Itsuku: a Memory-Hardened Proof-of-Work Scheme

Fabien Coelho and Arnaud Larroche and Baptiste Colin
firstname.lastname@mines-paristech.fr
MINES ParisTech, PSL Research University

Version 1.51 on November 29, 2017

Abstract

Proof-of-Work (PoW) schemes allow to limit access to resources or to share rewards for crypto-currency mining. The MTP-Argon2 PoW by Biryukov and Khovratovich is loosely based on the Argon2 memory-hard password hashing function. Several attacks have been published. We introduce a new transposed parallel implementation attack which achieves higher performance by circumventing apparent bandwidth requirements. We then present Itsuku, a new scheme that fixes known issues by changing MTP-Argon2 parameters and adds new operations to improve memory hardness. Our scheme is built on a simple security criterion: any implementation which requires half the memory or less should induce at least a $\times$ 64 computation cost for difficulty $d \leq 100$. The Itsuku proof size is typically $1/16$th of the initial scheme, while providing better memory hardness. We also describe high-end hardware designs for MTP-Argon2 and Itsuku.

Contents

1 Introduction

Proof-of-Work (PoW) functions [16] must 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 [3, 4, 15, 11, 19] or possibly other criteria. Memory-hardness claims have been made about latency, bandwidth or amount of memory required for a task. In the context of crypto-currencies, the solution-verification PoW scheme is an essential building block that allows to distribute the rewards between miners that maintain the distributed ledger, a.k.a. 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 considered a good deterrent.

We first describe and comment the initial MTP-Argon2 PoW scheme (Section 2). It shows a high memory requirement for the solver, but a much lower one for the verifier. This differ fundamentaly from memory-hard password hash schemes such as Argon2 [5] which require the same amount of memory on both sides. We then discuss in depth the various known attacks and present new ones (Section 3). This leads us to propose the Itsuku PoW, which fixes all known attacks (Section 4). The expected software and hardware performance of MTP-Argon2 and Itsuku implementations are discussed (Section 5). We then compare different memory-bound schemes (Section 6), before concluding (Section 7).

2 The MTP-Argon2 PoW

The *Egalitarian Computing* paper [9] presents Proof-of-Work schemes loosely based on the Argon2 [5] memory-hard password hashing function. It includes instantiation settings for crypto-currency applications, time-lock puzzles and disk encryption. Although the Argon2 function could be used directly for such purposes, its very high single-execution computation and memory cost is prohibitive for the verifier: Schemes displaying smaller verifier cost but yet requiring large memory are sought. We focus on the PoW scheme introduced in the paper and the proposed crypto-currency instantiation.

The PoW construction builds a Merkle tree over an array and pseudo-randomly selects a subset of the leaves based on the root hash of the tree as proof of computation, similarly to [12]: The feedback loop means that changing inconvenient leaves requires recomputing the root hash, thus would change the leaf selection. If enough leaves are provided, it ensures that most of the Merkle tree has been computed. However, unlike [12], the array is quite large and the PoW does not end at the Merkle tree, which is expected to be a small part of the PoW. It adds a classical iterative partial hash inversion search on top of the construction, which also depends on the contents of the large array. The idea is to enforce some memory-hardness property by showing that the array was mostly computed, stored and used in a large iterative search. 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 not directly related to the PoW scheme itself and it does not actually use the Argon2 function.

The following notations are taken from [9] and re-used:

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

T, number (power of 2?) of array elements of size x.

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

d, difficulty of the PoW, *i.e.* number of expected binary zeros at the beginning or end of the final hash value. In practice, a target limit value is rather used (*i.e.* final hash $\Omega < t \approx 2^{S-d}$) so that difficulty changes are continuous: d must not be assumed strictly as an integer.

$H_s(x)$, a variable-size (s bytes) cryptographic hash function. The paper uses BLAKE2 [6] ($1 \leq s \leq 64$) with an extension to larger sizes that costs about one call every 32 bytes (Section 3.2 of [5]). For

complexity purposes with BLAKE2, we count a unit of cost per 128-byte block of input data, which encompasses 12 calls to the underlying compression function G.

$x' = P(x)$, Permutation P. It performs 8 calls to compression function G loosely based on BLAKE2 internal round function (Annexe A of [5]). It takes 128 bytes at a time and uses 64-bit add ($+_{64}$) and xor ($\oplus_{64}$), 32-bit input 64-bit result multiplication ($*_{32}$ – deemed expensive on ASIC), circular 64-bit shift ($\gg_x$) and left 64-bit shift (a.k.a. multiply by 2) operations.

$B' = F(B_0, B_1)$, a compression function taken from Argon2 that takes two 1 KiB inputs, produces one 1 KiB output, and which relies on 16 calls to Permutation P internally (Section 3.4 of [5]). Total computation cost is about $16 \cdot 8 = 128$ compression function calls, or about $c_F = \frac{128}{12} \approx 11$ BLAKE2 128-byte processing calls.

$\phi(i)$, a contextual indexing function that depends 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 has a quadratic bias, so that array accesses are less unbalanced in the initial array-building sweep. Indexing function ϕ relies on the previous element and implies an integer modulo operation to select an element among the already computed ones, *i.e.*, we have: $\phi(i) = \phi_d(X[i-1]) \bmod (i-1)$ (where ϕ_d is a data-dependent function).

We describe MTP-Argon2 with a few changes from the original paper so that the algorithm is well defined; some redundancy is removed, the array and variable indexing starts from 0 following usual conventions, and some implicit choices are made explicit.

From I, L and d, we compute:

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

Search State Size Within the search loop, the search state is the current values of nonce N (say 11 bytes), iteration j (1 byte) and current hash Y (16 bytes), that is about 28 bytes. This is quite small and allows efficient parallel searches as discussed in Section 3.5.

Complexity The computation complexity in hash-block calls is $\frac{2x}{32} + c_F\,(T-2) + (c_X+1)\,T - 1 + (c_X\,L+1)\,2^d \approx (c_F + c_X + 1)\,T + c_X\,L\,2^d \propto \frac{c_F+c_X+1}{c_X}\,T + L\,2^d \approx 2.3\,T + L\,2^d$. The first part is the array construction cost, then the Merkle tree with first the hashing of leaves followed by the binary tree hashing itself, and finally the search loop on the nonces. On average during a search there are about $3\,T + L\,2^d$ memory accesses, which is comparable to the computation complexity.

Partial Dependency on Challenge The Merkle tree proof part of the PoW in [9] is expected to show that most selected array elements are such that $X[i] = F(X[i-1], X[\phi(i)])$ holds. This is a local property which does not depend on the challenge; thus it does not follow that these elements were computed from the provided challenge, nor that they were stored in memory instead of being possibly recomputed when doing the iterative search. Reusing a previous instance can work, as taken advantage of by the Dinur-Nadler attack discussed in Section 3.4. This issue can be addressed by using the challenge in hashes at every step of the computation, both when building the array and in the search loop.

Merkle Tree Openings Phase 7 provides the separate openings of the antecedents of the selected array elements, but does not include the selected elements themselves. One merged opening for all leaves can be proposed instead, which would reduce the number of nodes to provide, hence the overall proof size. Moreover, letting out the selected array elements allows the very efficient pseudo-random array attack (Section 4.1 of [14], described in Section 3.3). The selected array elements must also be included in the opening.

Crypto-currency Instantiation The instantiation for crypto-currency, called MTP-Argon2 in Section 4.5 of [9], uses the Argon2d (data-dependent) variant with 4 parallel lanes (discussed in Section 3.7), uses hash function BLAKE2 [6] for H, uses 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 $2^{22.6} + 2^{d+6.1}$ memory accesses on X. The choice of hash function BLAKE2 for a memory-hard PoW seems reasonable: it 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 appropriate.

Time-Area Proof A proof based on a simple abstract analytical model [9] suggests that, with the chosen parameters, ASIC implementations can be shrinked by a 1/12 factor at most. However, this proof has some drawbacks. This time-area cost analysis does not take into account efficiency properly. It focusses on area and time involved between computing cores and DRAM, whereas our hardware analysis in Section 5 shows that the performance bottleneck of a hardware implementation is memory bandwidth, which does not appear in the model (Equation (4) in [9]), although it is said that it could have a contribution. Moreover, our implementations show performance threshold effects on parallel implementations between slow off-die DRAM and faster on-die SRAM once the array can fit, that are not represented by the simple model. The proof is based on 64 KiB DRAM memory area being equivalent to one hash core, whereas our 4T SRAM implementations rather have 64 KiB equivalent to over five cores, allowing significantly more computation power by substituting. The proof seems to assume that a cheating test failure costs as much as a non cheating test failures ($(1-\varepsilon)^{-L}$ factor), which is not the case, as a cheating failure can be detected early when an unavailable block is required, as outlined by Equation (5) where 9 iterations are needed to provide over $\times 2^6$ security for a half array. Finally, the proof is not linked to the PoW structure and does not prevent the many attacks presented in Section 3. These drawbacks make us believe that a simple time-area analysis does not reflect the performance and cost tradeoffs of efficient MTP-Argon2 hardware implementations.

