# Efficient generation and dispersion of entropy during data preparation for Arweave.

Digital History Association (DHA) team

Weavemail: vLRHFqCw1uHu75xqB4fCDW-QxpkpJxBtFD9g4QYUbfw

Draft-5
December 2024

## Abstract

This paper introduces a major improvement to Arweave's data preparation algorithm, reducing upfront mining computation costs by significant margins for honest participants while preserving the network's existing security guarantees. We analyze the security framework of Arweave's existing data preparation scheme, where the opportunity for optimization arises, and outline the design of this new approach. By optimizing entropy generation and dispersion in the data preparation process, the proposed algorithm incentivizes wider-participation and the development of a more efficient storage market in the network.

## 1 Background

Arweave is a decentralized and permissionless permanent data storage network, backed by an endowment. Additional context on Arweave, its purpose, and the architecture of its core protocol can be found in its paper [18].

Arweave 2.8.x and prior versions of the network utilize a data 'packing' scheme that requires miners to perform work in order to incentivize them to make and store many replicas of the dataset, rather than one 'unpacked' copy mined across many identities. In order for this system to function correctly, the following invariant must hold:

$$m_s(d_{sz}, t) < m_p(d_{sz})$$

where,

$m_s(n, t) =$ Cost to a miner $m$ to store $n$ bytes for $t$.
$m_p(n) =$ Cost to a miner $m$ to pack $n$ bytes.
$t =$ Mean time between access of given bytes.
$d_{sz} =$ The number of bytes in a unit of data.

If this invariant holds, it is always preferable for miners to store the data in many packed copies, rather than to mine while packing 'on-demand' ($o$). However, satisfying this invariant places a compute burden on miners in the network that are executing the 'honest' strategy ($h$). Where possible, the design goal of Arweave's proof system is to minimize the requirements (such as these) on miners beyond the simple provisioning of storage capacity, as codified in the immutable principles of the network [12].

While preparing their data $d$ for storage, $h$ must perform $m_p$ work upon each individual piece of data that they would like to store using the RandomX CPU-bound work algorithm [14] developed by the Monero community. This unit of work produces an equivalently sized series of bytes $d_e$ that are sufficiently entropic [13, 15] that they cannot be effectively compressed. Assuming the on-demand packing invariant, after production of $d_e$ the rational strategy for a miner is to mask it against $d$ to produce the necessary $d_p$ that is utilized in proof generation, as

$$m_x m_s(d_p) < (m_s(d_u) + m_x m_s(d_e))$$

where,

$m_x =$ Number of replications of the data to mine.
$d_u =$ Unpacked form of $d$.
$d_p = d_u$ packed; masked against $d_u$ with $e(d_e, d_u)$.

At any point in time during mining, each packed 'partition' (a 3.6 TB section of the network's data) has a 'read-head'. The position of this read-head is pseudo-randomly determined by the VDF [6, 3] of the network, mixed with entropy from the miner's packing address and the partition number. Once the read-head (and subsequently, the miner's underlying storage medium) has 'seeked' to the new position in a partition, a number ($r$) of $d_p$ are read from the dataset. Because the position inside a partition is chosen in a way that miners cannot predict, $o$ would either have to recalculate $rd_p$ in real-time as their data is sampled, or store a cache of their packed data chunks. The caching strategy is only effective proportionate to the quantity of the stored chunks, resolving to simply storing $d_p$ as $h$ does in extremis.

One seek of the read-head for a miner results in the release of $r$ 'challenges' that may satisfy the network's global difficulty $n_d$ (granting the right to generate a block, as per Bitcoin [8] and similar Proof-of-Work networks). Each $r$ requires the reading of (and consequently, access to) one unit of data. This imparts a consistent work requirement of $rm_p$ upon $o$ in order to execute their strategy. Arweave 2.8 introduced the ability for a miner to optionally perform additional work during packing in order to lower their data read-speed requirement ($r$). This mechanism enforces a custom difficulty ($m_d$) associated with their packed data, granting them a modified $n_d$ that can be modeled in the formula. For the sake of comprehensibility, this equation omits modifiers to $n_d$ that relate to mechanisms outside the scope of this paper.

$$\mathbb{P}(Valid(r, m_d)) = \frac{n_d}{m_d}$$

where,

$Valid(r, m_d)$ : One of $m$'s proofs from $r$ satisfies $n_d$.

