Proofs of Useful Work from Arbitrary Matrix Multiplication

Ilan Komargodski
School of Computer Science and Engineering, Hebrew University of Jerusalem, Israel

Omri Weinstein
School of Computer Science and Engineering, Hebrew University of Jerusalem, Israel

Abstract

We revisit the longstanding open problem of implementing Nakamoto's proof-of-work (PoW) consensus based on a *real-world* computational task $T(x)$ (as opposed to artificial random hashing), in a truly permissionless setting where the *miner itself* chooses the input x. The challenge in designing such a Proof-of-Useful-Work (PoUW) protocol is to use the *native* computation of $T(x)$ to produce a PoW certificate with prescribed hardness and with negligible computational overhead over the worst-case complexity of $T(\cdot)$. This ensures malicious miners cannot "game the system" by fooling the verifier to accept with higher probability than honest miners while using similar resources. Indeed, obtaining a PoUW with $O(1)$-factor overhead is trivial for *any* task T, but also useless.

Our main result is a PoUW for the task of *matrix multiplication* $\mathsf{MatMul}(A, B)$ of *arbitrary* matrices, with $1 + o(1)$ multiplicative overhead compared to naïve MatMul. We conjecture that our protocol has optimal security, in the sense that a malicious prover cannot obtain any significant advantage over an honest prover. This conjecture reduces the hardness of our protocol to the task of solving a batch of correlated low-rank random linear systems, which is of independent interest.

Since matrix multiplications are the bottleneck of AI compute as well as countless industry-scale applications, this primitive suggests a concrete design of a new L1 base-layer protocol which nearly eliminates the energy waste of Bitcoin mining, allowing GPU consumers to *reduce their AI training and inference costs* by "re-using" them for blockchain consensus, in exchange for block rewards (2-for-1).

2012 ACM Subject Classification Security and privacy → Cryptography; Theory of computation → Cryptographic protocols; Theory of computation → Problems, reductions and completeness

Keywords and phrases Proof of work, blockchain, matrix multiplication, fine-grained complexity, random self-reducibility

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.2

Category Invited Talk

Acknowledgements We thank Yonatan Sompolinsky for multiple discussions on the topic of this work and for valuable comments on the manuscript. We thank Ohad Klein for valuable feedback and particularly for suggesting the FMM-based algorithm for computing the intermediates. We thank Eylon Yogev and Mark Zhandry for useful remarks and suggestions on this manuscript.

1 Introduction

The concept of proofs of work (PoW), i.e., easy-to-check proofs of computational effort, was proposed in 1992 in a seminal work of Dwork and Naor [14]. Their original motivation was to discourage spam email by requiring the sender to compute a moderately hard – but not intractable – function in order for the email to go through. This idea is far more general than combating spam; it can be used as a method for limiting access to any resource.

A concretely efficient and very simple instantiation of a PoW (in the random oracle model) was proposed by Back in 1997 under the name Hashcash and formalized five years later [4]. One way to describe Hashcash is via a protocol between a prover and a verifier,

wherein the verifier eventually becomes convinced that the prover did a moderate amount of work. The protocol works with a cryptographic hash function H, such as SHA256 or SHA3. At a high level, the verifier sends a random string σ as a challenge and the goal of the prover is to find a string x such that $H(\sigma\|x)$ starts with t leading zeros. Verification is very easy – just count leading zeros – and the soundness of the protocol relies on the fact that it is impossible to find such an x faster than by brute-force search, assuming H is a random oracle.

Perhaps the most well-known application of Back's PoW idea is in the Bitcoin network [24, 18]. Instead of deterring malicious email senders, Bitcoin's PoW enables competitive mining and thereby secures the network. A Bitcoin miner collects unconfirmed transactions to form a block, and a block is accepted by the network only if its hash meets a certain difficulty target (essentially described by the number of leading zeros in the hash output).

1.1 PoW Criticism

Undoubtedly, the realization of the Bitcoin network is a revolutionary achievement in distributed computing. Notwithstanding, a significant setback and common point of criticism is its energy-intensive nature and exorbitant environmental costs (consuming over 2% of annual electricity in the US alone).[1] To make matters worse, the continual acquisition of specialized hardware (ASICs) to retain competitive mining proliferates electronic waste, as these devices quickly become obsolete.