Choice of Length The proposed length L looks somehow arbitrary. It is partially justified by the time-area proof discussed previously. The final proof size and verification cost are proportional to L, so the lower the better. Other security criteria push towards a reasonably large L. First, it should be large enough so that computing only a fraction of X impairs the search algorithm significantly, but for that constraint a much smaller L could be chosen. Equation (2) in [12], for a related problem, provides a relative cost lower bound for a provable Merkle tree computation that depends on the required number of proofs $2L$, the number of leaves T and the fraction f of the array that has been computed, so that

$$L \geq \frac{-1}{2 \cdot \log_2 f} \cdot \log_2 T$$

Choosing $L = 70$ is consistent with this formula for $f = 0.9$. We discuss in Section 4 conditions under which a much smaller length L may be picked.

Constraints on Difficulty For the search to be as progress free as possible, that is to allow available computing power to contribute easily to mining, the array initialization phase must be negligeable, *i.e.*, $2.3\,T \ll L\,2^d$, which leads to $d > 16 \approx \log_2 2.3 + \log_2 T - \log_2 L$. A trivial constraint is that difficulty d is smaller than the hash size it targets, the limit of the partial inversion being a full inversion of the hash function, thus $d \leq 8 \cdot 16 = 128$. The actual choice for d between 17 and 128 depends on the implementation speed, available computation power, and desired frequency of finding a solution. For reference, as of Octobre 2017 bitcoin mining typically requires about 2^{72} relatively inexpensive SHA-256 operations for each block[1]. A working memory-hard scheme would be expected to be significantly harder. If we focus on computation only, BLAKE2 hardware efficiency is similar to SHA-256 [18], and one candidate hash with default settings requires 631 invocations, so the relative computation ratio over Bitcoin's SHA-256 is about $2^{9.3}$. Itsuku (Section 4) involves 15 hash calls ($2^{3.9}$ computation ratio). 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.

Proof Size The PoW is composed of $2L$ 1 KiB array elements and their associated per-leaf openings. With 16 byte hashes, the MTP-Argon2 PoW size is about $16 + L \cdot \lceil \frac{1}{8} \log_2 T \rceil + 2 \cdot L \cdot 1024 + 2 \cdot L \cdot \log_2 T \cdot 16$ bytes (first a rounded nonce, then selected leaf indexes, followed by their antecedents, and finally their openings). This leads to 186 KiB, three quarters of which are array elements. Such a size is a significant burden on the crypto-currency chain that we will aim at reducing.

Number of Antecedents The proposed scheme 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, or the array element could depend on more previous array elements, *i.e.*

$$X[i] = F(X[\phi_0(i)], X[\phi_1(i)], \ldots, X[\phi_{n-1}(i)]). \tag{1}$$

Adding dependencies increases the cost of some attacks, but it also enlarges the proof size unless L can be significantly reduced. We will investigate such option carefully.

3 Attacks on MTP-Argon2

We discuss various attacks on MTP-Argon2. We measure the benefit of an attack as a couple (f, σ) with f the needed memory as a fraction of the standard implementation memory and σ the associated computation cost as a multiplier of the standard implementation cost. This criterion allows to choose parameters so as to provide a uniform security margin. We propose that for a memory-hard PoW function, any scheme requiring half the memory or less $f \leq \frac{1}{2}$ should induce a significant $\sigma \geq 64$ cost multiplier for conservative difficulty $d \leq 100$. These values result in quite inefficient hardware attack implementations, as shown in Section 5.

3.1 Raw Recomputation Attack

A baseline attack that 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 $2^{263.1}$

[1]BTC block height 488101 hash [1] is `00000000000000000762eecdcc661d556425f254ef83c49280556c0ea4d9edf`

calls to F (per a numerical simulation). The attack benefit is about ($\approx 0.0, c_F \cdot 2^{263.1}$): no memory is needed, at the price of many computations.

3.2 Memory Saving Attack

Section 4.2 in [9] presents a *memory saving* attack and argues that the resulting space-time complexity is prohibitive for significant compression.

Let us consider $f = \frac{1}{2}$ compression of array X where one in two elements are discarded. 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 recomputation cost is it to apply F on the available previous 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: $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot \frac{3}{2} + \frac{1}{4} \cdot \frac{3}{2} + \ldots \approx 2$. For a limited recomputation cost, the memory requirement can be halved. This attack benefit is ($0.5, \approx \frac{c_X + c_F}{c_X} \approx 2.2$): half of memory is saved but accessing an array element costs c_F on average instead of 0, which impacts the overall search cost.

This cost increases very quickly when the memory saving is increased further (Section 5.1 in [5]). The resistance to memory compression can be improved by making F rely on more antecedents, as outlined in Equation (1).

3.3 Pseudo-Random Array Attack

This attack is described in Section 4.1 of [14]. As the openings provided by the MTP-Argon2 PoW do not include the selected array element themselves, they do not need to belong to the Merkle tree; hence the tree may not be based on F.

Consider building a pseudo-random array from some cheap PRNG, *e.g.* $X[i] = R(i)$, compute its Merkle tree root, then at Phase 5b build a fake element from regenerated antecedents $X'[i] = F(R(i-1), R(\phi(i)))$. The proof is accepted because antecedents do belong to the Merkle tree and the F property on the selected array elements holds. A suspicious verifier may detect such an attack if a selected array element is also an antecedent of some other selected array element. This check has a low $\frac{2L^2}{T}$ success probability and can be easily avoided by the prover. Also, if an odd array element is selected, then its even predecessor opening should include its hash, which can be checked by a verifier. If performed, the probability of this check to detect an issue is $(1 - \frac{1}{2}^L)$, which is very high if L is large. The efficiency of this attack is about ($0.0, \approx \frac{c_X + c_F}{c_X}$): no memory is needed, but one call to F is needed on each array accesses. This attack also works with a challenge-specific function F. The selected array elements and all their antecedents must be included in the collective opening, so the F property is shown to hold within the Merkle tree itself.

3.4 Dinur-Nadler Precomputation Attack

A detailed and theoretically convincing *cheating* attack is presented in [14], which is somehow a clever application of the *block modification* attack (Section 4.2 in [9]) 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 [9] can be avoided by precomputing a special array X which *mostly* respects F computations. The iterative construction is broken at regular points, which allow to reconstruct missing X values from a few kept values. The attack uses a parametric t compression so that only T/t regularly spaced *control blocks* are really 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 [14]). 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 [14] 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 [14]). We refine this evaluation to

$$c_F \cdot \left(\sum_{i=1}^{t-2} i(t-1)t^{-i} + (t-2)t^{(2-t)} \right) \cdot t^{t-2} \cdot \frac{T}{t} = c_F \cdot \left(1 + \frac{1}{t-1} \cdot \left(1 - \frac{1}{t^{t-3}} \right) \right) \cdot T \cdot t^{t-3} \qquad (2)$$

which 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 term is the cost of the one that 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. The main term is the same as the approximation provided [14].

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 Y_L is evaluated to $c_F \cdot t \cdot \frac{t}{2} \cdot (\frac{t}{t-1})^L$ (Equation (9) in [14], with t the average number of iterations along L, $\frac{t}{2}$ the average cost of recomputing a block, and the last term the expected number of attempts to compute one hash). This search cost can be refined, based on c_F and c_X costs, with the formula

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

The summation computes the cost of stopping when computing Y_j because a control block is found, with probability $\frac{1}{t}$, for which the attacker does not have suitable antecedents to produce as proof. The second part is the cost for the computation that got through the whole loop. The final term is the expected number of attempts to get one final value. Again, the main term is the same as [14].