The network's global challenge rate $n_r$ is reduced for $m$ in accordance with $m_d$:

$$m_r = \frac{n_r}{m_d}$$

Subsequently, $m$ is able to read from their disk at rate $m_r$, while their likelihood of finding a valid work proof remains unchanged:

$$\mathbb{P}(Valid(r, m_{d=1})) = \mathbb{P}(Valid(r, m_{d=x})) \; \forall x \in [2..n_{dm}]$$

where $n_{dm}$ is the maximum admissible packing difficulty in the network (32, as of Arweave 2.8). This limit is necessary as $m_{d=n}$ mining a block requires all network verifiers to perform $p_{d=n}$ at $nm_p$ cost, bared by the validator not block producer.

## 2 Potential efficiencies

In the ideal case, the Arweave data preparation scheme would minimize honest miner compute costs ($h_p$), while maximizing the costs for a miner executing the on-demand strategy $o_p$. While the on-demand mining invariant referenced above and in the Arweave paper remains a valid constraint, this specification describes an algorithm that increases $o_p$ while maintaining existing $h_p$ levels. The effect of this algorithm is a significant reduction in the overhead (non-$m_s$) costs for honest miners, while also allowing the network to purchase storage from $m_h$'s at a more efficient price.

In Arweave 2.8, $p_{d=n}$ is performed by running RandomX using the standard 2 MiB scratchpad $n$ times sequentially, mixing the scratchpad entropy with an AES-256 Feistel cipher [7] $n$ times during execution. The first 8 KiB of the resulting scratchpad is then taken and used as $d_e$ in $e(d_e, d_u)$, yielding $d_p$. Notably, this mechanism discards almost all of the sufficiently entropic data that it produces, approximately $\approx 99.6\%$). This observation gives rise to the opportunity for generating significant efficiencies.

A simple but flawed solution to the inefficient use of this entropy would be to allow miners to use the full scratchpad output as $d_e$ to mask against their unpacked data in $e$ consecutively. However, this mechanism would mean that $o$ would be required to compute only $\frac{1}{256}rm_p$, as the $d_e$ for $r_1$ would be reusable for all $r_{2..256}$ of the subsequent challenges. As a consequence, the on-demand packing invariant would break and $o$ would become the dominant strategy for mining, rather than $h$.

Additionally, we note that in Arweave 2.8 $h$ must perform $m_x m_d m_p(d_{sz})$ work in order to pack their replicas, while $o$ performs only $trm_x m_p$ calculations (where $t$ is the duration of their mining period). Over time, the total units of $m_p$ for $o$ is likely to exceed that of $h$, however $o$'s requirement to perform $m_p$ in real-time gives rise to the potential to differentiate and discourage the strategy in ways that do not affect $h$.

## 3 The data activation algorithm

Building on the insights above, we propose a data preparation scheme that captures the described efficiencies, for deployment in the next proposed version of the protocol (Arweave 2.9). We call this mechanism 'activation' rather than packing, as while it serves the same security purposes for the network as traditional packing, it removes virtually all noticeable computation requirements for honest miners (approximately 99.3% less than Arweave 2.8's $p_{d=1}$ and 99.95% less than $p_{d=16}$, depending upon parameters). The algorithm introduces the following parameters:

$a_l$ = Parallel 'lanes' of computation.
$a_m$ = Number of entropy 'mixes' between lanes.
$a_z$ = Number of zones in a single partition.
$a_d$ = Effective difficulty, implying $m_r$.
$a_{gm}(x)$ = Generator function for mixing data $x$.