Thus, an outstanding question is whether the mining process must be so wasteful and serve no purpose beyond securing the blockchain. A similar question was raised already by Dwork and Naor [14]:

> *"Finally, the evaluation of the pricing function serves no useful purpose, except serving as a deterrent. It would be exciting to come up with a scheme in which evaluating the pricing function serves some additional purpose."*

1.2 Proofs of Useful Work

The above challenge is nowadays commonly referred to as a *proof of useful work* (PoUW). Intuitively, a proof of work is *useful* if it simultaneously satisfies three properties:

- *Economic value*: the results of computations performed by miners should be of interest, and moreover someone must be willing to pay for the result of the computation (formally, a task is κ-useful if someone is willing to pay κ dollars for its completion).
- *Efficiency*: the algorithm used to solve the problems is essentially optimal, ensuring there is no "waste" in mining compared to solving the task in an external setting.
- *Security*: it should be infeasible to fool the verifier into believing a problem was solved using bounded resources while it was not.

The first formal treatment of PoUW we are aware of is due to Ball et al. [5], who suggested a proof of work (in the plain model) based on the conjectured hardness of various fine-grained algorithmic problems. The proof was termed useful because the miner could choose the instance of the task itself. Unfortunately, the overhead of the honest prover was significant, requiring the prover to solve poly-logarithmically many instances of the task just to convince the verifier that a single instance was solved. In a follow-up version [6] the authors retracted all claims about "usefulness."

[1] `https://ccaf.io/cbnsi/cbeci/ghg/comparisons` (accessed 02.04.2025).

The absence of a PoUW (prior to this work) is not for lack of trying: it has been the holy grail of distributed consensus since the early days of Bitcoin, as it avoids the *tradeoff* between resource-efficiency and security present in proof-of-stake (PoS) and other energy-efficient alternatives. As Vitalik Buterin, co-founder of Ethereum, articulates in his 2019 blog post:[2]

> *"If either an efficiently verifiable proof-of-computation for Folding@home can be produced, or if we can find some other useful computation which is easy to verify, then cryptocurrency mining could actually become a huge boon to society ... even being socially beneficial by providing a public good."*

In the same post, Buterin adds that the challenge of obtaining a proof of useful work is "probably not feasible"; the transition of Ethereum from PoW to PoS is, to a large extent, attributed to this impossibility conjecture. We refer to a survey of Dotan and Tochner [13] for prior approaches, identifying for each one which properties of a "non-wasteful" and "truly useful" proof of work it fails to satisfy.

1.3 Our Contributions

Our main contribution is the first construction of a *truly useful* PoW protocol, for the omnipresent task of *matrix multiplication* (MatMul). At a high level, our (GPU-compatible) protocol re-uses native $\mathsf{MatMul}(A, B)$ computations of *arbitrary* matrices (submitted by the miner at her own discretion), to produce a verifiable PoW certificate obeying Nakamoto's Poisson distribution, *with* $1 + o(1)$ *multiplicative overhead* over the worst-case time for computing $A \cdot B$ (e.g., $(1 + o(1))n^3$ for square matrices, assuming naïve MatMul is the baseline). We argue why MatMul satisfies the PoUW axioms:

- *Economic value*: matrix multiplication is the backbone of many industry-scale applications. In particular, it is (by far) the bottleneck of AI training and inference, where giant MatMuls (as large as 40,000 × 40,000 in LLMs) are required for both forward and backward propagation, and are the primary reason for the exorbitant electricity costs of AI (cf. GPT-4 training estimated at $300M, with the next model projected to cost far more). Beyond AI, matrix multiplication is central to vector databases, quantum simulation, graphics rendering, and statistical physics, to mention a few.
- *Efficiency*: our method has minimal overhead over direct matrix multiplication. The running time is $N \cdot (1 + o(1))$, where N is a well-known baseline for matrix multiplication (in practice, $N = O(n^3)$), with no hidden large constants.
- *Security*: soundness relies on a hardness assumption about a certain *direct-product* problem of *correlated* instances (essentially, random low-rank linear systems). Under this conjecture, no adversary with comparable runtime can obtain non-trivial advantage. In particular, multiplying the all-zeroes matrices $A = B = \mathbf{0}^{n \times n}$ using our protocol is *as hard as any other product*.