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. although the special array would be leaked when providing openings which each include many control blocks. In order to thwart this attack, a simple counter measure, not mentioned in [14], is to make F depend on the challenge so that precomputations have to be performed for each challenge.

t	$\log_2$ Eq (2)	$d + \log_2$ Eq (3)
5	29.41	$d + 29.73$
10	47.86	$d + 19.81$
15	71.44	$d + 17.28$
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	$d + 15.40$

Table 1: Dinur and Nadler Attack Cost Numerical Evaluation in Hash calls

Table 1 shows numerical evaluation of the precomputation and search phase $\log_2$ costs of the Dinur-Nadler attack on MTP-Argon2, 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[2] compared to the non-cheating search. The attack benefit, without taking into account precomputation costs, which are assumed amortizable over many challenges, is ($\approx 0.0, 68.4$): almost no memory is needed, but at a significant cost. We show in Section 5.6 that such multiplier makes the attack only weakly beneficial even with favorable assumptions. The attack is avoided if several antecedents in the array construction are chosen with data-independent functions. It is also mitigated by a challenge-specific F which makes precomputations to be recomputed for each challenge.

3.5 Parallel Searches Attack

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 memory bandwidth [11], at least when the memory size required is larger than the available cache. Thus bandwidth [19] and possibly cache hit ratio are the relevant factors for miners who want to compute the hash function on a set of nonces. When under a memory bandwidth bottleneck, the prover is interested in performing more computation per loaded data, or with data already available in cache. To do that, the search paradigm can be transposed so that instead of fetching array X elements needed for updating the Y value for nonces, it rather fetches or uses the current 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 profitable if the search state for a nonce is small compared to an array element. A possible hardware implementation of such a solver is outlined in Section 5.5, with either a limited number of search states kept on-die or a large number stored in external memory. This issue is avoided if the search state is larger than array elements. This also impacts ASIC design if threads are needed to mask memory access latencies and the search state must be replicated.

3.6 Hash Composability Attack

The composability property of hash functions may change the memory requirements significantly. Hash functions use an internal state that 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_α, the final extraction is H_ω, and the $H_\pi(\text{block})$ is the state updating function, so that for instance $H(B_1 \parallel B_2) = H_\omega \circ H_\pi(B_2) \circ H_\pi(B_1) \circ H_\alpha$.

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 [9] is really implemented as $H(x \parallel y)$ and that on each call parameters are sorted so that the most recently known value is processed first by the hash function.

3.7 Parallel Function Attack

The Argon2 [5] specification emphasizes the internal parallelism of the memory-hard hash function with a parametric loosely interdependent number of lanes which induces issues of its own (Attacks 1 and 4 in [8]). Using parallelism makes sense for password checking on multi-core devices. On the verifier side, the hash value must be recomputed for checking the password, thus taking advantage of parallelism provides both speed and yet consumes resources. On the password-cracking-attacker side, parallelism will be used anyway to enumerate passwords. Whether this property is desirable for crypto-currency memory-hard PoW is at the least debatable.

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

The alternative is to have either a fully sequential or a highly parallel array construction phase. With the sequential (*i.e.* 1 lane) approach, all provers are penalized as they have a significant setup cost before the partial hash inversion search can start. The PoW is less progress free, and the penalty worsens with larger arrays. With a highly parallel approach, *e.g.* with independent segments of array elements, the PoW is more progress free as provers can use more of their processing power for the initialization phase. If segments are short enough, the verifier could actually verify fully the construction phase by recomputing the segments that appear in the opening. However, short segments are easier to recompute on the fly, thus could weaken the memory hardness of the approach. We investigate with Itsuku (Section 4) the segmented array approach with a clear security criteria. The result will be that such segments must be quite long, typically 2^{12} to 2^{15} elements, still allowing a lot of parallelism, but keeping a high full-verification cost.

4 The Itsuku PoW

We describe Itsuku, a new memory-hard PoW scheme closely inspired by MTP-Argon2 [9]. From this initial design, we retain the Merkel tree loopback which originates from [12], the iterative loop and the biased ϕ function, extend the array building parallelism for progress freeness, change most parameters and functions and add new operations to strengthen memory hardness. The new scheme fixes known issues presented in the previous section. Its design follows a simple and homogeneous security principle: any search implementation that requires half the array memory or less should imply a $\times$ 64 computation cost for a conservative difficulty $d \leq 100$. We also aim at reducing the proof size, which is an important criterion for applying the scheme to crypto-currencies.

The Itsuku search algorithm is, from I and d, with $P = \frac{T}{\ell}$, do:

1. Build challenge-dependent memory $X_I[0 \dots T-1]$ as P independent segments of length ℓ:
   (a) $X_I[p\ell + i] = H_x(i \parallel p \parallel I)$ for $0 \leq p < P$ and $0 \leq i < n$
   (b) $X_I[p\ell + i] = F_{p,I}(X_I[p\ell + \phi_0(i)], \dots, X_I[p\ell + \phi_{n-1}(i)])$
   for $0 \leq p < P$, $n \leq i < \ell$ and assuming $\forall k, 0 \leq k < n, 0 \leq \phi_k(i) < i$
2. Compute Merkle tree root hash Φ of X with $H_M^I(e) = H_M(e \parallel I)$
   (a) Let $B[0 \dots 2T-2]$ be an array of $2T-1$ elements of size M bytes
   (b) Compute array X element leaf hashes: $B[i+T-1] = H_M^I(X_I[i])$ for $0 \leq i < T$
   (c) Compute intermediate node hashes: $B[i] = H_M^I(B[2i+1] \parallel B[2i+2])$ for i in $T-2 \to 0$
   (d) Root hash Φ is $B[0]$
3. Choose Nonce N
4. $Y_0 = H_S(N \parallel \Phi \parallel I)$
5. For $1 \leq j \leq L$ compute:
   (a) $i_{j-1} = Y_{j-1} \bmod T$
   (b) $Y_j = H_S(Y_{j-1} \parallel X_I[i_{j-1}] \oplus I)$
6. Back sweep over intermediate hashes in reverse order
   $\Omega = H_S(Y_L \parallel \dots \parallel Y_{1-L \bmod 2} \oplus I)$
7. If Ω has d binary leading zeros, the PoW search ends, otherwise go to Step 3
8. Final output is $(N, \mathcal{I}, \mathcal{L}, \mathcal{Z})$ where:

- $\mathcal{I} = \{i_j\}$ for $0 \leq j < L$, the indexes of selected leaves.
- $\mathcal{L}$ is for selected leaves either their n ordered antecedents if $i_j \bmod \ell \geq n$, or the leaves themselves otherwise (although they could also be recomputed in this case).
- $\mathcal{Z}$ is the collective *opening* (Merkle tree proof) of both the selected leaves and their antecedents if any.

Some provisions could be made to avoid repeating a leaf contents if it appears twice through some ordering convention.

The verification algorithm is, from I, d and the proof output:

1. With $\mathcal{I}$ and $\mathcal{L}$, compute $X[i_j]$ leaves with $F_{p,I}$.
2. With $X[i_j]$, $\mathcal{L}$ and $\mathcal{Z}$ compute Φ, the root hash of the Merkle tree.
3. With N, Φ, I and $X[i_j]$ compute Ω and keep the set of selected indexes.
   (a) Check that selected indexes are exactly $\mathcal{I}$.
   (b) Check that Ω has d binary trailing zeros.

Preferred Parameters The choices determine whether the security constraint is met. Our preferred parameters, justified below, are: T at least 2^{25} and scheduled to grow depending on d or chain height, $x = 64$, $\ell = 2^{15}$, $n = 4$, $M \approx \frac{d+\log_2((c_X+1/2)\cdot L)+6}{8}$, $L = 9$, $S = x$, ϕ_k functions...

Proof Size The formula is extended to include the number of antecedents and to provide a merged opening instead of independent per-leaf openings, to

$$16 + L\left\lceil \frac{1}{8}\log_2 T \right\rceil + n\,L\,x + (n+1)\,L\,(\log_2 T - \log_2 L - \log_2(n+1))\;M \tag{4}$$

which, with our parameters for 2 GiB and realistic $d = 70$, leads to about 11 KiB.