Proposals for these parameters, as well as constants utilized by RandomX itself, are described in section 4.

At a high level, the data preparation algorithm executes as follows:

1. The network's data partitions are sliced into $a_z$ approximately equally sized zones.
2. For $z = 1..a_z$:
   (a) Miners perform $a_l$ operations of $p_{d=1}$, using the offset of the data they are attempting to prepare (as well as their packing key, etc) as

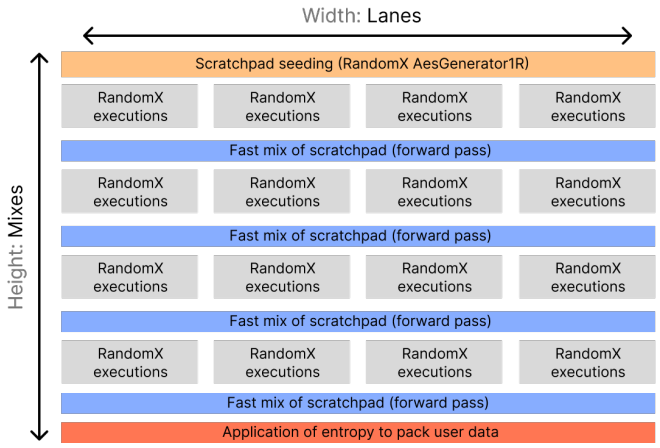


Figure 1: RandomXSquared uses a 'square' of RandomX invocations in order to generate and mix entropy across a scratchpad of increased size.

their seed entropy. Each operation ('lane') uses a unique nonce, generating a unique 2 MiB scratchpad with the same non-compressibility properties of Arweave 2.8.

(b) $a_m$ times during the execution of RandomX upon each scratchpad computation is paused and entropy is mixed across all $a_l$ using $a_{gm}$ across a concatenation of the scratchpad states $a_{s[1..a_l]}$ as input.

(c) After the final RandomX executions have finished, the mix generator $a_{gm}$ is executed upon the concatenated scratchpad an additional time.

(d) The resulting combined result of all lanes is treated as a single, large $d_e$ of size $a_l * 2\text{MiB}$.

(e) Rather than using only the first 8 KiB (as in Arweave 2.8), $d_e$ is split into $\frac{8KiB}{a_l * 2\text{MiB}}$ chunks of entropy $d_{e[1..a_c]}$.

(f) $d_p$ is calculated for the appropriate subchunk in each zone as $e(d_{e[n]}, d_u)$.

This algorithm is composed of two core elements: the data packing loop which merges network data and generated entropy, as well as the algorithm for generating the entropy itself. We discuss each in turn in the following sections.

### 3.1 Efficient distribution of non-compressible entropy across the dataset

As shown in listing 1, the Arweave 2.9 data preparation scheme uses the full entropy generated by the RandomXSquared algorithm, rather than just the first 8 KiB of the final scratchpad in the execution chain. By distributing the created entropy across an entire partition, rather than sequentially, we additionally avoid the problem outlined by the naïve algorithm described in section 2: In this scheme, $o$ would be forced to perform the work required to generate the entropy used by $h$ in many places, only to discard it (wasting the associated $m_p$ work), or store it – just like $h$. This improvement alone yields a $p_d \frac{d_{sz}}{8KiB} \frac{|R_{pad}|}{8KiB}$ decrease in the total number of individual RandomX instructions that $h$ must execute in order to pack a replica of the network's dataset. In concrete terms, using equivalent parameters from Arweave 2.8 for a $m_{d=1}$ miner this mechanism makes the process of packing 256 times more efficient for $h$, without affecting the $h_p : o_p$ ratio. For a $p_{d=32}$, this algorithm (with the parameters proposed in section 4) 8,192 times more efficient.

