based on current high-end GPU technology of the various algorithms and attacks, which shares Array X among many (100's to 1000's) BLAKE2 cores, and suggests that the performance bottleneck is memory bandwidth rather than size.
- a refinement of the evaluation of the Dinur-Nadler attack costs.

Following [7], we recommand that an ASIC-expensive hash function, involving costly operators available in general purpose CPU and GPU such as large width multiplication or division, should be designed. From this perspective BLAKE2 could probably be improved upon, as the design criteria and selection process for hash functions are usually in the opposite direction, trying to minimize the hardware footprint by using simple logical operators. Such a function combined to the above PoW design would help achieve a better specialized hardware resistant scheme. A simple way to build such a function is to take an existing efficient hash function and add a xor layer which uses these operators, *e.g.* with $h(s, x_{0...15})$ the compression function which updates 64-byte internal State s with 128-byte input x, build $y_{0...7}$ as $y_i = (x_i \cdot x_{2i} + x_{i+1}) \bmod \left(x_{2i+1} | 2^{9i}\right)$ and then perform $h'(s, x) = h(s, x) \oplus y$, adding 32 integer arithmetic and 16 logical operations on 64-bit integers.

This report is argumentative and qualitative in nature and lacks proven justifications for some aspect of the proposals. In particular, we relied on numerical evaluations for some costs because they depend on biased pseudo-random functions. The Python implementations used may have bugs that could change the resulting figures significantly. Moreover, as the simulations were quite slow, the number of iterations was not as high as required for good precision. The source code for these scripts, used for Tables 1a, 1b and 2, are available on request.

PoW functions are an ecological hazard: avoid them if you can.

Thanks

We would like to thank Itai Dinur, Niv Nadler, Alex Biryukov and Dmitry Khovratovich for discussions and feedback, Bahamut for proofreading and Doubloon Skunkworks for support.

References

[1] Martín Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. Moderately Hard, Memory-bound Functions. In *10th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2003.

[2] Martín Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. Moderately Hard, Memory-Bound Functions. *ACM Trans. Inter. Tech.*, 5(2):299–327, 2005. A previous version appeared in NDSS'2003.

[3] Biryukov Alex, Daniel Dinu, and Dmitry Khovratovich. Argon2: the memory-hard function for password hashing and other applications. Technical report, University of Luxembourg, Luxembourg, March 2017. Version 1.3, `https://www.cryptolux.org/images/0/0d/Argon2.pdf`.

[4] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5, January 2013. Version 2013.01.29, `https://blake2.net/blake2.pdf`.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak SHA-3 Submission (Version 3), January 2011.

[6] Marc Bevand. Attacks on Merkle Tree Proof, August 2017. `http://blog.zorinaq.com/attacks-on-mtp/`.

[7] Alex Biryukov and Dmitry Khovratovich. Egalitarian Computing. In *25th USENIX Security Symposium*, pages 315–326, Austin, Texas, USA, August 2016.

[8] Boolberry Team. Block Chain Based Proof-of-Work Hash and Wild Keccak as a Reference Implementation. `http://boolberry.com/files/Block_Chain_Based_Proof_of_Work.pdf`, August 2014.

[9] Fabien Coelho. Exponential memory-bound functions for proof of work protocols. Research Report A-370, CRI, École des mines de Paris, September 2005. Also Cryptology ePrint Archive, Report 2005/356.

[10] Fabien Coelho. An (Almost) Constant-Effort Solution-Verification Proof-of-Work Protocol Based on Merkle Trees. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, number 5023 in LNCS, pages 80–93. Springer, June 2008. Extended version available as Cryptology ePrint Archive Report IACR 2007/433.

[11] Itai Dinur and Niv Nadler. Time-Memory Tradeoff Attacks on the MTP Proof-of-Work Scheme. IACR 497, Ben-Gurion University, Israel, May 2017.

[12] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 426–444. Springer, 2003.

[13] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology—CRYPTO '92*, pages 139–147. Springer, 1992.

[14] Pouya Hashemi and T. B. Hook. CMOS Device Technology Enablers and Challenges for 5nm. In *Symposium on VLSI*, Tokyo, Japan, June 2017.

[15] Luca Henzen, Jean-Philippe Aumasson, and Raphael C. W. Phan. VLSI Characterization of the Cryptographic Hash Function BLAKE. *IEEE Transactions on Very Large Scale Integration Systems*, 19(10):1746–1754, 2011.

[16] NVIDIA. NVIDIA Tesla V100 GPU Architecture Whitepaper 1.0, 2017. `https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf`.

[17] Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. In *BSDCan 2009*, Ottawa, Canada, May 2009. `https://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf`.

[18] Seigen, Max Jameson, Tuomo Nieminen, Neocortex, and Antonio M. Juarez. CryptoNight Hash Function. CryptoNote Standard 008, March 2013. `https://cryptonote.org/cns/cns008.txt`.

[19] Nicolas van Saberhagen. CryptoNote v 2.0. White paper, October 2013.

[20] William. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

Typeset with LaTeX, document revision 1854