Complexity Let c_X be the cost for computing one array X element hash at Steps 2b or 5b and c_F the cost of calling $F_{p,I}$ once as Step 1b. The search computation complexity in hash-block calls is about $(c_F + c_X + 1)\cdot T + \left(1 + (c_X + \frac{S}{128})\cdot L\right)\cdot 2^d \propto L\,2^d$ (if X initialization is negligeable). The search memory access complexity on array X is about $(n+1)\,T + L\,2^d$. With our prefered parameters above, we have about $2^{26.6} + 2^{d+3.9}$ hash operations. The verification complexity is about

$$c_F L + (n+1)\,c_X\,L + (n+1)\,(\log_2 T - \log_2 L - \log_2(n+1))\;L + 1 + \left\lceil \left(c_X + \frac{1}{2}\right)\;L \right\rceil \approx$$
$$\left(c_F + (n+2)\,c_X + (n+1)\;(\log_2 T - \log_2 L - \log_2(n+1)) + \frac{1}{2}\right)\;L$$

i.e. with our parameters around 945 hash operations, most of which for the opening check. Actually checking the X values is parallel and would cost another $n\,L\,c_F\,\frac{\ell}{2} \approx 2^{19.2} \approx 600,000$ hash calls, which although large is still doable in 1/8 second with 2 MiB of memory on one 3 GHz core.

Dependence on Challenge I One key enabler of the Dinur-Nadler attack is that array X dependency on the challenge was not really checked because compression function F was fixed; thus expensive precomputations could be done to build a special array independent of challenge I and reuse it for each PoW. To avoid this kind of issue, a challenge-dependent function F_I is devised, so that pre-computations needed for an attack would have to be specific to each challenge. Similarly, the hash function for computing the Merkle tree and used in the search is also made challenge-specific. This provides an indirect way to check that the PoW is fully specific to the challenge, as the challenge is used at every step by the prover and the verifier.

Element Size x MTP-Argon2 array X element size is 1024 bytes. It requires a special F compression function to build new elements from their two antecedents. Moreover, as the search state (about $S+12$) is smaller than an array element, it allows efficient parallel searches (Section 3.5), which share loading array elements from memory between several searches, improving the performance under a memory bandwidth bottleneck, as shown in Section 5.5. Also, the array elements constitute a major part of the final proof size. These arguments push towards a smaller size. We choose $x = 64$ bytes which is the largest hash size provided by a single call to the BLAKE2 hash function, so that the F compression function can be directly based on H, leading to $c_F = c_X = 1$.

Hash Function H BLAKE2 [6] choice as a base hash function is sound. A single processing call produces a 64 bytes hash from a 128 bytes input. As the BLAKE successor, its security has been extensively discussed and is well understood, and hardware designs have been proposed for this family of hash functions [18]. We think that using unaltered secured hash functions allows to build trust into the schemes that use them; so reducing rounds or changing operators or constants of existing hash functions (see Section 6) should be avoided. In order to ensure the dependency on the challenge, we have used $H(\ldots \parallel I)$ when the input is expected below 128 bytes (Steps 1a, 2 and 4), and $H(\ldots \oplus I)$ (applied at the end) when the input is expected to be larger (Step 5b and Step 6), so that the dependency does not add significant computation costs to the scheme.

Enhanced Hash Function H Following [9], we recommand that a fast ASIC-expensive hash function, involving costly operators available in CPU and GPU such as large width multiplication or division, should be designed. From this perspective, BLAKE2 can be improved upon, as the design criteria and selection process for hash functions are rather to minimize hardware footprint and maximise throughput by using simple logical operators. A convenient function can also be built on top of a cryptographic hash function by adding an external xor layer that uses expensive operators, *i.e.* $H'(v) = H(v) \oplus E(v)$, with $v = v_{0\ldots15}$ decomposed as 16 unsigned 64-bit integers with indexing modulo its size, then for $0 \leq i < 8$

$$q_i = v_i \cdot v_{i+8} + v_{i+4},\ d_i = v_{i+12}|2^{32} \quad \text{and} \quad E(v) = \|_{i=0}^{7}(q_i \div d_i) \lll 32 \oplus (q_i \bmod d_i)$$

adds 32 integer arithmetic and 24 bitwise operations on 64-bit integers. The combining construction retains the proven security of H and adds the cost of E.

Compression Function $F_{p,I}$ With $x = 64$ the array building compression function can use hash H. In passing, we want to avoid special properties such as $F(B, B) = 0$ which can propagate. Also, we want to avoid simplifications in case the ϕ_k functions collide for an index. The function must depend both on I and p so that each challenge and parallel segment has its own unique computations, avoiding potential Bevand-like attacks [8]. We suggest to rely on $+_{64}$ modular addition on the array element considered as a vector of 8 unsigned 64-bit integers to combine array elements so that only one hash call is needed, for $i \geq n$:

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

Array Size T In order to maintain the same overall array X size, its number of elements is scaled proportionnaly to the reduction, so that Tx achieves the expected memory footprint, *i.e.* $T = 2^{25}$ for 2 GiB. Building array X with H is slightly more expensive with 16 hash calls per KiB instead of 11 in the MTP-Argon2 scheme. Requiring the initialization phase to be negligeable compared to the search phase implies $d > \log_2 T + \log_2 3 - \log_2 L$, which with our chosen L translates to $d \geq 24$.

Size T Growth Section 5 shows hardware implementations with 2 GiB arrays. The efficiency of a dedicated chip changes significantly with the area budget considered, which depends on evolving technology. We thus advise to consider extending T to 2^{26} or even 2^{27} to avoid efficient implementations allowed by improved technologies in the mid term. We also suggest that the overall array size must not be fixed but allowed to grow. An additional benefit of a varying memory size requirement is that it provides another deterrent against ASIC investments, as they would become more risky if they stop working for some size. A possible scheme to achieve a long term growth is to make the array size depend on difficulty d, which is automatically adjusted in a cryptocurrency in order to control the blockchain throughput. For instance, a linear formula like $\log_2 T = 21 + \frac{d}{8}$ would inflate the array from 1 GiB with $d = 24$ up to 16 GiB with $d = 63$. The parameters should be chosen carefully based on projected mining capabilities and technology evolution.

Another option is to specify the size of the array depending on the blockchain height, so that it grows with age, *e.g.* for doubling every 6 years: $\log_2 T = 25 + \frac{current\ height}{6 \cdot blocks\ per\ year}$. The drawback of age-dependent growth is that it is hard to foresee the long term technology pace and limit and the mining resources dedicated to the ledger. Accepting non power of two sizes can smooth the slope at the expense of some code complications, such as modulo operations in the search and handling uneven Merkle trees. This could lead to defining parallelism P_d for fixed segment length ℓ ($T = P \cdot \ell$) with some formula such as $P_0 = \frac{2^{21}}{\ell}$ and for $d > 0$ some rational progression on 64-bit integers, *e.g.* $P_d = {}^{4{,}580{,}259}/_{2^{22}} \cdot P_{d-1}$, or even some hardwired precomputed values. This would help prevent miner instability, as suddenly doubling the memory requirement could rule out part of available mining power from one block to the next.

Constant Array X? In the context of crypto-currencies PoW, does it matter if the array depends on challenge I? It depends. A desirable property is that the PoW verifier can check the PoW locally, without actually allocating the array. If this property is not needed for the use case, say because the verifier already shares large amount of data with miners, then array X may be constant or shared somehow. A possible construction scheme for building a large array X could be to use the Argon2 scheme [5] using at least two sweeps ($t > 1$) on some password, or possibly to rely on blockchain data [10]. With this approach, the memory does not need to be checked and the whole Merkle tree (Phase 2) can be skipped, as well as sending array elements (Phase 8). This would remove most of the weight of the PoW. This approach would result in a more classical partial hash inversion search intertwined with array accesses. In the following, we will assume that the array is challenge-specific and that its construction must be checked, without verifiers needing to allocate a large shared array.

Hash 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 for threaded parallel searches. Percival [22] defines memory-hard as a function that requires memory proportionally to its computation cost. The back sweep ensures that the internal memory requirement for computing the PoW candidate from the nonce is proportional to the computation, especially as recomputation tradeoffs cannot be beneficial: although intermediate Y_j values could be recomputed from available data, keeping their value around takes less space than their dependencies as $S = x$, thus are not worth it. If pipelining is used, the average number of values needed is about $\frac{L}{2} + 1$ per search and should be considered to dimension the memory requirements.