Depending upon parameters, this algorithm also grants significant benefits to honest miners in the form of highly reduced read speed requirements. In Arweave 2.7.x, miners needed to read from their drives at $r$ of 200 MiB/s (approximately 800 chunks per second). This was due to the need to for potential $o$'s to be forced to do enough work such that the $h_p : o_p$ ratio stayed within safe margins. In practice, this placed a very significant strain on miners with many replicas of the network's dataset due to the volume of data that needed to be transferred between disks and main memory in order for proofs to be computed. Arweave 2.8 improved upon this by allowing miners to trade-off additional upfront packing work in exchange for reduced $r$ (and effective challenge difficulty) during mining.

The proposed algorithm in this paper takes the improvements from Arweave 2.8 a significant step further: By forcing miners attempting the $o$ strategy to perform the work required for $h$ to pack an extremely large number of chunks, $r$ can safely be reduced globally. For example, using parameters $a_l = 4$ and $a_d = 3$ would safely reduce $r$ globally to approximately 5 MiB/s – 20 chunks of data, while simultaneously keeping the $h_p$ costs for $o$ the same. These parameters would grant $h$ an extremely significant reduction in practical mining costs without the overhead of packing their data to higher difficulties, as Arweave 2.8 made possible.

### 3.2 RandomXSquared

Arweave 2.9 utilizes a new algorithm (RandomXSquared, shown in 1) for its entropy generation scheme, constructed on-top of RandomX.

RandomXSquared is designed to meet the following objectives:

1. **Efficient generation of entropy:** The algorithm must maximize the amount of *sufficiently random* data that it produces for a given quantity of work. Notably, we do not see *cryptographically secure* entropy generation as a necessary objective. Data resulting from the algorithm must be random enough

**Algorithm 1** Data enciphering and entropy distribution across partitions, using RandomXSquared.

$\mathrm{RX2_{pad}} \leftarrow 2$ MiB
$\mathrm{RX2_{progs}} \leftarrow \frac{2048}{\mathrm{a_m}}$
$\mathrm{z_{sz}} \leftarrow \frac{|\mathrm{RX2_{pad}}| \times \mathrm{a_l}}{c_{sz}}$
$\mathrm{a_z} \leftarrow \mathrm{ceil}(\frac{\mathrm{P_{sz}}}{\mathrm{z_{sz}}})$

**for** $z \leftarrow 0$ **to** $a_z - 1$ **do**
    $\mathrm{s} \leftarrow \mathrm{SHA256}(\mathrm{m_{addr}} \parallel \mathrm{P_n} \parallel z)$
    $\mathrm{z_e} \leftarrow \mathrm{RandomXSquared(s)}$
    **for** $c \leftarrow 0$ **to** $\mathrm{z_{sz}} - 1$ **do**
        $c_i \leftarrow p_n p_{sz} + c z_{sz} + z$
        **if** $\mathrm{d_u} \leftarrow \mathrm{Read}(c_i)$ **is not null then**
            $\mathrm{d_p} \leftarrow \mathrm{e}(\mathrm{z_{e[c]}}, \mathrm{d_u})$
            $\mathrm{Write}(c_i, d_p)$
        **end if**
    **end for**
**end for**

the a would be $o$ cannot predict any significant strings of the algorithms outputs without first performing the intended work.

2. **Random-access resistance:** The resulting entropy produced by the scheme must be created in a single, non-divisible unit of work, such that a would-be $o$ miner cannot truncate any significant part of the work in order to access only a subset of the resulting bytes.

3. **Parameterizable CPU-bound workload:** In order to make the Arweave network's on-demand safety margin (the ratio of $h_s : o_p$) secure, the algorithm must enforce a workload with *predictable* execution speeds with low volatility over time. A workload tailored to execution upon general-purpose CPUs appears to be the best route to this object, as general-purpose computation is the form that has already experienced the highest incentive to maximize efficiency. Subsequently, work of this kind is less likely to experience extreme declines in its execution time than less well-optimized forms workloads.

