Time-Memory Tradeoff Attacks on the MTP Proof-of-Work Scheme[⋆]

Itai Dinur and Niv Nadler

Department of Computer Science, Ben-Gurion University, Israel

Abstract. Proof-of-work (PoW) schemes are cryptographic primitives with numerous applications, and in particular, they play a crucial role in maintaining consensus in cryptocurrency networks. Ideally, a cryptocurrency PoW scheme should have several desired properties, including efficient verification on one hand, and high memory consumption of the prover's algorithm on the other hand, making the scheme less attractive for implementation on dedicated hardware.

At the USENIX Security Symposium 2016, Biryukov and Khovratovich presented a new promising PoW scheme called MTP (Merkle Tree Proof) that achieves essentially all desired PoW properties. As a result, MTP has received substantial attention from the cryptocurrency community. The scheme uses a Merkle hash tree construction over a large array of blocks computed by a memory consuming (memory-hard) function. Despite the fact that only a small fraction of the memory is verified by the efficient verification algorithm, the designers claim that a cheating prover that uses a small amount of memory will suffer from a significant computational penalty.

In this paper, we devise a sub-linear computation-memory tradeoff attack on MTP. We apply our attack to the concrete instance proposed by the designers which uses the memory-hard function Argon2d and computes a proof by allocating 2 gigabytes of memory. The attack computes arbitrary malicious proofs using less than a megabyte of memory (about 1/3000 of the honest prover's memory) at a relatively mild penalty of 170 in computation. This is more than 55,000 times faster than what is claimed by the designers. The attack requires a one-time precomputation step of complexity 2^{64}, but its online cost is only increased by a factor which is less than 2 when spending 2^{48} precomputation time.

The main idea of the attack is to exploit the fact that Argon2d accesses its memory in a way which is determined by its previous computations. This allows to inject a small fraction of carefully selected memory blocks that manipulate Argon2d's memory access patterns, significantly weakening its memory-hardness.

Keywords: Cryptocurrency, proof-of-work, Merkle Tree Proof, memory-hard function, Argon2, time-memory tradeoff, cryptanalysis.

[⋆] This research was supported in part by the Israeli Science Foundation through grant No. 573/16 and by the Lynn and William Frankel Center for Computer Science.

1 Introduction

Proof-of-work (PoW) schemes were introduced by Dwork and Naor [9] as a computational technique to combat junk mail, or more generally, to limit access to a shared resource. The main idea is to require the user to compute a moderately hard function in order to gain access to the resource, thus preventing excessive use. Since their introduction, many PoW schemes have been proposed, and they have recently become a very popular area of research with the rise of Bitcoin [14] and cryptocurrencies in general.

In the cryptocurrency setting, PoWs play a major role in maintaining consensus among cryptocurrency network nodes about the state of the distributed blockchain ledger. The proofs are generated by miners, each proof computed over a block of recent transactions. A new transaction block propagates through the cryptocurrency network and its PoW is verified by the nodes which update their local view of the blockchain accordingly.

Since verification is performed by every node in the network in order to check that new transaction blocks are valid, slow (or resource consuming) verification could expose the nodes to denial of service attacks, and in addition, increase the risk of forks (inconsistent state among the nodes). Therefore, one of the most important properties for a PoW scheme in the cryptocurrency setting is *efficient verification* of proofs. At the same time, computing proofs should ideally be as efficient (in terms of cost) on general CPUs as it is on custom designed application-specific integrated circuits (ASICs). This is desirable to combat centralization, where the majority of mining is performed by centralized mining ASIC rigs, as it is today for Bitcoin. Indeed, mining centralization in Bitcoin is a direct result of Bitcoin's SHA-256-based ASIC-friendly PoW scheme. This concentration of power is considered by many as contradictory to Bitcoin's philosophy.

One of the main ideas to combat mining centralization is to use *memory-hard functions*. Such functions require a large amount of memory to compute and pose substantial computational penalties on algorithms that attempt to compute them with less memory. The use of memory-hard functions aims to diminish the advantage of ASICs over standard PCs, as memory-intensive operations are not much more efficient on dedicated hardware compared to general CPUs. In this context, we also mention the notion of a proof-of-space (formulated by Dziembowski et al. in [10] and independently by Ateniese et al. in [3]) that offers similar security guarantees as memory-hard functions in PoW schemes. The main conceptual difference is that an honest prover in a proof-of-space scheme generally does not have to perform a computational task (besides allocating some space), whereas in a PoW scheme based on a memory-hard function, an honest prover is required to execute some non-trivial computation besides memory allocation.[1] In the cryptocurrency setting, the difficulty of this computation is fine-tuned

[1] Additionally, the protocol between the prover and verifier in [10] was defined as interactive and hence unsuitable to cryptocurrencies (but this is a superficial restriction that can be lifted).

to keep the rate at which new transaction blocks arrive on the network stable (typically, every few minutes).

The most popular memory-hard function among cryptocurrencies (such as Litecoin [11]) is scrypt [16]. However, despite its memory hardness, a noticeable shortcoming of scrypt is that tuning it to use substantial memory does not provide efficient verification. As a result, scrypt (as used by Litecoin) does not consume substantial memory and thus has been shown to have a very efficient hardware implementation [8].

Very recently, at the USENIX Security Symposium, 2016, Alex Biryukov and Dmitry Khovratovich presented the MTP (Merkle Tree Proof) PoW scheme [7]. MTP is claimed to offer essentially all properties[2] desired from a cryptocurrency PoW scheme, including fast verification and ASIC-resistance. As a result, it has received substantial attention from the cryptocurrency community.

MTP uses a design that combines a memory-hard function with a Merkle hash tree [13]. This design resembles the one of [12] that also proposes to build a Merkle hash tree on top of data computed by a memory-hard function (in the form of a graph with high pebbling complexity). The MTP design is also related to the proof-of-space scheme proposed in [10] (that uses an interactive proof protocol).

The memory-hard function used by MTP (denoted by $\mathcal{F}$) receives as an input a seed I and computes an array of T blocks $X[1], X[2], \ldots, X[T]$ (for a parameter T).[3] Each block $X[i]$ is generated using the internal compression function F of $\mathcal{F}$, which takes $X[i-1]$ and $X[\phi(i)]$ as inputs. The function $\phi(i)$ is an *indexing function* (defined for $\mathcal{F}$) that outputs a block index in the range $[1, \ldots, i)$.

The prover's algorithm in MTP has two phases, where in the first phase it computes $X[1], \ldots, X[T]$ using $\mathcal{F}$ from the input seed, and stores these blocks in memory. In addition, it computes a Merkle hash tree with root Φ over $X[1], \ldots, X[T]$, effectively committing to the array's value. In the second phase, the algorithm picks an arbitrary nonce N, and accesses L blocks (for a parameter L) of $X[1], \ldots, X[T]$ at pseudo-random locations in a sequence $X[i_1], X[i_2], \ldots, X[i_L]$. In this sequence, each i_j effectively depends on Φ, N and on the values of the previous blocks in the sequence. Finally, a value Y_L (which depends on all blocks in the sequence) is computed and tested against the difficulty level of the scheme (Y_L needs to have d trailing zeros for a parameter d). Assuming the test passes for some nonce N, then $(\Phi, N, \mathcal{Z})$ is a valid proof, where $\mathcal{Z}$ contains the *openings* of $2L$ blocks $\{X[i_j - 1], X[\phi(i_j)]\}$. An opening of a block $X[i]$ is evidence that it is a leaf in the Merkle hash tree at position i. Each opening contains internal node values that allow to calculate the path from the block $X[i]$ to the Merkle hash tree root Φ.

[2] We only mention here the few desired properties of PoW schemes which are relevant for this paper. For a more comprehensive list, refer to [7].

[3] Blocks in the array computed by $\mathcal{F}$ should not be confused with transaction blocks in the blockchain.

Given a proof $(\Phi, N, \mathcal{Z})$, the verifier checks that the openings $\mathcal{Z}$ are valid (namely, lead to Φ), reproduces all $X[i_j] = F(X[i_j - 1], X[\phi(i_j)])$, and verifies that Y_L passes the difficulty filter.

Biryukov and Khovratovich proposed a concrete instantiation of MTP with the memory-hard function Argon2 [6] (see Figure 1), the winner of the Password Hashing Competition [15]. Argon2 has two variants, Argon2i and Argon2d, where the first uses *data-independent indexing* while the latter uses *data-dependent indexing*. In a function with data-dependent indexing, the indexing function $\phi(i)$ depends on previously computed blocks, whereas in a function with data-independent indexing it does not. The proposed MTP instantiation uses Argon2d as $\mathcal{F}$, where $\phi(i)$ depends on the value of the previously computed block $X[i-1]$. Argon2d was shown to offer somewhat better resistance to computation-memory tradeoffs compared to Argon2i (refer to [6] for details).