Search Length L In [12], a $\log_2 T$ formula computes the number of leaves of a Merkle tree to prove its whole computation. This formula applies when the Merkle tree constitutes most of the computation, *i.e.* $2\,T > L\,2^d$. If we assume that the search phase is preeminent, the number of proofs can be much reduced because the search loop low success rate ensures that the array is scanned extensively. In this context, the search length needs to be large enough to ensure that if array X is only partially available the search phase would cost significantly more so as to respect our security

constraint. If fraction f of the array is available, then the search cost in hash calls with $c_X = 1$ is

$$\left(\sum_{k=0}^{L-1}(1+k)f^k(1-f) + \left(1+\left\lceil\frac{3}{2}L\right\rceil\right)f^L\right)\cdot f^{-L}\cdot 2^d = \left(\left\lceil\frac{1}{2}L\right\rceil + \frac{f^{-L}-f}{1-f}\right)\cdot 2^d$$

where the summation computes the weighted costs of attempts on nonces failing because of missing array elements at step $k+1$, followed by the cost of the one that succeeds to produce an Ω, multiplied by the expected number of attempts to produce a working Ω and finally the number of attempts to succeed as a PoW. The search cost multiplier, which we want above 64 when $f \leq \frac{1}{2}$, is then

$$m(f,L) = \frac{\lceil 1/2 \cdot L\rceil + \frac{f^{-L}-f}{1-f}}{\lceil 3/2 \cdot L\rceil + 1} \tag{5}$$

$L = 9$ gives safe enough $m(\frac{1}{2}, 9) = 68.5$ (but also $m(\frac{1}{2}, 8) = 39.6$, $m(\frac{3}{4}, 9) = 3.7$ and $m(\frac{3}{4}, 8) = 3.2$).

Number of Antecedents n This 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 average recomputation cost $c_R = 1.0$ when half the array is available. We investigate $n \geq 3$ in order to enlarge the cost in this case, so as to fulfill our security objective. The larger the parameters, the more array elements are included in the PoW when providing antecedents, thus the larger the proof, but also the larger the recomputation cost. The investigation in the next paragraphs leads us to $n = 4$.

Indexing Functions ϕ_k and Bias We are going to evaluate the recomputation cost for a missing array element when every other element is skipped. This cost is very sensitive to the number of antecedents and to the choice of indexing functions ϕ_k with $0 \leq k < n$. We considered the following debatable mixture of data-dependent and independent functions up to 6 antecedents: $\phi_0(i) = i - 1$, $\phi_1(i) = \phi(i)$, $\phi_2(i) = \frac{\phi(i)+i}{2}$, $\phi_3(i) = \frac{7 \cdot i}{8}$, $\phi_4(i) = \frac{\phi(i)+3 \cdot i}{4}$ and $\phi_5(i) = \frac{3 \cdot \phi(i)+i}{4}$, where $\phi(i)$ is the data-dependent quadratic-biased function defined in the Argon2 scheme, or possibly a cubic-biased version, with $0 \leq \phi(i) < i$. A side effect of adding dependencies with several data-independent indexing functions (ϕ_0 and ϕ_3 above) is that the scheme is immune to the Dinur-Nadler attack, as the necessary precomputations cannot fall on control blocks for all i.

n \ $\log_2 \ell$	9	10	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	1.0	1.0
3	3.9	4.5	5.2	5.8	6.5	7.2	7.9	8.7	9.3	10.2
4	17.1	27.4	43.6	69.3	110.0	174.9	**278.4**	442.7	704.7	1125.4
5	32.7	58.7	105.3	189.2	**340.0**	612.0	1101.3	1982.8	3569.4	6429.6
6	37.3	69.6	129.3	**240.2**	446.0	828.2	1536.2	2853.4	5295.5	9815.7

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

Recomputation Cost c_R Table 2 shows the average recomputation cost c_R in F calls to access array elements when every other element is available, for various segment lengths and numbers of dependencies, using the particular ϕ functions presented previously. The value is an average of 200 to 100,000 simulations, depending on the simulation time. This cost is very sensitive to the chosen ϕ functions: more or less biased functions result in longer pseudo-random walks for recomputing values, thus higher costs. The typical random walk length at index i from the beginning of the segment is typically $\beta \log_2 i$ where β depends on the bias. Other storage strategies we have tested, such as keeping the first half and second half, or even mixed strategies with every-other for the first half followed by full storage for the next quarter and just missing elements for the last one, resulted in significantly larger

access costs: we conjecture that the every-other approach might the best possible one, or close to. A rough analytical model can also be derived: when every other element is available, the cost of accessing an element is either 0 (the element is available) or it must be recomputed. When it is recomputed, its predecessor is necessary available, thus $(n-1)$ other antecedents must be accessed, and may or may not be available recursively. Accessing element i should thus cost about $c(i) = 1/2 \cdot (1 + (n-1)\, c(\phi(i)))$, which with $n > 3$ leads to $c(i) = \frac{1}{n-3}\,(i^{\beta(\log_2(n-1)-1)} - 1)$. When averaging over segment length ℓ, with $H_n^{(m)}$ the generalized harmonic number, we then have

$$c_R \approx \frac{1}{n-3}\left(\frac{1}{\ell} \cdot H_{\ell-1}^{(-\beta\cdot(\log_2(n-1)-1))} - 1\right)$$

Note that this approximation does not take into account collisions on recomputed elements.

Segment Length ℓ and Parallelism P Based on the previous recomputation cost evaluation and with $c_X = 1$, the associated cost multiplier for the search algorithm is about $\frac{1+(c_R+3/2)\cdot L}{1+3/2\cdot L} \approx \frac{2\cdot c_R}{3}$ if $c_R \gg \frac{3}{2}$ and $L \gg 1$, so we are looking for $c_R > 96$. We suggest to take an additional at least $\times\, 2$ margin to account for possible clever storage strategies. Small number of antecedents do not allow to reach the target cost level. As expected, the greater the number of antecedents or the array size, the higher the cost, hence the multiplier. Choosing an even (for F symmetry) but not too large (for limiting the number of leaves to provide in a proof) results in the number of dependencies $n = 4$, with $\ell = 2^{15}$ and $c_R = 278.4$. The choices induces a significant parallelism $P = 2^{10}$ for 2 GiB. If half-array storage strategies are better analysed, we may go as low as $\ell = 2^{13}$ leading to $P = 2^{12}$. While a parallel initialization benefits largely from cache effects on each independent segment, thus contribute to better progress freeness, the later search which encompasses all segments does not.

Hash Size M To reduce the PoW size, the Merkle tree computation can use a smaller hash size without ampering the overall security [12]. With a simple criterion that one inversion should cost more than the whole PoW, for our preferred parameters this leads to $2^{8\cdot M} \geq 2^{26.6} + 2^{d+3.9}$, which is roughly satisfied with a $\times\, 2^6$ margin if $M = \lceil \frac{d+\log_2(1+c_X L+\lceil L/2\rceil)+6}{8} \rceil$ and if the initial phase is negligeable. For realistic $d \leq 70$ this gives $M \leq 10$, and for conservative $d \leq 100$, $M \leq 14$. Note that collisions on Φ do not constitute a replay, as computations both before and after depend on challenge I.

5 Software and Hardware Implementations

This section evaluates optimistic software and hardware implementations of both MTP-Argon2 and Itsuku. We show that CPU or GPU software implementations are expected to be computation-bound, but the dedicated hardware bottleneck is rather memory bandwidth.

We first describe the expected performance on CPU and GPU (Section 5.1). Then we present the optimistic hardware hypotheses which are used for designing hardware PoW solvers (Section 5.2). We consider different hardware implementations: direct approach (Section 5.3), half-array compression (Section 5.4), parallel searches (Section 5.5) and Dinur-Nadler attack (Section 5.6). The designs are instanciated based on current hardware capabilities, examplified by the NVIDIA Volta V100 GPU [20] (June 2017 – 12 nm process, 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), and possible future larger capabilites, such as a 30 billion transistor chip by 2019 [17] and hypothetical 50 billion and 100 billion transistor chips by 2022 and 2026 respectively (Section 5.7).

5.1 Software Implementations

We first present the expected CPU and GPU software implementation performances.

CPU Performance BLAKE2 processes 947 MiB/s [2] on one core of an Intel Core i5-6600 (Skylake microarchitecture, 3.3 GHz, 14 nm, Q3'2015), that is about 3.33 cycles per byte. Given the typical memory bandwidth of multi-core CPU in tens of GB/s, the performance is likely to be computation-bound. We thus expect 427 cycles per 128-byte hash call, *i.e.* a one-core hash tick runs at 7.77 MHz ($2^{22.9}$ hashes per second). As the 2 GiB initialization phase is around 100 M ($2^{26.6}$) hash calls, the initialization computation could take about 13 seconds on a single core, or even more because of memory latency when building with 1 lane [8]. This duration may be considered large for a PoW expected to return a proof every few minutes, and justifies our effort to parallelize the initialization phase significantly. Provided that memory latency can be masked during the search phase, each core would probably produce 12,300 ($\approx 2^{13.6}$) candidate-Ω per second for MTP-Argon2, and about 518,000 ($\approx 2^{19}$) candidate-Ω per second for Itsuku.