Before describing RandomXSquared in-depth, we will first explore the design and properties of RandomX itself, as well as its suitability as a primitive for CPU-bound entropy generation.

#### 3.2.1 RandomX

RandomX is a hashing algorithm designed for use in proof-of-work contexts, rather than in the generation of cryptographically secure commitments [14]. Its core design goal is the construction of a work mechanism that could not be feasibly improved with the use of custom-built ASICs, such that the Monero network's miners could operate competitively using only commodity, non-specialized hardware. A RandomX hash is generated by first creating a 'scratchpad' of approximately L3 CPU-cache size and initial register states, seeded with entropy created by a BLAKE-2b [2] hash of its inputs. This BLAKE-2b hash is extended using an $AES1R$-based generator [1]. Additionally, a large dataset that serves as a 'ROM', optimized for random access, is created from an intermediate Argon2 [4] derived cache. While Argon2 itself does meet some of our design requirements, as noted by Boneh et al in [5], it does not enforce sufficiently strong guarantees against work-truncation attacks to make it random-access resistant. Additionally, it is optimized for memory-bound work in a similar fashion to Cuckoo Cycle [17], which experienced significant execution speed volatility when GPU-based implementations were created.

After initialization, RandomX runs pseudo-randomly generated programs based on its input entropy. While the output of a RandomX execution is always deterministic, the algorithm is designed such that it is not to be predictable without expending a specific (tunable) amount of CPU work. In order to achieve this, each RandomX program execution involves parts of its VM's register states being periodically written into the scratchpad in a manner that cannot can only be calculated via execution of the programs themselves. Subsequently, over many iterations a RandomX scratchpad becomes increasingly mixed and cannot be reproduced without re-execution of the CPU instructions, or caching of memory writes (ISTORE instructions). Finally, after many iterations (2048 by default) of a number of unique RandomX programs, a fingerprint of the scratchpad is generated and combined in another BLAKE-2b hash with the register states. This hash represents the result that is given to the caller.

#### 3.2.2 ISTORE caching attacks

While ISTORE caching attack vectors are not a relevant concern for RandomX's original use case (Monero proof-of-work mining), without additional measures they could pose a significant risk to alternative uses of RandomX's work-bound entropy generation. An example attack flow is as follows:

1. Eva initializes a bitmap of size $\frac{|RX_{pad}|}{RX_{word}}$ bytes. Using the standard RandomX parameters, the resulting bitmap would be 32.7kb for a 2 MiB scratchpad.

2. Eva augments the RandomX virtual machine's implementation of the ISTORE instruction to additionally set the corresponding bit for the memory cell in the bitmap to '1'.

3. The full RandomX hash is executed.

4. A memorized representation of the scratchpad entropy is constructed by storing the bitmap and a

concatenation of each of the final values for the data at the associated memory locations.

5. When Eva needs to access the entropy again, she simply executes the fast BLAKE2b and $AES1R$ generator calls as normal to initialize the scratchpad, but instead of executing the programs she copies and replaces the data at the appropriate slots using her bitmap and cache.

The RandomX parameter tuning documentation [16] encourages users not to choose parameters that break the following invariant:

$$(128 + |RX_{prog}|RX_{istores})(RX_{progs}RX_{its}) \geq |RX_{pad}|$$

With the default $RX_{istores}$ and $RX_{prog\_sz}$ parameters, this equation simplifies to $RX_{total\_istores} \geq |RX_{pad}|$. That is; there should be at least as many ISTORE operations across the totality of instructions executed in the generation of a hash as there are cells in the scratchpad. The design documentation [15] also notes that this leads to 66% of the scratchpad being overwritten. In the context of our hypothetical attack, Eva would be able to compress an average scratchpad by 32.5% as her bitmap is only 1.5% of the size of the entropy.