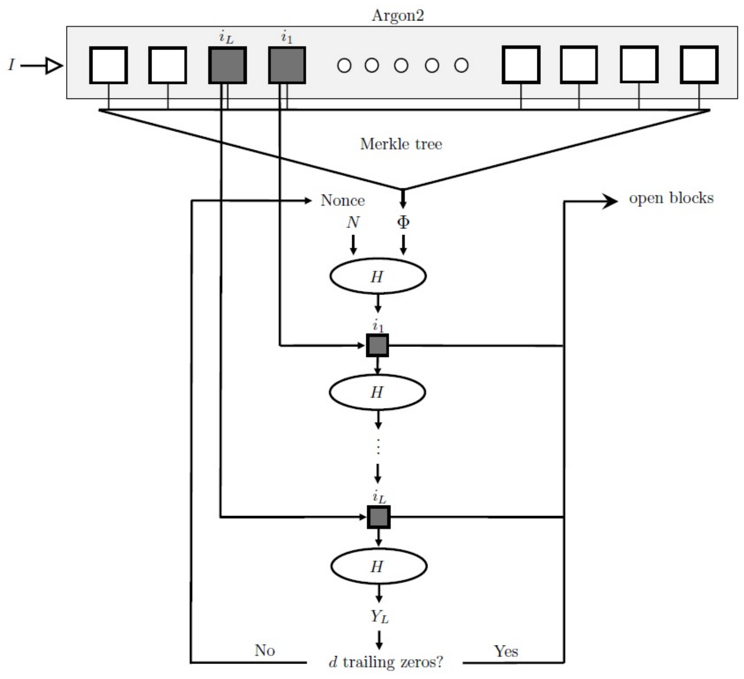

Fig. 1. The MTP PoW Scheme

The concrete instantiation of MTP requires 2 gigabytes of memory and computes a proof shorter than a megabyte. Moreover, its verification algorithm requires only several hundreds of hash computations, and thus MTP provides efficient verification. MTP is memory-hard, assuming that the underlying function Argon2d is such. However, the combination of these two advantages comes at a cost, since the verification algorithm only checks part of the memory computed by the prover. This opens the door for a new type of attack on MTP, where a cheating prover computes a *malicious proof* that passes the verification algorithm, yet the computed block array contains some entries which are inconsistent with the Argon2d compression function. More specifically, there exist indexes i for which $X[i] \neq F(X[i-1], X[\phi(i)])$.

In the cryptocurrency setting, it is likely that malicious proofs will eventually be detected by fully validating nodes (which store the full blockchain) that perform additional checks on new transaction blocks beyond running the simple verification algorithm. However, typically (as in Bitcoin) most nodes in the network are SPV (Simple Payment Verification) clients that are interested in a small number of particular transactions, and only execute the efficient verification algorithm in order to check the validity of new transaction blocks. Hence malicious proofs that propagate through the network may cause it to enter an inconsistent state (namely, to fork) and result in denial of service, depending on the rate at which they arrive. While the consequences of malicious proofs depend on the specific cryptocurrency, it is important that they are not much easier to generate (in terms of cost) than honest proofs, as otherwise, attackers will have additional incentives to compute them. More specifically, it needs to be shown that MTP is *immune to cheating strategies.*

Assume that a cheater computes a malicious proof that passes verification, but is inconsistent with the internal Argon2d compression function F on some fraction ϵ of the block array $X[1], \ldots, X[T]$. Roughly speaking, the MTP designers claim that for a large ϵ, the cheater cannot efficiently produce a consistent sequence of blocks $X[i_1], X[i_2], \ldots, X[i_L]$ of length $L = 70$. More precisely, since ϵ is large, the cheater will pay a large computational penalty (of roughly $(1-\epsilon)^{-70}$) in order to produce $X[i_1], X[i_2], \ldots, X[i_{70}]$, where all blocks $X[i_j]$ are consistent ($X[i_j] = F(X[i_j - 1], X[\phi(i_j)])$) and all pairs $\{X[i_j - 1], X[\phi(i_j)]\}$ have valid openings leading to Φ. On the other hand, when ϵ is small, $X[1], \ldots, X[T]$ are almost always consistent with F, and hence the cheater has to allocate essentially as much memory as an honest prover and there is no efficiency gain.

In this paper, we analyze the immunity of MTP against cheating strategies. Our main result shows how to compute a malicious proof much more efficiently than claimed by the designers, obtaining a sub-linear computation-memory tradeoff algorithm for a surprisingly small amount of memory. In particular, when applied to the proposed MTP instance, our attack computes a malicious proof using a fraction of about 1/3000 of the honest prover's memory (which is less than a megabyte), at a relatively mild penalty of about 173 in computation complexity. On the other hand, according to the analysis of the authors for our parameters, the cheater's computation complexity should increase by a significant factor of about 10 million. Hence our attack improves upon the analysis of [7] by a factor of more than 55,000.

The metric which is used in [7] to evaluate the computational cost of implementing an algorithm in ASICs is the *time-area product.* In this metric, the algorithm's memory requirements translate into area in hardware. Using the time-area product metric, our attack costs less than the honest prover's algorithm by a multiplicative factor of 113, improving on the analysis of [7] (in which the cheater's algorithm is more expensive by a factor of 294) by more than 33,000.

As noted above, the consequences of our attack on a cryptocurrency using MTP with Argon2d are hard to predict. However, since a cheating miner has a computational advantage of about 113 over honest miners, then such a miner can potentially overwhelm the network with malicious proofs even when possessing only a small fraction of the total computation power.

To explain the weakness of MTP, we view the computation of Argon2d as a directed acyclic graph, whose vertices are Argon2d's array blocks indexes and a directed edge connects vertex j to i if $j = i - 1$ or $j = \phi(i)$ (namely, $X[j]$ is used in computing $X[i]$). After the prover commits to the hash values of the graph vertices by computing the Merkle hash tree root, MTP challenges the prover to compute hash values of randomly chosen vertices (namely random block values). The goal of an attacker (a cheating prover) is to store only a small fraction of the vertex hash values, but still answer the challenges quickly. The ability to quickly answer random challenges on such a hash graph with limited storage strongly depends on the graph's structure. Indeed, Argon2d's graph seems to offer a very steep computation-memory tradeoff curve that forces the attacker to spend significant computation when storing a small fraction of the hash values. However, recall that since MTP does not challenge the prover on all hash values of the graph nodes, an attacker can cheat and compute a small fraction of inconsistent hash values ($X[i] \neq F(X[i-1], X[\phi(i)])$) with substantial probability of not being caught. Moreover, since Argon2d uses data-dependent indexing, then the structure of the graph depends on hash values of its own vertices. Therefore, the attacker can exploit the inconsistent hash values and compute a manipulated graph with a much weaker computation-memory tradeoff curve compared to Argon2d.

The general outline of our attack is identical to the one considered by the MTP designers, namely, we replace Argon2d with a function which is consistent with its compression function on almost all blocks (thus using a small ϵ). However, this highly consistent function can be computed with substantially less memory compared to Argon2d at a modest sub-linear computation-memory tradeoff. This is quite surprising, as it demonstrates that although computation-memory tradeoffs for Argon2d are exponential, manipulating it in a small number of places can drastically reduce its memory-hardness.

The attack works by injecting into Argon2d's block array malicious *control blocks* that exploit in the strong way the data dependency of the indexing function of Argon2d. Control blocks significantly weaken the memory-hardness of (the slightly modified) Argon2d, allowing to compute it with a linear computation-memory tradeoff. However, this approach does not seem to lead to sub-linear computation-memory tradeoffs, which appear out of reach at first sight. Indeed, consider a significantly weakened Argon2d variant, in which we completely eliminate the indexing function component from the compression function and define

$X[i] = F'(X[i-1])$ instead (for some function F').[4] If we store one out of t blocks (for some positive integer t), then a random block falls at expected distance of $(t-1)/2$ from a stored one, and its computation requires $(t-1)/2$ compression function calls on average. Therefore, the computation-memory product is reduced by $(t-1)/2t \approx 1/2$, and we cannot do much better than a linear computation-memory tradeoff even for a weak function. This simple argument shows that storing full blocks is wasteful if we aim for sub-linear computation-memory tradeoffs. To overcome this obstacle, we store a succinct representation of each control block (which is much smaller than a standard block), and these representations are essentially the only data kept in memory (besides a very small fraction of full blocks).

The control blocks in our attack are computed during a one-time precomputation phase, after which proofs can be found for arbitrary inputs. The preprocessing complexity is 2^{64}, which is challenging, yet feasible for an ambitious attacker. We stress that the preprocessing phase is very easy to parallelize and requires less than a megabyte of memory. Moreover, the online complexity of the attack is not drastically reduced when using shorter precomputation. For example, even if we spend a trivial amount of 2^{48} computations during preprocessing, our online attack costs less than the honest prover's algorithm by a factor which is larger than 60.

While the parameters of our attack are optimized for MTP when instantiated with Argon2d, it is applicable to the MTP scheme when instantiated with any memory-hard function that uses data-dependent indexing. The most natural way to avoid the attack is to use a memory-hard function with data-independent indexing (such as Argon2i). Such an instantiation of MTP resists our attack, but interestingly, it still has some undesired properties that we discuss towards the end of this paper.

The rest of this paper is organized as follows. We describe the egalitarian computing framework of [7] in Section 2, while the description and previous analysis of MTP is given in Section 3. An overview of our attack is given in Section 4, its details are described in Section 5, while in Section 6 we analyze the full attack. Finally, we describe extensions of the attack in Section 7, discuss countermeasures in Section 8 and conclude the paper in Section 9.

2 Egalitarian Computing Framework [7]

In this section, we summarize the egalitarian computing framework, as described in [7].

Given a function $\mathcal{H}$, the goal of the attacker is to minimize the cost of computing $\mathcal{H}$ on hardware (ASICs), while keeping its running time close to that of a standard implementation (typically x86). On ASICs, the memory size M translates into a certain area A and the running time T is determined by the

[4] The graph structure of this weakened variant is a hash chain, as the graph structure of scrypt [16].

length of the longest computational chain and memory latency.[5] The cost of the attacker in the framework is measured by the time-area product AT.

Given that the standard implementation of $\mathcal{H}$ consumes M units of memory, the attacker aims to compute $\mathcal{H}$ using only αM memory for $\alpha < 1$. Using a computation-memory tradeoff algorithm specific to $\mathcal{H}$, the attacker has to spend $C(\alpha)$ times as much computation as the standard implementation and his total running time increases by the factor $D(\alpha)$ (where $C(\alpha)$ may exceed $D(\alpha)$ since the attacker can parallelize the computation).