GPU Performance BLAKE2 hash computations are amenable to parallel SIMD implementations, thus suit the GPU programming model and available operators. If memory bandwidth is a bottleneck, the $N_{\mathcal{B}}$ figures (Section 5.3) in Ω/tick would provide a hint of the expected performance, provided that cores can keep up with the computations. On a GPU V100 this would mean that Itsuku performs at up to 1.7 GΩ/s. With 5,120 cores running at 725.5 MHz, that would imply one hash performed every 143 cycles. However, BLAKE2 performance figures rather suggest at least 427 cycles per hash calls on 64-bit architectures; thus on GPU the performance would rather be computation-bound at about 566 MΩ/s (about 2^{29} Ω/s), typically consumming 1/3 of the available memory bandwidth.

5.2 Hardware Implementation Hypotheses

Instead of considering abstract implementations and time-area ratios as [9], we discuss ambitious hardware designs for PoW solvers that pipeline hash cores to implement the search loop and produce one candidate Ω per tick. We intentionally make optimistic assumptions to be on the safe side; so the performance figures must be interpreted as upper bounds that may not be really achieved by actual implementations: transistors are switched freely between hash cores and SRAM, the internal (on die) bandwidth is never a bottleneck, and cache lines fit our element size perfectly.

The gate-equivalent (GE) budget $\mathcal{G}$ of the chip is shared between BLAKE2 hash cores of area $\mathcal{H}$ (in GE) running at $\mathcal{F}$ (in Hz) and on-die SRAM of area $\mathcal{M}$ GE per byte and if necessary external DRAM access with bandwidth $\mathcal{B}$. PoW variants are abstracted with the size of the array $Tx = 2$ GiB, the number of search loop hashes $\mathcal{C}_0$ and array element accesses L to compute a candidate Ω, the cost of computing or hashing an array element (c_F or c_X), and the search state size $\mathcal{S}$.

In order to evaluate the area cost for a BLAKE2 core, we consider BLAKE-64 VLSI implementations. A high performance 8G (compression function) hardware is evaluated to 128 kGE and 15 cycles running at 298 MHz to process a 128-byte input block (Table III in [18]), 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 $\mathcal{H} = 100$ kGE can run in 10 cycles at 300 MHz, which will be considered as 1 tick, *i.e.* a tick is one 128-byte input block hashing and runs at $\mathcal{F} = 30$ 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. With 4T SRAM, we have $\mathcal{M} = 8$ GE/B.

5.3 Direct Approach Hardware

We investigate the implementation of a pipelined PoW solver with the direct method. Each solver accesses L array elements per tick. There are two cases: either the array can be stored on die ($2\text{ GiB} \cdot \mathcal{M} \leq \mathcal{G}$), or not.

On-die Array A 2 GiB 4T SRAM array requires 17.2 GGE (69.7 billion transistors), way beyond current capabilities. However, once available, the array would consume part of the area budget and the

remainder could be used for active hash cores. If we assume optimistically that on-die array accesses are not a bottleneck, then we can cram $N_{\texttt{sram}} = \frac{\mathcal{G}-2\ \text{GiB}\cdot\mathcal{M}}{\mathcal{C}_0\cdot\mathcal{H}}$ on-die PoW solvers producing $N_{\texttt{sram}}\,\mathcal{F}$ candidates Ω per second, with an implied on-die bandwidth of $x\,L\,N_{\texttt{sram}}\,\mathcal{F}$ bytes per second.

In DRAM Array Otherwise the array would be stored in DRAM, with the cpu area budget shared between hash cores and cache. In order to mask the induced latency [25], typically $\theta = 9$ threads holding $\mathcal{S}$ bytes of search state can be used. The performance bottleneck is either the computation or the memory bandwidth. The number of PoW solvers for a given area budget is $N_{\mathcal{G}} = \frac{\mathcal{G}}{\mathcal{C}_0\mathcal{H}+\theta L\mathcal{S}\mathcal{M}}$. The number of PoW solvers than can be fed with a given bandwidth is $N_{\mathcal{B}} = \frac{\mathcal{B}}{xL\mathcal{F}}$. As we typically have $N_{\mathcal{B}} \ll N_{\mathcal{G}}$, most of the area budget can be dedicated to cache, which reduces the bandwidth requirement and thus improves markedly the overall performance. The cache hit ratio is then $\gamma = \frac{\mathcal{G}-N'_{\mathcal{B}}(\mathcal{C}_0\mathcal{H}+\theta L\mathcal{S}\mathcal{M})}{2\ \text{GiB}\cdot\mathcal{M}}$, with the number of solvers to feed $N'_{\mathcal{B}} = \frac{\mathcal{B}}{(1-\gamma)xL\mathcal{F}}$. The memory-bottleneck PoW throughput is then $N'_{\mathcal{B}}\,\mathcal{F}$ candidates Ω per second. The overall bandwidth limited design is not very efficient because only a small part of the area is actually dedicated to computing.

5.4 Half-Array Compression Attack Hardware

The half-array compression attack consists in keeping part of the array and recomputing the missing elements when needed, as discussed in Section 3.2. We assume that every other element of the array is kept. Let c_R be the average number of F calls to access and possibly recompute one array element from its antecedents. The number of cores for a fully pipelined PoW solver is then $\mathcal{C}_\partial = \mathcal{C}_0 + c_R \cdot c_F \cdot L$. Moreover, the average number of actual memory accesses for an array element is $m_\partial = (n-1)\cdot c_R + 1$, either by fetching it directly or recursively via its dependencies for recomputation. As with the previous case, either the half array fits on die, or not.

On-die Half Array If the 1 GiB half array fits on die, then the remainder area can be used for hash cores, providing $N_\partial = \frac{\mathcal{G}-1\ \text{GiB}\cdot\mathcal{M}}{\mathcal{C}_\partial\cdot\mathcal{H}}$ PoW solvers, which output $N_\partial\,\mathcal{F}$ candidates Ω per second. The implied on-die bandwidth is $m_\partial\,x\,L\,N_\partial\,\mathcal{F}$ bytes per second, which would be typically very large.

In DRAM Half Array Otherwise the half array must be stored in DRAM. The number of solvers for a given area budget, including thread states, is $N_{\partial,\mathcal{G}} = \frac{\mathcal{G}}{\mathcal{C}_\partial\mathcal{H}+m_\partial L\theta\mathcal{S}\mathcal{M}}$. The number of solvers for a given bandwidth is $N_{\partial,\mathcal{B}} = \frac{\mathcal{B}}{m_\partial xL\mathcal{F}}$. In practice $N_{\partial,\mathcal{B}} \ll N_{\partial,\mathcal{G}}$, *i.e.* the performance is bandwidth-limited. Similarly to the direct approach, the remaining area can be used for cache with a cache hit ratio $\gamma = \frac{\mathcal{G}-N'_{\partial,\mathcal{B}}\mathcal{C}_\partial\mathcal{H}}{\mathcal{M}\cdot 1\ \text{GiB}}$, with the number of PoW solvers that can be fed $N'_{\partial,\mathcal{B}} = \frac{\mathcal{B}}{(1-\gamma)m_\partial xL\mathcal{F}}$. The memory bottleneck PoW throughput is then $N'_{\partial,\mathcal{B}}\,\mathcal{F}$ candidates Ω per second.

Pseudo-Random Array This attack [14], discussed in Section 3.3, is similar to a half array as elements are recomputed on the fly with one F call applied on regenerated antecedents. It can be thwarted by actively checking openings for inconsistencies or including the selected array elements in the PoW openings. The PoW solvers can produce about $N_{\text{PRA}} = \frac{\mathcal{G}}{(\mathcal{C}_0+c_F L)\cdot\mathcal{H}}$ candidates Ω per tick.

5.5 Parallel Search Hardware

We now investigate a parallel search hardware implementations (Section 3.5). The first approach is to keep search states on die, the second is to communicate both array elements and search states from and to external DRAM. The implementations achieve the best performance for MTP-Argon2 with our hypothetical hardwares (Section 5.7), unless the array is stored on-die with a very high bandwidth.