Due to the birthday paradox and associated results [11], attempting to rectify this issue by simply increasing the total number of RandomX instructions that are executed (or, as the algorithm allows for, increasing the frequency of ISTORE operations relative to others in the instruction set) has only limited impact. Multiplying the total amount of work performed in the entropy generation process would still leave Eva with the ability to discard 12.5% of the entropy.

As noted, this attack vector is not of significant consequence to Monero, or to Arweave 2.8. Arweave 2.8 utilizes only the first 8 KiB of the scratchpad which receives significantly higher numbers of writes due to RandomX's L1-L3 cache utilization parameters. This attack would, however, make the proposed Arweave 2.9 data preparation scheme vulnerable to 'packing compression' (a mixture of on-demand mining behavior and RandomX ISTORE cachine behaviors) if left unaddressed.

## 3.3 ISTORE caching resilience with RandomXSquared

In order to alleviate the issues with using only a 'stock' deployment of RandomX as an entropy generation source for Arweave data preparation, we propose RandomXSquared: A simple construction on top of RandomX that intends to maximize the amount of sufficiently random entropy that can be generated in an atomic unit of CPU-bound work. Additionally, RandomXSquared is structured such that only minimal quantities of work can be truncated by an attacker that wishes to access only a small subset of the generated entropy.

**Algorithm 2** RandomXSquared: A grid of parallel RandomX channels, yielding a variable quantity of data that cannot be randomly accessed without full computation. This listing describes a sequential implementation of the algorithm for brevity, although implementations parallelized by lanes is also possible.

```
1: function RandomXSquared(s)
2:     b ← Array[a_l × |RX_pad|]
3:     for l ← 0 to a_l − 1 do
4:         RX_init(b_l, s ‖ l)
5:     end for
6:     for m ← 0 to a_m − 1 do
7:         for l ← 0 to a_l − 1 do
8:             RX_exec(b_l, RX_progs)
9:         end for
10:        b ← a_gm(b)
11:    end for
12:    return b
13: end function
```

RandomXSquared operates by constructing a square of 'lanes' and 'rows' of individual RandomX contexts. Each lane in RandomXSquared has a RandomX scratchpad with a unique seed as its starting entropy. The initial scratchpads in each lane are created using RandomX's standard $AesGenerator1R$ generator. Each lane can be executed in parallel on separate CPU threads, with the sequential (non-parallelizable) mix steps momentarily merging execution and ensuring that entropy is shared across all lanes. The result is a non-compressible chunk of data with a width proportionate to $a_l|R_{pad}|$.

## 3.4 Work-Truncation attack resilience

By mixing entropy regularly across every byte of every scratchpad on multiple occasions during execution, ISTORE caching attacks of the previously described form become impossible. If executed exactly as described previously, Eva would end up storing the full scratchpad size, plus her additionally generated bitmap. A modification of the ISTORE caching approach in the RandomXSquared context would be to generate caches at each of the mix phases of the execution. This approach, too, would lead to the attacker storing more than $|RX_{pad}|$ bytes, as every additional mix phase (assuming the same amount of work at each layer) would generate a cache of equivalent size to a base RandomX hash.

When utilizing RandomXSquared, random access to any individual subchunk of data should not be possible without having first computed all of the associated lanes of RandomX execution, as the entropy of one is mixed with the entropy of all others at multiple stages during execution. An exception is in the final scratchpad entropy mix, in which hypothetically $o$ could ignore com-

| Work Mult. | RandomX | RandomXSquared |
|---|---|---|
| 1x | 63.3% | 63.3% |
| 2x | 86.6% | **126.6%** |
| 3x | 95.3% | **189.9%** |

Table 1: ISTORE cache attack storage requirements for RandomX alone and RandomXSquared. While RandomX tends towards all memory cells being written to during execution (yielding an ISTORE cache of $|RX_{pad}|$ for the attacker, the cost to attack RandomXSquared scales linearly with work, *exceeding* $|RX_{pad}|$ rapidly.

pute after the final write to a subchunk that they want to access. This issue is alleviated by ensuring that the choice of $a_{gs}$ has a very low proportionate execution cost relative to the totality of work in the RandomXSquared run. Specifically, the work truncation possible on an average unit of compute can be calculated as $\frac{1}{2}\frac{c(a_{gs})}{c(RX2)}$.