In order to obtain a running time of $T \cdot D(\alpha)$ (despite having to perform $C(\alpha)$ as much computation), the attacker has to place $\frac{C(\alpha)}{D(\alpha)}$ additional cores on chip. Therefore, the time-area product changes from AT to AT_α where

$$AT_\alpha = A \cdot (\alpha + \frac{\beta C(\alpha)}{D(\alpha)}) \cdot T \cdot D(\alpha) = AT(\alpha D(\alpha) + \beta C(\alpha)), \tag{1}$$

and β is the fraction of the original memory occupied by a single computing core. There is additional cost in case of significant communication between the computing cores, but this cost is irrelevant to this paper, as our attack does not use extensive communication.

A function is defined to be *memory-hard* if any algorithm that computes it using M memory units has a computation-space tradeoff $C(\alpha)$ where $C(\cdot)$ is at least a super-linear function of $1/\alpha$.

3 Description and Previous Analysis of MTP [7]

In this section, we provide a brief description of MTP and summarize its preliminary analysis as given in its specification. For more details, refer to [7].

MTP uses as a building block a memory-hard function $\mathcal{F}$ that takes as input a password P (which may be null in the cryptocurrency setting) and a salt S. It fills T blocks of memory $X[1], X[2], \ldots, X[T]$ of a certain size, and then may overwrite them several times. However, in MTP the function $\mathcal{F}$ does not overwrite the memory.

Each block $X[i]$ is generated using the internal compression function F of $\mathcal{F}$, which takes $X[i-1]$ and $X[\phi(i)]$ as inputs, where $\phi(i)$ is an *indexing function* (defined for $\mathcal{F}$) that outputs a block index in the range $[1, \ldots, i)$.

Another building block used in MTP is a Merkle hash tree, which is described in Appendix A. MTP is defined using global parameters T, L and d (difficulty level) and a hash function H. We denote the output size of H in bytes by h.

The prover's algorithm takes as input a challenge[6] I. The output of the algorithm is a proof $(\Phi, N, \mathcal{Z})$, as described below.

[5] Note the distinction between computation complexity which is measured according to the total size of the algorithm's circuit and time complexity which is measured according to the depth of the circuit. In this paper, both complexities are measured in terms of basic function (compression function or hash function) invocations.

[6] In the cryptocurrency setting, I depends on the hash value of the previous transaction block in the blockchain and the transactions included in the current block.

```
1. Compute $\mathcal{F}(I)$ and store the $T$ blocks $X[1], X[2], \ldots, X[T]$ in memory.
2. Compute the root $\Phi$ of the Merkle hash tree over the blocks
   $X[1], X[2], \ldots, X[T]$.
3. Select nonce $N$.
4. Compute $Y_0 = H(\Phi, N)$.
5. For $1 \leq j \leq L$:
       $i_j = Y_{j-1} \pmod{T};$
       $Y_j = H(Y_{j-1}, X[i_j]).$
6. If $Y_L$ has $d$ trailing zeros, then output $(\Phi, N, \mathcal{Z})$ as the proof-of-work,
   where $\mathcal{Z}$ contains the openings (defined in Appendix A) of $2L$ blocks
   $\{X[i_j - 1], X[\phi(i_j)]\}$. Otherwise, go to Step 3.
```

In the following, we refer to Step 5 as computing a *chain* of values. The prover's algorithm requires T blocks of memory (in addition to a small amount of memory, required to compute the Merkle hash tree) and its running time is about $T + 2^d \cdot L$. The generated proof size is dominated by $\mathcal{Z}$ and is slightly more than $2L$ blocks in addition to $2L \cdot \log(T) \cdot h$ bytes for the openings.

The verifier's algorithm is given a proof $(\Phi, N, \mathcal{Z})$ as input and it outputs 'Yes' if the proof is valid, and otherwise 'No'.

```
1. Verify all block openings $\mathcal{Z}$ using $\Phi$.
2. Compute $Y_0 = H(\Phi, N)$.
3. Compute from $\mathcal{Z}$ for $1 \leq j \leq L$:
       $i_j = Y_{j-1} \pmod{T};$
       $X[i_j] = F(X[i_j - 1], X[\phi(i_j)]);$
       $Y_j = H(Y_{j-1}, X[i_j]).$
4. If $Y_L$ has $d$ trailing zeros, output 'Yes', otherwise output 'No'.
```

3.1 Previous Tradeoff analysis of MTP [7]

We summarize the previous tradeoff analysis of MTP, as described in its specification.

Denote by M the total amount of memory consumed by a standard implementation of the MTP prover's algorithm. The analysis of [7] aims to deduce the function $C'(\cdot)$ for MTP, given the function $C(\cdot)$ for the underlying memory-hard function $\mathcal{F}$. This analysis is divided into several possible cheating strategies.

Memory savings: A memory-saving prover can use αM memory for $\alpha < 1$. In [7] it is asserted that the computation penalty of such a prover is increased by $C'(\alpha) = C(\alpha)$ to $C(\alpha)(T + 2^d \cdot L)$.

Block modification: A cheater can compute a different function $\hat{\mathcal{F}} \neq \mathcal{F}$ by computing inconsistent intermediate blocks $X[i] \neq F(X[i-1], X[\phi(i)])$. If the fraction of inconsistent blocks is ϵ, then the probability that only consistent blocks are accessed in the second phase of the prover's algorithm (namely, during the L block computation starting from Φ) is

$$\gamma = (1-\epsilon)^L.$$

Thus, the authors of [7] conclude that the cheater's time is increased by the factor $1/\gamma$.

Overall cheating penalties: A cheater can use both strategies above by storing only αT blocks and additionally allowing ϵT inconsistent blocks. According to [7], it is possible to combine to results from the analysis strategies above and conclude that the cheater makes at least

$$\frac{C(\alpha+\epsilon)(T + 2^d \cdot L)}{(1-\epsilon)^L} \tag{2}$$

calls to the compression function F. This gives $C'(\alpha+\epsilon) = \frac{C(\alpha+\epsilon)}{(1-\epsilon)^L}$ for a cheating prover.

Parallelism Both the honest prover and the cheater can parallelize the computation for different nonces N. According to [7], the latency $D'(\cdot)$ of the cheater's computation for MTP will be

$$D(\alpha+\epsilon). \tag{3}$$

3.2 Instantiation of MTP

Biryukov and Khovratovich propose to instantiate MTP with the memory-hard function Argon2d [6] and the hash function Blake2 [4].

Argon2d uses a block size of 1 KB. Its T blocks are arranged in a matrix, where in MTP it is of size 4×4. A row of a matrix is called a *lane*, while a column is called a *slice*. Each of the 16 matrix entries is called a *segment*. Segments of the same slice (column) are computed in parallel, and may not reference blocks from each other. All other blocks can be referenced.

Below we describe the indexing function of Argon2d according to the 2-dimensional matrix notation. However, throughout most of this paper, we will index the memory of Argon2d as a single dimensional array for the sake of simplicity, as done in the MTP algorithm specification (the translations of indexes from the single to the 2-dimensional case and and vise-versa are straightforward).

The indexing function of Argon2d for a block with index $[i][j]$ (where $i \in \{0,1,2,3\}$, $j \in \{0,1,\ldots,T/4\}$ refer to lane and block index in the lane, respectively) is defined using the value of the previous block in the same lane $X[i][j-1]$. The value of the 32 least significant bits (LSBs) of this block is denoted by $J_1 \in \{0,\ldots,2^{32}-1\}$ and value of the next 32 bits is denoted by J_2.

If block $X[i][j]$ is in the first slice, then the lane (row) number of the indexing function value (which is denoted by l) is set to the current lane index $l = i$. Otherwise, the 2 LSBs of J_2 determine l.

Next, we compute the value of ϕ, $[l][z]$, using i, j and J_1. The details of this computation are not important for the rest of this paper and are given in Appendix B for the sake of completeness. We only note that the index z is computed using a simple function that defines a non-uniform distribution (according to the randomness of J_1) over a prefix of blocks in lane l, where more weight is placed on the larger indexes (closer to j).

The parameters of MTP selected in [7] are $T = 2^{21}$ (hence the prover's algorithm requires $2^{21} \cdot 2^{10} = 2^{31}$ bytes of RAM, or 2 gigabytes) and $L = 70$. The output of hash function Blake2 is truncated to 128 bits, and thus $h = 16$. Note that the difficulty level d is determined by the application (typically a cryptocurrency).

Previous Tradeoff Analysis for the MTP Instantiation The tradeoff analysis for the concrete instantiation above relies on the functions $C(\cdot)$ and $D(\cdot)$ for Argon2d. Some values of these functions at specific points (taken from [6, 7]) are given here[7] in Table 1.

α	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$
$C(\alpha)$	1.5	4	20.2	344	4660	2^{18}
$D(\alpha)$	1.5	2.8	5.5	10.3	17	27

Table 1. Time and Computation Penalties for Argon2d

Relying on the generic MTP tradeoff analysis (summarized in Section 3.1), the authors of [7] plug into Equation 1 the values of $C'(\cdot)$ for MTP (given by Equation 2) and $D'(\cdot)$ (given by Equation 3), obtaining

$$AT_\alpha = AT\frac{\alpha D(\alpha+\epsilon) + \beta C(\alpha+\epsilon)}{(1-\epsilon)^L}. \tag{4}$$

Based on this equation, it is shown that for the concrete instance described above, the time-area product can be reduced by the factor of 12 at most, assuming that each Blake2b core occupies an equivalent of 2^{16} bytes, namely $\beta = 2^{16}/2^{31} = 2^{-15}$.

4 Overview of the Attack on MTP

In this section, we give a general overview of our improved tradeoff analysis for MTP, while pointing our where the previous analysis of [7] fails.

[7] These are the best known tradeoff parameters at the time of writing.

4.1 A Trivial Attack

We start by describing a trivial attack on MTP and a simple fix which avoids this attack with very little overhead. The attack is based on the observation that the verifier does not actually check that the proof corresponds to the particular challenge I. As a result, a cheating prover can simply replay a proof for some old challenge I' (generated for the same difficulty level d), and this proof would pass verification for the current challenge I as well. Note that in the cryptocurrency setting, storing the previously generated proofs does not prevent this attack, since not all previously generated (valid) proofs are included in the blockchain, or even reach all the nodes in the network.