On-die Search States Instead of fetching needed array elements, on-die parallel searches move forward depending on arriving array elements. Let $\mathcal{S}+4$ be the search state size and a minimum array and linked-list data structure to store searches per array element. Holding one search per array element on average requires about $T\,(\mathcal{S}+4)\,\mathcal{M}$ area; so the maximum number of searches per array element is at most $n_{\|} \leq \lfloor \frac{\mathcal{G}}{T(\mathcal{S}+4)\mathcal{M}} \rfloor$. The memory bandwidth limits the number of array elements that can be brought to $\mathcal{B}/x$ per second, each pushing forward $n_{\|}$ searches, implying $n_{\|}\, c_X\, \mathcal{B}/x$ hashes per second to do so, thus requiring $n_{\|}\, c_X\, \mathcal{B}/x\, \mathcal{F}$ array element processing hash cores. One final hash is produced every such $c_X L$ hashes, thus the overall performance is $n_{\|}\, \mathcal{B}/x\, L$ candidate Ω per second. Another interpretation of this figure is that the array element throuput $\mathcal{B}/x\, L$ for computing one Ω is shared between $n_{\|}$ searches. For each array element loaded, the solver accesses $n_{\|}(\mathcal{S}+4)$ search state bytes. The overal data moved internally for each element is $\mu\, n_{\|}\,(\mathcal{S}+4)+x$, (with $\mu = 2$ without back-sweep and $\mu = 3$ with it), for the array element and for updating (read and write) their corresponding search states. The implied on-die internal bandwidth is thus $\left(1+\mu\, n_{\|}\,{}^{(\mathcal{S}+4)}/_{x}\right)\mathcal{B}$, which is above but typically close to the DRAM bandwidth.

In DRAM Search States Another implementation is that both array elements and search states are stored in a large DRAM, *e.g.* 32 GiB shared between 2 GiB of array and 30 GiB of search states which allows for about 480 MTP-Argon2 search states for each array element. In that setting, array elements are sent to the chip, then search states are sent to be updated and stored back in DRAM. Loading an array element can be amortized on many states, so that the performance limit is really the transfers in and out of search states $\mathcal{B}/\mu(\mathcal{S}+4)$ requiring each $\mathcal{C}_0/L$ cores to be processed. Also, when there is a back-sweep, on average for each current search about $L/2$ hash results of size $\mathcal{S}$ must be kept in memory, instead of juste one without back-sweep. It generates $\mathcal{B}/\mu\,(\mathcal{S}+4)\,L\,\mathcal{F}$ candidate Ω per second. This can be larger that the previous on-die approach depending on the relative strength of the bandwidth compared to the area that can be used to store search states. The implied internal on-die bandwidth is basically $\mathcal{B}$, *i.e.* the same as the external memory bandwidth.

5.6 Dinur-Nadler Attack Hardware

The Dinur-Nadler attack has been summarized in Section 3.4 with a multiplier cost c_{DN} applied to the search phase. When the attack is carried out, the memory requirement for array X is much lower; thus it is assumed as negligeable and available on die. The number of on-die pipelined PoW solvers is then $N_{\text{DN}} = \frac{\mathcal{G}}{c_{\text{DN}}\,\mathcal{C}_0\,\mathcal{H}}$, which provide $N_{\text{DN}}\,\mathcal{F}$ candidates Ω per second. The implied on-die bandwidth is low as elements are recomputed and would fit on close-to-computation caches.

5.7 Hardware Implementation Evaluation

This section discusses the expected performance of 2 GiB PoW variants.

PoW name	Parameters									Costs		
	c_F	c_X	n	x	T	L	S	bs	P	c_R	$\mathcal{S}$	c_{DN}
MTP-Argon2 [9]	9	11	2	1024	2^{21}	70	16	F	1(4)	1.0	28	68.4
Itsuku, Section 4	1	1	4	64	2^{25}	9	64	T	2^{10}	278.4	380	–

Table 3: MTP Variants for 2 GiB array

Table 3 summarizes the parameters of MTP-Argon2 and Itsuku. Column bs tells whether there is a back-sweep over Y values to compute the final Ω. The number of search loop hashes is $\mathcal{C} = 1 + c_X \cdot L + \lceil \frac{L}{2} \rceil \cdot bs$. Search state size $\mathcal{S}$ is taken as a $\frac{1}{2}LS$ average over the pipeline length. Itsuku is immune to the Dinur-Nadler attack if several indexing functions are data-independent.

Table 4 shows performance on an NVIDIA Volta V100 GPU inspired hardware. The pseudo-random array attack on MTP-Argon2 is supposed so be thwarted somehow, so its performance is only shown for illustrating its effect. Otherwise, for the transistor budget, most of the area is dedicated

PoW name	Implementations		$\mathcal{C}$	$N_{\mathcal{G}}$	$N_{\mathcal{B}}$	Ω/tick	MΩ/s
MTP-Argon2 [9]	DRAM full array	and cache	631	83.41	0.47	0.67	20.1
	DRAM half array	and cache	1401	37.58	0.23	0.59	17.6
	Pseudo-random array	*thwarted?*	1401	37.65	–	*37.65*	*1129.6*
	Parallel search	$N_{\parallel} = 9$	2641	–	4.19	4.19	125.6
		in DRAM	4695	–	7.44	**7.44**	**223.2**
	Dinur-Nadler attack	$c_{\mathtt{DN}} = 68.4$	43160	1.22	–	1.22	36.7
		$c_{\mathtt{DN}} = 116.2$	73322	0.72	–	0.72	21.6
Itsuku, Section 4	DRAM full array	and cache	15	3002.95	57.87	**82.51**	**2,475.2**
	DRAM half array	and cache	2521	11.3	0.07	0.18	5.3
	Parallel search	$N_{\parallel} = 0$	–	–	–	–	–
		in DRAM	49	–	3.27	3.27	98.0

Table 4: MTP variants with NVIDIA Volta V100 GPU technology ($\mathcal{G} = 5.275$ GGE, $\mathcal{B} = 1$ TB/s)

to on-die memory, but the Dinur-Nadler implementations. The designs are not very efficient, because the algorithms' bottleneck is the memory bandwidth and the over 50,000 BLAKE2 hash cores that could fit on die cannot be usefully employed. The best performance of MTP-Argon2 is achieved by the transposed parallel search with states in DRAM, thanks to the small search state and relatively good bandwidth. The half-array compression performance is quite close to the full array performance because the additional costs are masked by the improved cache hit ratio. Both Dinur-Nadler performance figures are on the optimistic side because the precomputation costs are amortized on a very large number of PoW. Itsuku cannot implement the on-die transposed search because of the large memory requirement induced by the larger search state size and number of array elements. Its half-array implementation performance is very low because of the high number of array element accesses to recompute missing elements.

PoW name	Implementations		$\mathcal{C}$	$N_{\mathcal{G}}$	$N_{\mathcal{B}}$	Ω/tick	MΩ/s
MTP-Argon2 [9]	DRAM full array	and cache	631	118.59	0.60	1.07	32.0
	DRAM half array	and cache	1401	53.43	0.30	1.91	57.3
	Parallel search	$N_{\parallel} = 13$	4960	–	7.86	7.86	235.8
		in DRAM	6104	–	9.67	**9.67**	**290.2**
	Dinur-Nadler attack	$c_{\mathtt{DN}} = 68.4$	43160	1.74	–	1.74	52.1
		$c_{\mathtt{DN}} = 116.2$	73322	1.02	–	1.02	30.7
Itsuku, Section 4	DRAM full array	and cache	15	4269.59	75.23	**130.43**	**3,913.0**
	DRAM half array	and cache	2521	16.07	0.09	0.57	17.1
	Parallel search	in DRAM	63	–	4.20	4.20	126.0

Table 5: Hypothetical 30 billion transistor chip by 2019 ($\mathcal{G} = 7.5$ GGE, $\mathcal{B} = 1.3$ TB/s)

Table 5 shows similar performance figures for an hypothetical 30 billion transistor chip, for which we assume a somehow slower-paced memory bandwidth improvement. The overall results are similar to the previous case. For MTP-Argon2 the best performance is achieved by the parallel search. The half-array compression performance beats the direct approach, because the chip is nearly large enough to hold the half array; thus the cache hit ratio is high. This effect does not apply to Itsuku, thanks to the significant recomputation costs.

Table 6 shows performance figures on a 50 billion transistor chip. The transposed parallel search performs better with states stored on-die compared to DRAM because of the decreasing relative bandwidth we have chosen. For this area, the half array fits on die, thus achieves the best performance but for Itsuku, which is limited by its high recomputation cost.

Finally, Table 7 shows performance figures on a 100 billion transistor chip. For this area size, the whole array fits on die and there is significant room for hash cores.