# 4 Parameters

As described in section 3, utilizing RandomXSquared introduces a number of new parameters to the network. Given that these parameters are global (affecting all miners – and consequently, users – of the network), they must be chosen with care. In this section we will outline a set of proposed parameters, their implications and potential trade-offs.

The parameters that we propose for adoption in Arweave 2.9 are as follows:

$$a_l = 4$$
$$a_m = 4$$
$$a_d = 3$$
$$a_{gm}(x) = \text{CRC32} \rightarrow \text{Shuffle [block=6 bytes]}$$

For $a_{gm}$, we propose a combination of a local (applied to each individual RandomX scratchpad) CRC32, followed by a global application of a deterministic shuffling algorithm across the full concatenation of scratchpads. The CRC32 algorithm was original described in 1961 by Peterson and Brown et al [9]. Since that time it has been widely deployed and is implemented in hardware accelerated form in commodity Intel, AMD, ARM, and RISC-V processors. This proposal employs the CRC-32C Castagnoli variant. Additionally, the proposed shuffling algorithmic is static and ensures next lane will use some portion of data from all of other lanes. Utilizing a 6 byte block size ensures that each RandomX scratchpad word (8 bytes) is shifted and mixed. This mechanism guarantees entanglement of each write in one lane will affect every execution of RandomX in other lanes during later mix steps. As noted in section 3.2, RandomXSquared does not require that $a_{gm}$ creates outputs that satisfy the typical requirements of cryptographic security, but instead that they create a *sufficiently random* stream of bytes that $o$ cannot reliably infer the result without executing the surrounding RandomX computations. Additionally, as noted in section 3.4 it is imperative that the execution cost of $a_{gm}$ remains low relative to an $RX2$ invocation, such that work-truncation attacks are not viable. For this reason, the CRC32 → shuffle construction is proposed, as opposed to computationally expensive alternatives (SHA-2, BLAKE-3, etc).

## 4.1 Parameter implications

The implications of the suggested parameters are as follows:

- **Reduced compute requirements for data preparation.** An $a_l$ value of 4 would yield a 99.6% reduction in the amount of compute required by honest miners, when compared against Arweave 2.7 and Arweave 2.8 with a packing difficulty of 1. Arweave 2.8 does not allow for packing greater than a difficulty of 32, rendering a real-world comparison with the proposed Arweave 2.9 global difficulty of $a_d = 3$. However, taking the highest available packing difficulty of $p_{d=32}$ renders an improvement of 99.987% (one $m_p$ in Arweave 2.9 for every 8,192 $m_p$ in Arweave 2.8 at $p_{d=32}$). In our benchmarks, excluding bookkeeping overhead this parameter should allow even extremely low-power, low-cost hardware (for example, an ARM Raspberry Pi 5) to prepare a replica of the network's data in an acceptable timeframe.

- **Reduced read-rate requirements during mining.** Utilizing a global effective difficulty ($e_d$) setting of 50 would yield a necessary read rate ($r$) for honest miners of 1 MB per second, rather than the standard rate of 50 MB/s for data packed with $p_{d=1}$ in Arweave 2.8, and 200 MB/s for all data in Arweave 2.7.x.

- **Increased on-demand mining safety margin.** An additional effect of the parameters $a_l = 4$ and $e_d = 50$ is that the on-demand mining safety margin of the network – the ratio $c(h_p) : c(o_p)$ of profitability for honest ($h$) miners against on-demand ($o$) miners – would increase approximately 52.87%.