In order to avoid this simple attack, we can add to the proof the opening of the first block $X[1]$. The verifier would then compute $X[1]$ directly from the challenge and check this additional opening with little added cost.

In the following, we assume that this simple countermeasure is implemented and focus on attacks which are less trivial and more difficult to counter.

4.2 Weaknesses of MTP

We describe two related weaknesses of MTP that will be exploited in our attack. These weaknesses are not specific to the use of Argon2d in MTP, and apply to MTP when instantiated with essentially any function $\mathcal{F}$ that uses data-dependent indexing.

1. A cheating prover is allowed to modify blocks so that they are inconsistent with the compression function F. Thus, the cheater may compute a function which is completely different than $\mathcal{F}$. When the compression function F uses data-dependent indexing (such as in Argon2d), the cheater can inject (potentially very few) inconsistent blocks that influence its indexing function, weakening the computation-memory tradeoff resistance of (the modified) $\mathcal{F}$, which we denote by $\hat{\mathcal{F}}$. We conclude that it is non-trivial to relate the memory-hardness of $\mathcal{F}$ to the memory-hardness of its potentially modified variant $\hat{\mathcal{F}}$ computed by the cheater to obtain a proof for MTP. More specifically, the main flaw in the analysis of [7] is in equations 2 and 3, where the tradeoff functions $C(\cdot)$ and $D(\cdot)$ for $\mathcal{F}$ are used to compute the overall cheating penalties. However, the (potentially much weaker) tradeoff functions for $\hat{\mathcal{F}}$ should have been used instead.
2. The cheater can use preprocessing (which is independent of a challenge I) to speed up the online computation that begins once a challenge I is received. At first sight, it may not be clear how the cheater can benefit from preprocessing, as the function $\mathcal{F}$ has to be applied online to a challenge I whose value cannot be predicted in advance, and computations of $\mathcal{F}$ with different pseudo-random challenges are generally unrelated (especially when $\mathcal{F}$ uses data-dependent indexing). However, recall that the cheater can manipulate the function $\mathcal{F}$ and compute a different function $\hat{\mathcal{F}}$ both in preprocessing and online. In our attack, we show how to carefully choose $\hat{\mathcal{F}}$ such that the

preprocessing computation is made independent of the online challenge I, but nevertheless reduces the online complexity of computing a valid proof for MTP on arbitrary challenges.

4.3 General Description of the Attack

In this section, we show how to exploit the first weakness described above in order to obtain an efficient computation-memory tradeoff for MTP. The attack is mostly independent of the specification of the compression function F of Argon2d, but makes use of its data-dependent indexing function $\phi(i)$, which depends on $X[i-1]$. Hence, the algorithm of the attack should be adjusted when applied to MTP instantiated with $\mathcal{F}$ that has a different data-dependent indexing function.

The main idea of the attack is to compute a function $\hat{\mathcal{F}}$ that has weaker computation-memory tradeoff resistance compared to $\mathcal{F}$, yet the number of consistent blocks in $\hat{\mathcal{F}}$ (with respect to the compression function F of $\mathcal{F}$) remains relatively high.

The computation of an arbitrary block $X[i+1]$ depends on $X[i]$ and $X[\phi(i+1)]$, where for Argon2d $\phi(i+1)$ depends on $X[i]$. Each one of these two blocks depends on two other previous blocks and so forth. If we store only a small fraction α of the T blocks, then computing $X[i+1]$ will require computing the hash labels (block values) for a graph of blocks with in-degree 2 of size $C(\alpha)$, which seems to be exponential in $1/\alpha$ as shown in Table 1. However, if we allow some inconsistent blocks, we can store a small subset of blocks (denoted by S) in memory and manipulate $\phi(i+1)$ by changing the value of $X[i]$ such that $X[\phi(i+1)] \in S$. Proceeding the same way, we manipulate all blocks at even indexes so that for each such block, its successor is computed using the manipulated block and another block in S (stored in memory). Unfortunately, this strategy leads to a large fraction of inconsistent blocks $\epsilon \approx 1/2$, and results in a major penalty in the second phase of the proof computation, which is roughly[8] $1/(1-\epsilon)^L$, as described in Section 3.1.

To obtain a more efficient attack we extend the above approach by looking at $X[i+2]$ and noticing that $X[\phi(i+2)]$ depends on $X[i+1]$, which in turn, depends on $X[i]$. Therefore, we can try to compute a value for $X[i]$ such that both conditions $X[\phi(i+1)] \in S$ and $X[\phi(i+2)] \in S$ are satisfied. This strategy reduces the fraction of inconsistent blocks to $\epsilon \approx 1/3$, which is smaller than $1/2$, but still results in a significant penalty in the second phase of the proof computation.

Generalizing the above idea further, let t be a small integer (our concrete attack uses $t = 20$). We partition the T computed blocks into about T/t consecutive intervals of size t (for the sake of simplicity we assume that t divides T). The first block $X[i]$ in each interval satisfies $i \equiv 0 \pmod t$ and we write it as $i = s \cdot t$ for a positive $s < T/t$. Such blocks are called *control blocks* and they

[8] The proposed instance of MTP uses $L = 70$, and hence the penalty is about $1/(1-\epsilon)^L \approx 2^{70}$.

serve two purposes simultaneously: first, they will be the only blocks stored in memory (up to small modifications described later), and hence they make up the set S. Consequently, a control block with index s serves as a "target" for array blocks which belong to intervals with indexes larger than s, thus allowing to recompute them using little memory. Second, besides serving as a target for array blocks in later intervals, each control block ensures that the large hash label graph for each of the blocks in its own interval collapses from expected exponential size in $1/\alpha$ to a linear size. Next, we describe how this is achieved.

Control block $X[s \cdot t]$ is computed such that all $t-1$ conditions

$$X[\phi(s \cdot t + 1)] \in S_s, X[\phi(s \cdot t + 2)] \in S_s, \ldots, X[\phi(s \cdot t + t - 1)] \in S_s$$

are satisfied (see Figure 2), where S_s contains all the previously computed control blocks (namely, $S_s = \{X[s' \cdot t] \,|\, 0 \le s' \le s\}$). We slightly relax the $t-1$ conditions above for s and $\ell \in \{1, 2, \ldots, t-1\}$ as follows

$$X[\phi(s \cdot t + \ell)] \in S_s \cup \{X[s \cdot t + 1], \ldots, X[s \cdot t + \ell - 1]\}. \tag{5}$$

In other words, we allow $\phi(s \cdot t + \ell)$ to fall into the range of indexes $s \cdot t + 1, \ldots, s \cdot t + \ell - 1$ (as computing $X[s \cdot t + \ell]$ requires computing earlier blocks in the interval anyway, this relaxation does not increase the complexity of computing $X[s \cdot t + \ell]$).

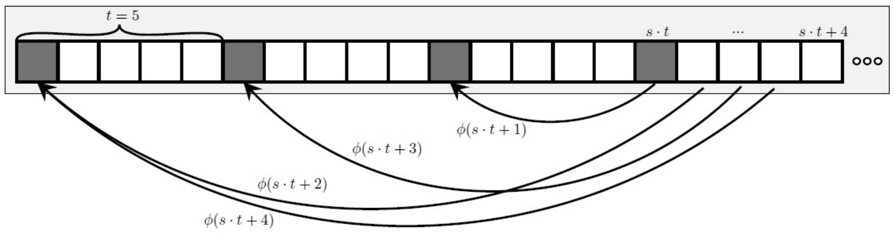

Fig. 2. Satisfying 4 Conditions for $t = 5$

For an integer s and $\ell \in \{1, 2, \ldots, t-1\}$, we have

$$\begin{aligned} & X[s \cdot t + \ell] = F(X[s \cdot t + \ell - 1], X[\phi(s \cdot t + \ell)]) \\ & = F(F(X[s \cdot t + \ell - 2], X[\phi(s \cdot t + \ell - 1)]), X[\phi(s \cdot t + \ell)]) = \ldots \\ & = F(F(\ldots F(X[s \cdot t], X[\phi(s \cdot t + 1)]), \ldots, X[\phi(s \cdot t + \ell - 1)]), X[\phi(s \cdot t + \ell)]). \end{aligned} \tag{6}$$

Given that the conditions of Equation 5 are satisfied for any s and ℓ, then all the above $\ell + 1$ blocks are stored in memory (or fall into the same interval as $X[s \cdot t + \ell]$). Therefore, computing $X[s \cdot t + \ell]$ given only the values of control blocks with indexes $1, 2, \ldots, s$ is performed using at most ℓ calls to the compression function F as claimed (starting from $X[s \cdot t+1] = F(X[s \cdot t], X[\phi(s \cdot t+1)])$ and proceeding according to Equation 6).

The full attack has two phases that correspond to the two phases of the honest prover's algorithm. In the first phase, we compute the control blocks in their natural order and simultaneously compute the Merkle hash tree over all the blocks $X[1], X[2], \ldots, X[T]$. Since the blocks are computed in their natural order, the root Φ of the Merkle hash tree can be computed on-the-fly while keeping in memory the roots of at most $\log(T)$ sub-trees, and joining them (using the hash function H) whenever possible. Overall, the memory complexity of this phase is dominated by the storage of T/t control blocks. Note that only the control blocks are inconsistent with F and hence $\epsilon = 1/t$. In the second phase of the attack, we pick arbitrary values for the nonce N and hope to find a valid proof such that the L blocks in the chain (involved in the computation of $Y_1, \ldots, Y_L$) do not fall onto the inconsistent control blocks (and Y_L has d trailing zeros as required from a valid proof).

5 Details of the Attack

We now describe our basic attack in detail and then extend and optimize it in various ways.