PoW name	Implementations		$\mathcal{C}$	$N_{\mathcal{G}}$	$N_{\mathcal{B}}$	Ω/tick	MΩ/s
MTP-Argon2 [9]	DRAM full array	and cache	631	197.66	0.93	3.27	98.1
	SRAM half array		1401	89.04	–	**27.91**	**837.3**
	Parallel search	$N_{\parallel} = 20$	11738	–	18.60	18.60	558.1
		in DRAM	9390	–	14.88	14.88	446.4
	Dinur-Nadler attack	$c_{\mathtt{DN}} = 68.4$	43160	2.90	–	2.90	86.9
		$c_{\mathtt{DN}} = 116.2$	73322	1.70	–	1.70	51.1
Itsuku, Section 4	DRAM full array	and cache	15	7115.99	115.74	**372.74**	**11,182.1**
	SRAM half array		2521	26.79	–	15.51	465.5
	Parallel search	in DRAM	97	–	6.47	6.47	194.0

Table 6: Hypothetical 50 billion transistor chip by 2022 ($\mathcal{G} = 12.5$ GGE, $\mathcal{B} = 2.0$ TB/s)

PoW name	Implementations		$\mathcal{C}$	$N_{\mathcal{G}}$	$N_{\mathcal{B}}$	Ω/tick	MΩ/s
MTP-Argon2 [9]	SRAM full array		631	395.31	–	**123.93**	**3,718.0**
	SRAM half array		1401	178.09	–	117.13	3,513.9
	Parallel search	$N_{\parallel} = 40$	35213	–	55.81	55.81	1,674.2
		in DRAM	14085	–	22.32	22.32	669.7
	Dinur-Nadler attack	$c_{\mathtt{DN}} = 68.4$	43160	5.79	–	5.79	173.8
		$c_{\mathtt{DN}} = 116.2$	73322	3.41	–	3.41	102.3
Itsuku, Section 4	SRAM full array		15	14231.97	–	**5213.42**	**156,402.6**
	SRAM half array		2521	53.57	–	65.10	1,953.1
	Parallel search	in DRAM	145	–	9.67	9.67	290.0

Table 7: Hypothetical 100 billion transistor chip by 2026 ($\mathcal{G} = 25.0$ GGE, $\mathcal{B} = 3.0$ TB/s)

Hardware Efficiency We analyse the performance of the PoW variants on the increasing area and bandwidth hardware. For the MTP-Argon2 PoW, the bandwidth bottleneck and small search state size favor the transposed search. It is beaten by the half-array compression implementation once it fits on die, and finally by the on-die full array, which both assume a very high on-die bandwidth. The efficiency can be mesured as the best hardware performance in MΩ/s divided by the area budget in billion transistors. For MTP-Argon2 implementations the efficiency per increasing budget is 10.58, 9.67, 16.75 and 37.18. For Itsuku, the direct implementation is always the best one, thanks to its design with a large search state size and number of dependencies between array elements. The hardware efficiency is 117.3, 130.4, 223.6 and 1564.0. There is a large efficiency improvement once the array fits on die, hence the necessity that it should not be a fixed size.

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 rewards 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. Thus PoW schemes impervious to FPGA or ASIC implementations are sought, focusing in particular on memory-bound approaches. Memory boundness has been defined in term of latency [3, 15, 4, 11], bandwidth [19] or size [22, 9]. The standard approach is to combine hashing and memory access in some pseudo-random way. We discuss here two particular schemes: the CryptoNight [23] and Wild Keccak [10] hash functions.

6.1 CryptoNight

The CryptoNight hash function [23] proposed for CryptoNote [24] builds a 2 MiB scratchpad of pseudo-random data with numerous read/write operations and hashes the result to get the final

value, combining Keccak [7] (a.k.a. SHA-3), repeated AES simple round iterations, XOR, the Keccak permutation, and one of BLAKE, Groestl, JH and Skein hash functions. The choices of the various parameters (*e.g.* 2^{19} iterations) is not justified, nor the apparent over-complication with multiple and partial cryptographic primitives. A possible insentive to chose a number of different hash functions is that a hardware implementation would require them all. However, as some hash functions are seldom used, it would be efficient just to implement the frequently used ones in hardware and keep the others in software, so this explanation is not convincing. 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. An analysis of the complexity, cost and of potential performance bottlenecks would have been 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: CryptoNight is certainly an expensive hash function 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 expensive for ASIC, it is still doable, especially as technology improves.

6.2 Wild Keccak

The Wild Keccak hash function [10] has been proposed for the Boolberry project. It aims at reducing 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 is a modification of Keccak [7] 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-byte blocks 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-Argon2 and Itsuku.

All proposed schemes but Itsuku 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-Argon2 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 build trust; 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 that lack precise and extensive cryptographic arguments and justifications. Although this does not mean that they are weak, providing such discussions and choosing 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 hundreds for MTP-Argon2 and Itsuku when counting Merkle tree opening checks. 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-Argon2 and Itsuku.

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 approach relies on the Merkle tree proof to only convey part of the array, which is thus not fully needed for verification. As a consequence proofs are significantly larger for MTP-Argon2 and Itsuku, from 186 KiB down to 11 KiB. 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 that proposals vary so widely on that point: CryptoNight deems 2 MiB as enough, Wild Keccak starts with about 100 MiB, and MTP-Argon2 requires 2 GiB. We think that even more is needed to provide enduring resistance to the scheme.

7 Conclusion

We bring a new memory-hard PoW based on [9] that attempts to counter known attacks [14, 8] on the scheme. Key contributions, which are steps toward a better memory-hard PoW scheme, include:

- a new parallel search attack on MTP-Argon2 which can achieve good performance under limited bandwidth condition.
- a refinement of the evaluation of the Dinur-Nadler attack costs, with precise closed formulae.
- to make all primitives (F, H) depend on challenge I and based on a cryptographic hash function.

  Even if the Dinur-Nadler attack is not practical, such a dependency ensures that any pre-computation-based attack would have to be specific to the challenge, which is a good property.
- to consider using a larger number of smaller elements for array X, thus reducing the proof size and making a parallel search uninteresting once $x = S$.
- to enlarge the search state size per nonce, and a way to do so with a back sweep hashing on intermediate search hashes, so as to increase the memory requirement of threaded hardware implementations.
- to consider starting from a larger than 2 GiB array and include a way to increment its size as hardware capabilities evolve, for instance by linking array size to PoW difficulty.
- to build array X in parallel, making the scheme more progress free.
- to use a constant array X if possible for the use case, allowing a small PoW, as the whole MTP part is avoided.
- to reduce the hash size for the Merkel Tree depending on d, thus the proof size.
- parameters chosen based on a clear security criteria: that any implementation that requires half the memory or less should endure a 64-fold computation cost for conservative difficulty $d \leq 100$.
- 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, although CPU and GPU software implementations are expected to be computation-bound.

- a reference implementation for Itsuku [13] under development on Github.

We relied on numerical simulations to evaluate various costs, *e.g.* the recomputation costs presented in Table 2. The results are highly sensitive to the choice of ϕ functions. Of course, the implementations used may have bugs that could change the resulting figures significantly. The evaluation codes used are available online [13].

PoW functions are ecological hazards [21]: avoid them if you can.

Thanks

We would like to thank Itai Dinur, Niv Nadler, Alex Biryukov, Dmitry Khovratovich, Ling Ren, Claude Tadonki, Juan Agustín Rodriguez and Mark Bevand for discussions and feedback; Bahamut, Haroun Beltaïfa and Pierre Jouvelot for proofreading; Wolfram Alpha for partial help with some formulae; Doubloon Skunkworks and Zcoin for support.

References

[1] Bitcoin Block Height 488101. `https://blockchain.info/block-height/488101`, October 2017.

[2] BLAKE2 – fast secure hashing, November 2017. `https://blake2.net/`.

[3] 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.

[4] 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.

[5] 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`.

[6] 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`.

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] Baptiste Colin, Arnaud Larroche, and Fabien Coelho. Itsuku Reference Implementations, November 2017. `https://github.com/baptistecolin/itsuku/`.

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

[15] 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.

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

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

[18] 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.

[19] Ren Ling and Srinivas Devadas. Bandwidth Hard Functions for ASIC Resistance. IACR 225, MIT, Cambridge, MA, March 2017. `https://eprint.iacr.org/2017/225.pdf`.

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

[21] Karl J. O'Dwyer and David Malone. Bitcoin Mining and its Energy Footprint. In *Irish Signals & Systems Conference (ISSC) and China-Ireland International Conference on Information and Communications Technologies (CIICT)*, pages 280–285, Limerick, Ireland, June 2014. IET.

[22] 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`.

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

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

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