- **Increased proof validation times.** One drawback of this approach is an increase in the proof validation time for nodes in the network, although still within acceptable ranges (≈200ms on our reference hardware). While this work does represent an increase from Arweave 2.8, the practical impact of this effect is mediated by the introduction of parallelizability of proof validation. With $a_l = 4$ and the proposed fast choice of $a_{gm}$, miners would be able to parallelize large parts of validation on up to 4 threads.

A full model of the of the effects of these proposed parameters, as well as a simulation of the effects on entropy quality of additional mix phases ($a_m$) has been published to accompany this paper [10].

# 5 Conclusion

In this paper, we have proposed a new data preparation mechanism for Arweave which significantly enhances the efficiency of entropy generation and dispersion during mining operations. The introduction of RandomXSquared as a core component achieves several key objectives: reducing computational overhead for honest miners by significant margins, lowering the necessary data access rate for miners by 90%, while maintaining the existing security properties of the network.

# References


[1] Ako Muhamad Abdullah et al. "Advanced encryption standard (AES) algorithm to encrypt and decrypt data". In: *Cryptography and Network Security* 16.1 (2017), p. 11.

[2] Jean-Philippe Aumasson et al. "BLAKE2: simpler, smaller, fast as MD5". In: *Applied Cryptography and Network Security: 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings 11*. Springer. 2013, pp. 119–135.

[3] Lev Berman and Sergii Glushkovskyi. URL: https://github.com/ArweaveTeam/arweave/blob/master/apps/arweave/src/ar_vdf.erl.

[4] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: new generation of memory-hard functions for password hashing and other applications". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 292–302.

[5] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. "Balloon hashing: A memory-hard function providing provable protection against sequential attacks". In: *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*. Springer. 2016, pp. 220–248.

[6] Dan Boneh et al. *Verifiable Delay Functions*. Cryptology ePrint Archive, Paper 2018/601. https://eprint.iacr.org/2018/601. 2018. URL: https://eprint.iacr.org/2018/601.

[7] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. 5th. CRC Press, 2001, p. 251. ISBN: 978-0849385230. URL: https://archive.org/details/handbookofapplie0000mene/page/250/mode/2up.

[8] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: *Cryptography Mailing list at https://metzdowd.com* (Mar. 2009).

[9] William Wesley Peterson and Daniel T Brown. "Cyclic codes for error detection". In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235.

[10] Lev Berman Sam Williams James Piechota and Sergii Glushkovskyi. *Arweave 2.9 Parameter Tuning Model*. URL: https://docs.google.com/spreadsheets/d/1C0t8Yuz1SSOm-CVTtyoqgieC5wTagWacf7pV8zRRbk8/edit?usp=sharing.

[11] Kazuhiro Suzuki et al. "Birthday paradox for multi-collisions". In: *Information Security and Cryptology–ICISC 2006: 9th International Conference, Busan, Korea, November 30-December 1, 2006. Proceedings 9*. Springer. 2006, pp. 29–40.

[12] Arweave Core Team. *Principles of the Arweave network*. URL: https://arweave.net/1rL73ctmqTVv7qkqAsD4jIz5tU6WxgOAqABYuMHg5mQ.

[13] tevador. URL: https://github.com/tevador/RandomX/blob/master/src/tests/scratchpad-entropy.cpp.

[14] tevador. *RandomX*. https://github.com/tevador/RandomX. Accessed: December, 2024. Dec. 2022.

[15] tevador. *RandomX design*. https://github.com/tevador/RandomX/blob/master/doc/design.md. Accessed: December, 2024. Dec. 2022.

[16] tevador. *RandomX design*. https://github.com/tevador/RandomX/blob/master/doc/configuration.md. Accessed: December, 2024. Dec. 2022.

[17] John Tromp. "Cuckoo cycle: A memory bound graph-theoretic proof-of-work". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2015, pp. 49–62.

[18] Sam Williams et al. "Arweave: The Permanent Information Storage Protocol". In: (2023). URL: https://draft-17.arweave.net/.