The First Phase We fix a parameter $t > 3$, whose value will be specified later. The input to the first phase of the attack is the challenge I. The output of this phase is an array of control blocks CB (containing the values of about T/t control blocks) and the Merkle hash tree root Φ. In order to comply with the additional restriction imposed by the trivial attack of Section 4.1 (which binds the proof to I), we also compute the first $t-1$ blocks honestly and return them. As t will be very small compared to T (e.g., $t = 20$), this tweak has negligible additional cost in memory. For the sake of simplicity, we omit the details of the Merkle hash tree computation, which it is straightforward and has negligible cost.

1. Compute the first $t-1$ blocks $X[1], \ldots, X[t-1]$ honestly using the compression function F, and store them in memory.
2. Initialize the control block array $CB[1, \ldots, (T/t)-1]$. For each interval $s \in \{1, \ldots, (T/t)-1\}$:
   (a) Initialize the 32 LSBs J_1 of the current control block value $X[s \cdot t]$ and the 2 LSBs of J_2 such that $\phi(s \cdot t + 1) = (s-1) \cdot t$ according to the algorithm of Equation 10 (in Appendix B). This ensures the condition of Equation 5 (for $\ell = 1$), as the next block in the interval $X[s \cdot t+1]$ is computed using the previous control block $X[(s-1) \cdot t]$ (which is stored in $CB[s-1]$).
   (b) Ensure that the remaining $t-2$ conditions of Equation 5 hold using exhaustive search on the control block value $X[s \cdot t]$. Namely, for values of $k = 0, 1, \ldots$ let $X[s \cdot t] = J_1 + J_2 \cdot 2^{32} + k \cdot 2^{64}$, and perform:

i. For $\ell \in \{1, \ldots, t-2\}$, compute

$$X[s \cdot t + \ell] = F(X[s \cdot t + \ell - 1], X[\phi(s \cdot t + \ell)]),$$

where $X[\phi(s \cdot t + \ell)])$ is stored in memory or previously computed in the interval (as the condition of Equation 5 for the current value of ℓ was previously assured to hold).
Using J_1, J_2 for $X[s \cdot t + \ell]$, compute $\phi(s \cdot t + \ell + 1)$ and verify the condition of Equation 5 for $\ell + 1$. If the condition does not hold, choose the next control block value by returning to Step 2.(b). Otherwise, increment ℓ and continue. Once all $t-1$ conditions of Equation 5 hold, continue to the next step.

ii. Store the current $X[s \cdot t]$ value in $CB[s]$, update the Merkle hash tree computation accordingly, and increment s by returning to Step 2.

3. Return the control block array CB, the first $t-1$ blocks $X[1], \ldots, X[t-1]$ and the Merkle hash tree root Φ.

Computational complexity analysis: The computational complexity is dominated by Step 2 which computes about T/t control blocks, all of which satisfy the $t-1$ conditions of Equation 5. The first condition is enforced by Step 2.(a). Note the finding J_1, J_2 is trivial (as Equation 10 in Appendix B is very simple) and this value is not changed in Step 2.(b) as the value of $X[s \cdot t]$ is incremented by multiples of 2^{64}.[9]

Next, we estimate the complexity of Step 2.(b), which depends on the expected number of values of k that we have to try before all remaining $t-2$ conditions are satisfied. If the indexing function ϕ used in Argon2d was uniform over the previous indexes, then the probability for an arbitrary condition for s, ℓ in Equation 5 to be satisfied[10] would be slightly larger than $1/t$. The reason is that at any point in the computation, we store a fraction of (at least) $1/t$ of the blocks computed in the previous intervals, and moreover, we allow ϕ of each index to land in the current interval.

However, as noted in Section 3.2, the indexing function of Argon2d in nonuniform and places more probability weight on larger indexes. Nevertheless, the probability of satisfying an arbitrary condition s, ℓ remains at least $1/t$. This can be easily shown by considering all previous intervals shifted by one (whose index values modulo t are $[1, 2, \ldots, t-1, 0]$). The last block in each such shifted interval is a control block stored in memory and its probability weight is at least as large as the average weight of the interval (the sum of weights divided by t). Since we also allow ϕ of each index to land in the current interval, the probability of satisfying an arbitrary condition for s, ℓ is indeed at least $1/t$.

[9] Since only the 2 LSBs of J_2 determine the lane l, we could also increment $X[s \cdot t]$ by multiples of 2^{34}.

[10] The probability is taken over the choice of J_1, J_2 in $X[s \cdot t + \ell - 1]$, which we can assume to be uniform, given that the compression function F is pseudo-random.

From the analysis above (based on randomness assumptions on the compression function F), we conclude that the probability that all $t-2$ conditions in Step 2.(b) are satisfied for an arbitrary value of k is at least $(1/t)^{t-2}$. Hence we expect to try at most t^{t-2} such values in this step. For each value of k, the expected number of compression function evaluations is about $1+1/t+(1/t)^2+\ldots+(1/t)^{t-2}$, which is very close to 1 for the values of $t \approx 20$ we consider in this paper. We conclude that the expected complexity of Step 2.(b) is about t^{t-2}, and the complexity of the algorithm is about

$$T/t \cdot t^{t-2} = T \cdot t^{t-3}. \quad (7)$$

In Appendix C, we show how to reduce this complexity by a (small) factor of about $8t/(t+8)$ by exploiting the specific Argon2d compression function.

Memory complexity analysis: The efficiency of the attack in the time-area product metric crucially depends on reducing the required storage. The memory complexity of the first phase is dominated by storing T/t control blocks. To save memory, instead of storing each full control block, we simply store the corresponding value of k computed in Step 2.(b). We then easily reconstruct the control block value when we access it in the second phase of the attack. Since we expect to try t^{t-2} values of k, we require about $(t-2)\log(t)$ bits of storage on average per control block. However, we use static allocation of addresses in an array, and some blocks will require more storage than the average.

To overcome this problem, we allocate a small additional allowance of b bits per control block. The probability that these additional bits will not suffice to store k (i.e., we do not find an appropriate k after exhausting all $t^{t-2} \cdot 2^b$ possible values) is about $e^{2^{-b}}$. In our concrete analysis we use $b=5$, which gives a negligible probability of about e^{-32} that the storage for a single block does not suffice.

We conclude that the total memory complexity of the attack in bits is about

$$T/t \cdot (b+(t-2)\log(t)). \quad (8)$$

The Second Phase The input to the second phase of the attack is the output of the first phase, namely, an array of control blocks CB, the Merkle hash tree root Φ and the first $t-1$ blocks $X[1],\ldots,X[t-1]$. Its output is a valid (yet malicious) proof $(\Phi, N, \mathcal{Z})$. The algorithm tries different values for the nonce N until the corresponding chain of values does not access the control blocks (which are the only inconsistent blocks we have) and the last value computed in the chain Y_L has d trailing zeroes.

```
1. For nonce values $N = 0, 1, \ldots$
   (a) Compute $Y_0 = H(\Phi, N)$.
   (b) For $1 \leq j \leq L$:
       i. Compute $i_j = Y_{j-1} \pmod{T}$. Let $i_j = s \cdot t + \ell$, where $s \cdot t$ is the
          index of the control block in the interval of $i_j$. If $\ell = 0$ (namely,
```

i_j is an index of an inconsistent control block), increment N by returning to Step 1. Otherwise, continue.

ii. Compute $X[i_j] = X[s \cdot t + \ell]$, as specified in Equation 6.

iii. Compute $Y_j = H(Y_{j-1}, X[i_j])$ and increment j by going back to Step 1.(b).

(c) If Y_L has d trailing zeros, then output $(\Phi, N, \mathcal{Z})$ as the proof-of-work, where $\mathcal{Z}$ is the opening of $2L$ blocks $\{X[i_j - 1], X[\phi(i_j)]\}$. Otherwise, select another value for N by returning to Step 1.

Computational complexity analysis: To calculate the expected computational complexity, note that we have to compute about 2^d full chains of length L in Step 1.(b) until some Y_L has d trailing zeros. Since $\epsilon = 1/t$, an arbitrary chain extends to length L with probability $(1-1/t)^L$, and we expect to compute $2^d \cdot (1-1/t)^{-L}$ chains in total. As the average length of a chain is t, the algorithm computes an expected number of $2^d \cdot t \cdot (1-1/t)^{-L}$ blocks $X[i_j]$.[11] The cost of computing $X[i_j]$ for $i_j = s \cdot t + \ell$ is at most ℓ, where the expected value of ℓ is $t/2$. This gives a total expected complexity of

$$2^d \cdot t^2/2 \cdot (1-1/t)^{-L}. \tag{9}$$

Memory complexity analysis: The memory complexity remains similar to the previous phase.

5.1 Balancing the Phases

The complexity analysis above shows that there is a tradeoff between the two phases of the attack: larger values of t decrease the value of $(1-1/t)^{-L}$ which is the dominant factor in Equation 9 (for $L = 70$), and hence reduce the complexity of the second phase. At the same time, larger values of t increase the complexity of the first phase (given by Equation 7).

Besides the choice of t, another way to balance the phases is to reconsider the condition $X[\phi(s \cdot t + \ell)] \in S_s$ in Equation 5. This condition is equivalent to $\phi(s \cdot t + \ell) \equiv 0 \pmod t$. Assume we relax this condition (for all values of s, but for a specific value of $\ell > 0$) to

$$\phi(s \cdot t + \ell) \in \{0, 1 \ldots, m_\ell - 1\} \pmod t$$

for some $2 \leq m_\ell < \ell$. This increases the probability of satisfying the condition from about $1/t$ to m_ℓ/t, reducing the expected complexity of phase 1 by a multiplicative factor of m_ℓ. On the other hand, computing $X[s \cdot t + \ell]$ requires computing $X[\phi(s \cdot t + \ell)]$, which may no longer be stored in memory. This increases the average number of compression function evaluations (units) required to compute $X[s \cdot t + \ell]$ by an additive factor of $(m_\ell - 1)/2$ (since $\phi(s \cdot t + \ell)$ is

[11] This computation shows that the analysis of the block modification cheating strategy (given in Section 3.1) is slightly inaccurate, since it does not take into consideration the fact that the expected chain length computed by the cheater is shorter than L.

expected to land in the middle of the allowed interval of m_ℓ blocks modulo t). Moreover, the computation of all values $X[s \cdot t + \ell + 1], \ldots, X[s \cdot t + t - 1]$ also requires $X[s \cdot t + \ell]$ and hence the average computation complexity increases by $(m_\ell - 1)/2$ units for these as well. Overall, the expected computation of a block increases by an additive factor of $((m_\ell - 1)/2 \cdot (t - \ell))/t$. Therefore, it is more efficient to use this approach for large values of ℓ (which are close to the end of each interval).