1.3.1 MatMul Algorithms

The discovery of Fast Matrix Multiplication (FMM) algorithms by Strassen [26] and subsequent improvements (starting with [12] and most recently [28, 2]), allowing one to multiply $n \times n$ matrices in sub-cubic time $n^\omega \ll n^3$, had a profound theoretical impact. Unfortunately, FMM algorithms are unlikely to be practical on any imaginable hardware, as they are highly

[2] `https://vitalik.eth.limo/general/2019/11/22/progress.html`, Bullet 7.

asymptotic (earning the name "galactic algorithms") or require significant memory. Since our work is meant to serve as a practical alternative to Bitcoin, our analysis assumes *naïve* MatMul as the most practically-efficient baseline for $n \times k$ and $k \times m$ matrices, running in $\Theta(nkm)$ time. We stress that our security analysis holds *even in the presence of FMM algorithms*, assuming they are used to attack the protocol itself.

1.3.2 A Formal Definition and Framework

As an independent contribution, and to the best of our knowledge, we formulate the first non-trivial definition of proofs of work associated with computational tasks. This definition allows reasoning about different schemes and comparing them to one another (Section 3).

1.4 Related Work

There have been various proposals to eliminate the wastefulness of PoW mining by requiring the "waste" of a different resource. Most well known is "proof of stake," where participants lock up money and receive voting rights proportional to their holdings (e.g., [7, 20, 23]). Other resources, such as space [15, 25, 11], have been studied and deployed. While these avoid energy waste, they incur waste in other domains and introduce new economic and cryptographic tradeoffs.

Trapdoor Matrices and Algorithmic Speedup

In two recent works, Vaikuntanathan and Zamir [27] and Braverman and Newman [9] discovered techniques for sampling pseudorandom matrices such that a party holding a secret key can multiply them in near-linear time, relying on standard assumptions (learning parity with noise). This goal is incomparable and orthogonal to proofs of work, and the techniques seem unrelated: in a PoUW we want hardness to be *uniform* for everyone, even for those who choose the inputs, and there is no room for hidden trapdoors, since all computation is done locally by the party doing it. Whether ideas from these works have any indirect bearing on our protocol is an interesting question.

2 Overview of Our PoUW

We first recall the syntax and properties we want in a PoUW, focusing on matrix multiplication. In this overview we ignore economic value and focus on the technical aspects. A PoUW consists of two algorithms, Solve and Verify:

- $\mathsf{Solve}(\sigma, A, B)$ takes a seed σ and two $n \times n$ matrices A, B (the instance), and outputs a matrix C and a proof π. *Prover efficiency*: if $t(n)$ is the time to multiply A and B, then Solve runs in time $t(n) \cdot (1 + o(1))$; *proof size*: $|\pi| = o(t(n))$.
- $\mathsf{Verify}(\sigma, \pi)$ takes a seed σ and a proof π, and outputs a bit.

Correctness says that $C = A \cdot B$ (with probability 1) and $\mathsf{Verify}(\sigma, \pi) = 1$ with probability ε (over the randomness of Solve and a uniformly random σ). *Hardness* says that no procedure running in time $t(n) \cdot (1 + o(1))$ can create proofs accepted with probability noticeably higher than ε; that is, with the same computational budget there is no way to noticeably outperform the honest prover.

2.1 The Key Idea

The key idea underlying our PoUW is to encode noise into the input matrices, ensuring that any attempt to "shortcut" the computation would require effort comparable to performing the full computation. At the same time, we design an optimally efficient random self-reduction [8], incurring negligible cost. The latter is obtained by leveraging not only the task of producing an output, but – more importantly – enforcing unpredictability in the *transcript* of the computation.

2.2 A MatMul Random-Self-Reducibility-Based PoW

To gain intuition, we first design a proof of work based on multiplying two arbitrary matrices that is *not useful*. The idea is to ask the prover to compute a "noisy" matrix product instead of the original one:

$$C' = (A + E) \cdot (B + F), \tag{1}$$

where E and F are uniformly random $n \times n$ matrices. It is necessary for E and F to be *unpredictable* by the (malicious) prover, as otherwise it could pre-process the result of (1). This is achieved by deriving E and F from a random oracle applied to σ, A, B. We turn this into a proof of (non-useful) work: the prover computes $E, F = \mathcal{O}(\sigma, A, B)$, then $C' = (A + E)(B + F)$, then $z = \mathcal{O}(C')$, and outputs $\pi = (A, B, z)$; the verifier recomputes z and checks that it is below a threshold, so that $\Pr[\text{accept}] = \varepsilon$. Since E, F are uniform, C' is uniform no matter what A, B are, and no attacker can compute z faster than a generic matrix multiplication (else it would multiply *random* matrices in time $o(t(n))$, a major breakthrough).