For example, for $t = 20$ we can set $m_{19} = 2$, reducing the expected complexity of phase 1 by a multiplicative factor of $m_{19} = 2$, while the expected computation of a block increases by an additive factor of $((m_\ell - 1)/2 \cdot (t - \ell))/t = 1/2 \cdot (20 - 19)/20 = 1/40$. If we set $m_{18} = 2$, the expected complexity of phase 1 is also reduced by a multiplicative factor of $m_{18} = 2$, while the expected computation of a block increases by a larger additive factor of $((m_\ell - 1)/2 \cdot (t - \ell))/t = 1/2 \cdot (20 - 18)/20 = 2/40$.

Our optimized attack will use this method by relaxing the conditions for several indexes $\ell, \ldots, t - 1$ at the end of each interval (see Figure 3), assigning them respective values of $m_\ell, m_{\ell+1}, \ldots, m_{t-1}$ (allowing their index functions to land in a larger prefix of a previous interval). To simplify the analysis, we will make sure that all $m_\ell, m_{\ell+1}, \ldots, m_{t-1}$ are smaller than ℓ (which avoids more complex recursions). In total, the expected complexity of phase 1 is reduced by a multiplicative factor of

$$\prod_{j=\ell}^{t-1} m_j,$$

while the expected computation complexity of a block in phase 2 increases by an additive factor of

$$1/2t \cdot \sum_{j=\ell}^{t-1} (m_j - 1)(t - j).$$

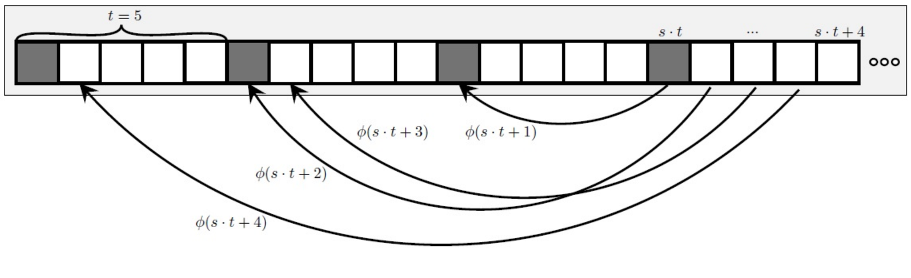

Fig. 3. Relaxing Equation 5 by Setting $m_3 = m_4 = 2$ for $t = 5$

We note that there is also a computational penalty on phase 1 of the attack due to the additional complexity required for calculating a block when exhaustively searching for control blocks. However, in our attack we make sure that there is a penalty only on several blocks at the end of each interval which are rarely computed compared to the other blocks.[12] In total, the overall penalty on phase 1 is negligible for the parameters we choose.

5.2 Using Preprocessing

Although we can select interesting parameters for the attack described above, it is still generally impractical in the standard cryptocurrency setting. The reason for this is that the tradeoff between the two phases forces us to spend considerable time on phase 1 in order to obtain a reasonable overall computation complexity. However, the probability of producing a valid proof in phase 1 is zero, implying that the attack is not a *progress-free* algorithm. As a result, the malicious prover (miner) has to spend significant computation on phase 1 for a certain challenge I with no chance of finding a valid proof in this phase. Once a new proof arrives on the network (which typically occurs every few minutes), this computational effort is lost.

To solve this problem, we modify the algorithm so that the first phase can be performed during preprocessing with no dependency on the challenge.[13] This is achieved by partitioning the Argon2d block array into two parts, where the first part is relatively small and will be computed honestly online using the actual challenge I. The second part contains most blocks in the block array, and we force it to be independent of the first part of the block array and the challenge. In other words, we disconnect the second part from the first, which allows us to perform the heavy control block computation during preprocessing. Details are given below.

We choose a small fraction $\delta \ll 1/t$ and leave the first δT blocks in each of the 4 lanes (rows) of Argon2d with an undetermined value during preprocessing (their value will only be determined online). In order to compute the remaining control blocks during preprocessing with no additional cost, we have to maintain the property that at all stages, each condition given by Equation 5 (for indexes s, ℓ) holds with probability of at least $1/t$. To achieve this, we set the $\delta T/(t-1)$ blocks that follow the δT prefix in each lane to some fixed value. We change the conditions given by Equation 5 by allowing $X[\phi(s \cdot t + \ell)]$ to land on the fixed-value blocks, but do not allow them to land on the undetermined prefix in each lane.

Thus, we start computing the control blocks in phase 1 from index $(1+1/(t-1)) \cdot \delta T$ in each lane. As we cannot compute the root of the full Merkle hash tree in preprocessing, we compute the roots of all the subtrees of the known blocks and pass them to the online phase. Since there are 4 lanes, the number of such

[12] Moreover, computing the blocks in their natural order in phase 1 is more efficient (on average) than computing an average block, as by the time we compute block $X[i]$ we have already computed its predecessor $X[i-1]$.

[13] This modification can be easily combined with the method of balancing the phases described above, but we describe each one of them separately for the sake of simplicity.

roots is at most $4\log(T)$, and they consume negligible memory (which can be reused once the Merkle hash tree computation in phase 2 is finished). Overall, the complexity of phase 1 when performed in preprocessing remains roughly the same.

During the online phase, we receive the challenge I, compute the first δT blocks in each lane honestly and finish the Merkle hash tree computation. We then execute phase 2 of the attack. Assuming that the prefix of size δT is sufficiently small, the memory complexity of the attack will be roughly as in Equation 8. The fraction of inconsistent blocks remains (at most) $\epsilon = 1/t$, hence the computation complexity remains unchanged.

6 Analysis of the Full Attack

In this section, we analyze the full attack. Our goal is to optimize the time-area product ratio for the attack between the honest and malicious provers according to Equation 1. While the previous sections analyzed the attack phases as a function of the parameters in symbolic form in order to maintain generality, we choose to perform the most of the analysis in this section in numerical form. The reason for this choice is that the symbolic expressions that can be derived in this section are rather complex and do not seem to provide additional insight into the attack. Hence, we favor readability over generality.

It is not obvious how to treat the preprocessing phase in the framework of Section 2. On one hand, a one-time precomputation should not be taken into account in Equation 1. On the other hand, a very long preprocessing phase will render the attack impractical. Our compromise is to set a hard limit on the complexity of the preprocessing phase and then optimize the online phase with respect to the framework of Section 2.

Before choosing concrete parameters for the attack in order to optimize the time-area product, we argue informally that it achieves a sub-linear computation-memory tradeoff for a wide range of parameters: recall that the online phase has complexity of $2^d \cdot t^2/2 \cdot (1-1/t)^{-L}$. Allowing for a sufficiently long preprocessing phase, we can choose $t < L$ sufficiently large such that the term $(1 - 1/t)^{-L}$ is small and the complexity becomes roughly $2^d \cdot t^2 < 2^d \cdot L \cdot t$. This is about t times larger than honest prover's complexity (which is more than $2^d \cdot L$). In terms of memory, recall that we store a succinct representation of a $1/t$ fraction of all blocks, whereas the honest prover stores all of them. Consequently, the computation-memory product of the attacker is reduced by a factor which is proportional to c/b, where c is the memory occupied by our succinct block representation and b is the memory occupied by a full block. Namely, the attack achieves a sub-linear computation-memory tradeoff depending on the ratio c/b. Next, we show how to actually obtain such a tradeoff by choosing concrete parameters and applying the optimizations described in the previous section.

We begin by setting a hard limit of 2^{64} for the preprocessing phase. We believe that this is a reasonable complexity, as it is feasible for a motivated attacker, but does not require exceptional computing power that is available only to very resourceful attackers. This feasibility assertion is based on the fact that the preprocessing phase can be easily parallelized (exhaustive search for each control block can be performed in parallel) and requires a small amount of memory (as we show later, it requires less than one megabyte).

We now reconsider the preprocessing complexity of $T \cdot t^{t-3}$ given by Equation 7 (without using the balancing technique of Section 5.1). This complexity can be slightly reduced by a factor of $8t/(t+8)$, as shown in Appendix C. Setting $T = 2^{21}$ as fixed by the MTP instance, we conclude that we can use $t = 14$, which keeps the preprocessing complexity below the hard limit of 2^{64}. However, these parameters do not give the optimal time-area ratio, as they do not use the balancing technique. To optimize the attack, we wrote a computer program that finds the best parameters for the balancing technique of Section 5.1 for a given value of t, limiting the values of all m_ℓ to powers of 2 for simplicity. We then searched for the best value of t exhaustively. Next, we describe the optimal parameters.

6.1 Concrete parameters

We use the balancing technique for $t = 20$ by setting $m_8, m_{11}, \ldots, m_{19}$ to the respective values of $2, 4, 4, 4, 4, 4, 4, 8, 8, 8, 8, 8$. These values reduce the complexity of preprocessing by a multiplicative factor of $\prod_{j=8}^{19} m_j = 2^{28}$ (from $T \cdot t^{t-3} \cdot (t+8)/8t = 2^{21} \cdot 20^{17} \cdot 28/160 < 2^{92}$ to less than 2^{64}). We also note that these values will increase the expected computation complexity of a block in phase 2 by an additive factor of $1/2t \cdot \sum_{j=8}^{19} (m_j - 1)(t - j) = 1/40 \cdot (1 \cdot 12 + 3 \cdot (11 + 10 + 9 + 8 + 7 + 6) + 7 \cdot (5 + 4 + 3 + 2 + 1)) = 6.75$.

Memory complexity: The memory complexity of the honest prover's algorithm is T blocks, and specifically for the MTP instance we have $T = 2^{21}$ blocks which consume 2^{34} bits. According to Equation 8, the memory complexity of the attack is $T/t \cdot (b + (t-2)\log(t))$ bits. However, this is changed by setting $m_8, \ldots, m_{19}$ as above, since computing a block in phase 1 now requires about $20^{18} \cdot 2^{-28} < 2^{50}$ computations (and $50 + b$ bits of storage on average). Using this value, and setting $T = 2^{21}$, $b = 5$ (as specified in Section 5) and $t = 20$, we obtain a total memory complexity of $2^{21}/20 \cdot (50 + 5) \approx 2^{22.45}$ bits, which is less than a megabyte. Hence, the ratio between the memory complexities of the honest and the malicious provers is

$$\alpha \approx 2^{22.45-34} = 2^{-11.55} \approx 1/3000.$$

Computation complexity: Recall that the computation complexity of the honest MTP prover's algorithm is $T + 2^d \cdot L > 2^d \cdot L$. For the MTP instance considered, we have $L = 70$ giving a complexity of at least $2^d \cdot 70$.

The computation complexity of phase 2 according to Equation 9 is $2^d \cdot t^2/2 \cdot (1 - 1/t)^{-L}$. Taking into account the balancing technique, the average computation complexity per block increases by an additive factor of 6.75 (from $t/2 = 10$ to 16.75). For $L = 70$ and $t = 20$, we obtain $2^d \cdot 16.75 \cdot 20 \cdot 36.25 \approx 2^d \cdot 12144$.

Hence, the ratio between the computation complexities of the honest and the malicious provers is

$$C(\alpha) \approx (2^d \cdot 12144)/(2^d \cdot 70) \approx 173.$$

Time-area product ratio: In terms of the time-area product ratio for the attack, according to Equation 1 we should evaluate $\alpha D(\alpha) + \beta C(\alpha)$, where $\beta = 2^{-15}$ for MTP. If we do not use any parallelism, then the total running time ratio is $D(\alpha) = C(\alpha)$ and we have

$$\alpha D(\alpha) + \beta C(\alpha) = C(\alpha)(\alpha + \beta) \approx 173(1/3000 + 2^{-15}) \approx 1/15.9.$$

In other words, the time-area product of the malicious prover is reduced by an approximate factor of 15.9.

To compute $D(\alpha)$ when using parallelism, observe that the circuit depth of the attack is increased by an expected factor of at most $(t+1)/2 = 10.5$ compared to the honest prover's algorithm.[14] This is because the expected circuit depth of computing an arbitrary block $X[i_j]$ in the chain is $(t-1)/2$ and computing the corresponding Y_j requires an additional hash function invocation (chains computed with different nonce values do not increase the circuit depth). We have

$$\alpha D(\alpha) + \beta C(\alpha) \approx 1/3000 \cdot 10.5 + 2^{-15} \cdot 173 < 1/113.$$

Consequently, the time-area product of the malicious prover is reduced by an approximate factor of 113.

6.2 Comparison with the Analysis of [7]

Plugging the parameter values of our attack $\alpha \approx 1/3000$ and $\epsilon = 1/20$ into Equation 2 (using Table 1), we obtain that the computational penalty should increase according to the MTP designers by a factor of more than

$$C(\alpha+\epsilon) \cdot (1-\epsilon)^{-L} = C(1/3000 + 1/20) \cdot (19/20)^{-70} \approx \\ 2^{5.2} \cdot C(1/19) > 2^{5.2} \cdot C(1/7) = 2^{5.2+18} = 2^{23.2},$$

which is about 10 million. This is more than 55,000 times as much as the 173 computational penalty we obtain. Using the actual value of $C(1/19)$ (instead of $C(1/7)$) would give a larger improvement factor.[15]

In terms of time-area product, our improvement ratio of 113 (or even the ratio of 15.9, obtained without exploiting parallelism) contradicts the claims of the MTP designers that the time-area product can be reduced by a factor of 12

[14] If we consider the initial computation of Argon2d by the honest prover, then $D(\alpha)$ can be even smaller.

[15] We do not have a concrete value of $C(1/19)$ for Argon2d, but based on the very fast growth of this function, the actual value of $C(1/19)$ should be significantly larger than $C(1/7) = 2^{18}$. Hence we expect our actual improvement ratio to be much larger than 55,000.

at most. Moreover, plugging our parameters into Equation 4 (using Table 1), we obtain a value which is more than

$$\beta C(\alpha + \epsilon) \cdot (1 - \epsilon)^{-L} > 2^{-15} \cdot 2^{23.2} = 2^{8.2}.$$

In other words, for our set of parameters, the time-area product of the attacker should increase by a factor of more than 294 according to [7]. As we actually reduce the time-area product by 113, we improve this analysis by a multiplicative factor which is more than 33,000. Once again, using the actual value of $C(1/19)$ (instead of $C(1/7)$) would give a larger improvement factor.

7 Extensions of the Attack

There are several possible extensions of the attack. In this section, we briefly mention two of them.

First, a natural question is how the attack behaves when changing the preprocessing complexity hard limit. Interestingly, the time-area product ratio (between the honest and malicious provers) does not drastically change when the preprocessing complexity varies in the region between 2^{48} and 2^{80}. For example, if we set the preprocessing complexity to a trivial value of 2^{48}, the time-area product ratio is still more than 60 in favor of the attacker.

Next, note that in order to detect inconsistency in the malicious proof, a verifier should (in addition to running the standard verification algorithm) compute Argon2d until the first block of the proof which in inconsistent with the Argon2d compression function. Detecting this inconsistency already involves non-trivial computation and memory. Furthermore, it is possible to further increase the amount of resources required for detection at the cost of increasing the amount of resources (namely, memory, time, or both) required for the attack.

More specifically, recall from Section 5.2 that we set a parameter δ which controls the number of prefix blocks that are computed honestly. So far, we assumed that $\delta \ll 1/t$, and thus this prefix can be neglected. However, we can set δ to a non-negligible value in order to force the verifier to compute more blocks to detect inconsistency. Obviously, this increases the memory complexity of the attack, but we can trade some of the memory for computation by storing only an α fraction of the prefix. As as result, we pay a computational penalty based on α according to Table 1 for these blocks. The total expected additional penalty for computing a block is multiplied by δ (since it only involves the prefix blocks).

8 Countermeasures

In this section, we discuss possible countermeasures for the attack.

We first consider simple ad-hoc countermeasures which try to detect some "non-random" properties of the proof. For example, one may observe that many blocks in the proof depend on control blocks which have a small value and occur at indexes which have the same offset modulo some number $t > 1$. However, such countermeasures are very easy to defeat. For example, we can start the exhaustive search for each control block from a pseudo-random value which depends on its index (and is not stored in memory). To eliminate the property that all control blocks have the same value modulo some number $t > 1$, we can use variable alignment of the blocks (e.g., by introducing a slightly shorter interval every 100 intervals). This may change the cost of the attack, as we need to keep track of the control block indexes, but all the additional work can be made negligible with appropriate choice of parameters.

Next, we consider other countermeasure options which adjust the parameters of MTP. For example, we can increase the value of L. This will definitely increase the complexity of the attack, but it will also increase the complexity of the verifier's (and honest prover's) algorithm. Overall, such countermeasures do not seem reasonable, as they do not eliminate the main properties of MTP which make the attack possible and they introduce additional overhead for the honest players.

Finally, the most reasonable countermeasure is to instantiate MTP with a memory-hard function that uses data-independent indexing (such as Argon2i). This may enable computation-memory tradeoff algorithms which are somewhat more efficient (for details regarding Argon2i, refer to [1, 2, 6]), but it completely resists the basic form of our attack. In the following, we call such an instantiation *data-independent MTP* in short.

It is natural to ask whether the analysis given by Equation 4 holds (even heuristically) for data-independent MTP. The answer to this question is negative and we demonstrate this by the following example: assume that two blocks X_1, Y_1 are computed as $X_1 = F(X_2, X_3)$, $Y_1 = F(Y_2, Y_3)$ and moreover, $Y_3 = X_3$ (they point to the same array index). Then, the attacker can set $Y_2 = X_2$, resulting in $Y_1 = X_1$. Altogether, the attacker introduced a single inconsistent block Y_2, but the result is that the two blocks Y_1, Y_2 do not have to be stored. This is inconsistent with Equation 4, since α is not directly reduced (the attacker can still compute all blocks with the honest prover's complexity), but the memory complexity is reduced by a factor of 2ϵ for a fraction of ϵ inconsistent blocks.

Overall, the above property does not immediately lead to a very efficient attack, but it seems undesirable in general. To avoid this self-similarity property, it is possible to use a different compression function F_i for each index i (essentially, setting i to be another input to F), as in the HAIFA mode-of-iteration construction for hash functions [5]. However, this tweak may still not be sufficient when using a compression function which is not collision resistent. In particular, if $X_1 = F_1(X_2, X_3)$, $Y_1 = F_2(Y_2, Y_3)$, the attacker may try to find a value for Y_2 such that $X_1 = Y_1$ (even if $Y_3 \neq X_3$), resulting in a similar property as above. When basing the functions F_i on Argon2's F, finding such collisions may be very simple, as it is trivial to find collisions in its compression function (which was not designed to resist to such attacks).

It may also be possible to exploit preprocessing in computing malicious proofs for data-independent MTP. Although it is unlikely that we can split the block array into completely independent parts (as in the case of Argon2d), we can make the first part (which depends on the online challenge) sufficiently small such that only a small fraction of blocks in the second part depend on it directly. This allows to compute most of the block array during preprocessing in a way that is highly consistent with the Argon2 compression function.

From the discussion above, we conclude that the analysis of data-independent MTP is non-trivial and we leave it to future work.

9 Conclusion

In this paper, we described a new cryptanalytic computation-memory tradeoff for MTP when instantiated with a memory-hard function that uses data-dependent indexing. When applied to the instance proposed by Biryukov and Khovratovich, our attack reduces the cost of a malicious prover's algorithm by a factor of 113 compared to the honest prover's algorithm. Finally, while data-independent MTP avoids the basic form of our attack, it may still be susceptible to its extensions and we leave its concrete instantiation and analysis to future work.

References

1. J. Alwen and J. Blocki. Efficiently Computing Data-Independent Memory-Hard Functions. In M. Robshaw and J. Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 241–271. Springer, 2016.
2. J. Alwen and J. Blocki. Towards Practical Attacks on Argon2i and Balloon Hashing. *IACR Cryptology ePrint Archive*, 2016:759, 2016. Presented at the IEEE European Symposium on Security and Privacy, 2017.
3. G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi. Proofs of Space: When Space Is of the Essence. In M. Abdalla and R. D. Prisco, editors, *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, volume 8642 of *Lecture Notes in Computer Science*, pages 538–557. Springer, 2014.
4. J. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. BLAKE2: Simpler, Smaller, Fast as MD5. In M. J. J. Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.
5. E. Biham and O. Dunkelman. A Framework for Iterative Hash Functions - HAIFA. *IACR Cryptology ePrint Archive*, 2007, 2007. http://eprint.iacr.org/2007/278.
6. A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 292–302. IEEE, 2016.
7. A. Biryukov and D. Khovratovich. Egalitarian Computing. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 315–326. USENIX Association, 2016.

8. A. N. J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction.* Princeton University Press, Princeton, NJ, USA, 2016.
9. C. Dwork and M. Naor. Pricing via Processing or Combatting Junk Mail. In E. F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.
10. S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. Proofs of Space. In R. Gennaro and M. Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 585–605. Springer, 2015.
11. C. Lee. Litecoin. https://litecoin.org/, 2011.
12. M. Mahmoody, T. Moran, and S. P. Vadhan. Publicly verifiable proofs of sequential work. In R. D. Kleinberg, editor, *Innovations in Theoretical Computer Science, ITCS '13, Berkeley, CA, USA, January 9-12, 2013*, pages 373–388. ACM, 2013.
13. R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.
14. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2009.
15. Password Hashing Competition. https://password-hashing.net/, 2015.
16. C. Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. http://www.tarsnap.com/scrypt/scrypt.pdf, 2009.

A Merkle Hash Trees

Merkle hash tree is a data structure proposed by Ralph Merkle in order to create digital signatures using symmetric cryptography primitives [13]. However, it has found many additional applications since its introduction.

The value of every non-leaf node in a Merkle hash tree is computed by hashing the concatenated values of its children nodes. An opening of a data block X_i is a proof that $H(X_i)$ is indeed a leaf in the tree contained in the tree at index i. The opening includes X_i itself and the values of the siblings of $H(X_i)$'s path to the root, which are sufficient to compute the full path. For example, the opening of X_3 in the tree of Figure 4 contains (besides the value of X_3) the values of $H(X_4)$ and $H(H(X_1), H(X_2))$.

Assuming that the hash function H is collision resistant, the Merkle hash tree construction guarantees that it is computationally hard to open a block in more than one way.

B The Indexing Function of Argon2d

Recall from Section 3.2 that given indexes i, j of a block $[i][j]$, we first compute J_1, J_2 and determine the lane of the referenced block, denoted by l. Next, we

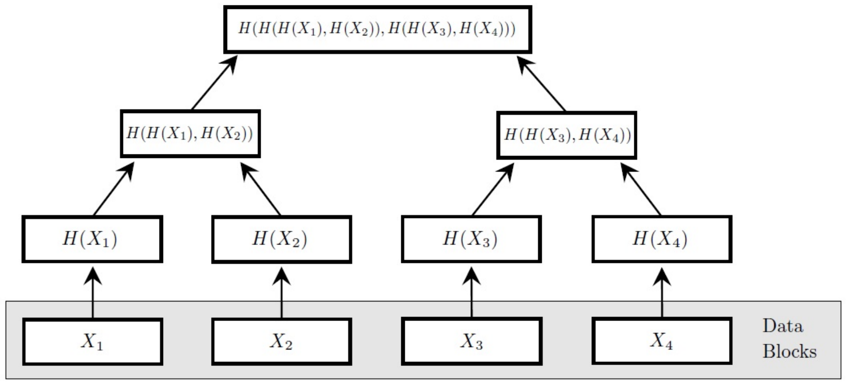

Fig. 4. Merkle Hash Tree

show how to compute the block index z within the lane. First, we determine the set of indexes that can be referenced given $[i][j]$ (denoted by $\mathcal{R}$) according to the following rules:

1. If l is the current lane, then $\mathcal{R}$ includes all previous blocks computed in this lane, excluding $[i][j-1]$.
2. If l is not the current lane, then $\mathcal{R}$ includes all blocks in the segments whose computation is finished in lane l. If $[i][j]$ is the first block of a segment, then the last block from $\mathcal{R}$ is excluded.

The size of $\mathcal{R}$ is denoted by $|\mathcal{R}|$. The value of J_1 determines the block index within this lane by computing (over the integers)

$$\begin{aligned} x &= (J_1)^2/2^{32}; \\ y &= (|\mathcal{R}| \cdot x)/2^{32}; \\ z &= |\mathcal{R}| - 1 - y. \end{aligned} \tag{10}$$

The value of ϕ is $[l][z]$. Note that ϕ for $[i][j]$ defines a non-uniform distribution over the indexes of $\mathcal{R}$, where more weight is placed on the larger indexes closer to j.

C Optimizing Phase 1 for Argon2d

In this appendix, we exploit the Argon2d compression function to optimize phase 1 of the attack (described in Section 5) by a factor of about $8t/(t+8)$. We start with a high-level description of the Argon2d compression function, which is sufficient to understand the optimization (more details can be found in [6]).

The compression function F takes as input 2 blocks of 1024 bytes, denoted by X and Y, and outputs the single block $G(X \oplus Y) \oplus X \oplus Y$, where G is a permutation on 1024 bytes. The permutation G views the 1024 input bytes as an 8×8 matrix A, where each entry is of size 16 bytes. It applies a 128-byte permutation P once to all 8 rows of the matrix and once to all 8 columns (altogether P is applied 16 times in G). The permutation P is the round-function of Blake2 [4], but we do not exploit its specification in the attack. In total, the main effort in computing F is in the 16 applications of the permutation P.

We now show how to reduce the expected number of invocations of P in phase 1 of the attack. Recall that in phase 1 we set the 64 LSBs of the control block to some fixed value (ensuring that $\phi(s \cdot t + 1) = (s-1) \cdot t$). We then do an exhaustive search on the remaining bits to find a value such that all the following blocks in the current interval (with indexes $2, 3, \ldots, t-1$ inside the interval) are also computed using previous control blocks.

The expected complexity of verifying that a control block value satisfies the constraints is about $1 + 1/t + (1/t)^2 + \ldots + (1/t)^{t-2}$, where the first term (1) in the sum stands for the first F computation (that takes as input the current control block $X[s \cdot t]$ and the previous one $X[(s-1) \cdot t]$). Note that $X[(s-1) \cdot t]$ never changes during the exhaustive search, while $X[s \cdot t]$ changes, but this change is limited to a single 16-byte entry[16] in the 8×8 matrix A (whose value is $X[s \cdot t] \oplus X[(s-1) \cdot t]$) at the input of G. Therefore, 7 out of 8 rows of A never change during this computation and their value after the application of P can be computed once, stored in memory, and reused. Moreover, in order to verify the next condition (namely, compute $\phi(s \cdot t + 2)$ of the next block), we only need to compute the 64 LSBs of the output of F. Hence, after computing P on the rows, we compute P on the first column, verify the condition, and only if it is satisfied we continue (otherwise, we change $X[s \cdot t]$ and recompute $F(X[s \cdot t] \oplus X[(s-1) \cdot t])$). The probability that the condition will not be satisfied is $(t-1)/t$, implying that we need to apply P only twice, equivalent to $2/16 = 1/8$ full F invocations. With probability $1/t$ we continue by first finishing the computation of $F(X[s \cdot t] \oplus X[(s-1) \cdot t])$ on the remaining 7 columns. We start computing the next invocation of $F(X[s \cdot t + 1], X[\phi(s \cdot t + 1)])$ by applying P to the 8 rows of A, and once more to the first column, giving the 64 output LSBs that allow to compute $\phi(s \cdot t + 2)$. Once again, we continue only if $\phi(s \cdot t + 2)$ satisfies the condition of Equation 5. Thus, with probability $1/t$, we compute $7 + 8 + 1 = 16$ additional P invocations, equivalent to a single F invocation.

In total, the expected complexity of verifying a single value of the control block in the exhaustive search is optimized to $1/8 \cdot (t-1)/t + 1/t + 1/t^2 + \ldots < 1/8 + 1/t$ invocations of F. Therefore, the total complexity of phase 1 is optimized by a factor of about $8t/(t+8)$.

We note that additional optimizations are possible by exploiting differential properties of P. However, we do not mention such optimizations here as they are more complex and do not seem to lead to a significant advantage.

[16] We iterate through the 64 MSBs of the first matrix entry (the 64 LSBs are fixed) and the total computation per control block is significantly lower than 2^{64}. Hence, we do not overflow the first matrix entry.