This PoW is not useful: the prover only outputs the noisy value C', not $A \cdot B$. Naïvely, one could compute the "noise" term $A \cdot F + E \cdot (B + F)$ and subtract it to recover $C = A \cdot B$ – but this requires two additional generic matrix products, so the honest prover performs 3 products (overhead $\alpha \geq 3$) while a malicious prover still does only one product. Hence this scheme is not a PoUW.

2.3 Usefulness via Transcript Unpredictability

The issue was that "peeling off" the noise was too expensive. Instead of sampling E, F as uniform, we sample them as *low rank* with parameter $r \ll n$: draw $E_L, F_L \in \mathbb{F}_q^{n \times r}$ and $E_R, F_R \in \mathbb{F}_q^{r \times n}$ and set $E = E_L E_R$, $F = F_L F_R$. Now peeling off the noise is cheap: computing $A \cdot F + E \cdot (B + F)$ involves products in which one factor is low rank, costing $O(n^2 r) = o(n^3)$.

However, with low-rank noise the *output* C' can no longer serve as a source of hardness: a malicious prover can choose $A = B = \mathbf{0}$, so its task reduces to multiplying the low-rank E and F, which costs only $O(n^2 r)$. We make the following novel observation: **instead of using *the output* of the computation as the proof of work, we use *the transcript* of the computation.** Consider multiplying $A' = A + E$ and $B' = B + F$ with a block variant of the classical algorithm: split into $r \times r$ blocks and iteratively compute

$$C'^{(\ell)}_{i,j} := C'^{(\ell-1)}_{i,j} + A'_{i,\ell} \cdot B'_{\ell,j}, \qquad i, j, \ell \in [n/r].$$

The sequence of $(n/r)^3$ intermediate matrices $\{C'^{(\ell)}_{i,j}\}$ is the *transcript* of the product. Our proof of work applies the random oracle to the transcript rather than to the output. The reason this yields hardness even for $A = B = \mathbf{0}$ is that each intermediate essentially depends

on the product of two $r \times r$ blocks of A' and B', which – because E, F are uniformly random low-rank matrices – are *marginally uniform*. Thus computing a single intermediate requires an $r \times r$ random matrix multiplication, and computing all of them requires solving a correlated batch of such problems, which we conjecture is as hard as performing the full computation.

The resulting scheme is efficient (encoding/decoding cost $O(n^2 r)$, hashing a transcript of size n^3/r^3) and, under our conjecture, optimally secure. We give the formal scheme and its two instantiations in Section 4, and detailed remarks on memory, verifier complexity, and the use of SNARKs to compress verification below.

▶ Remark 1 (Memory complexity). As described, storing the whole transcript is expensive. Instead of hashing the whole transcript, one can hash each of the $(n/r)^3$ intermediate matrices separately and use each as a potential proof; this also makes the number of "lottery tickets" scale with the matrix size, a useful feature when supporting varying matrix sizes. The same hardness argument applies, since each intermediate is a product of two marginally uniform matrices.

▶ Remark 2 (Verifier complexity and SNARKs). Our protocol is intended for the standard PoW setting, where a prover performs many independent attempts ("lottery tickets") until one succeeds, and verification is performed only for the single successful attempt. A single attempt corresponds to (roughly) one transcript computation for a fresh challenge and succeeds with tunable probability ε; thus the *amortized* cost of verification is negligible, exactly as in Bitcoin. To further reduce verifier cost, one can delegate verification to the prover by attaching a SNARK certifying that the verifier would have accepted; if privacy of A, B is desired, a zkSNARK can be used. Since this happens only when a block is mined (a rare event), the cost is amortized away.

3 A Definitional Framework for Proofs of Useful Work

A proof of useful work is a method for a prover to convince a verifier that it performed sufficient computation to solve a particular computational task. Importantly, the task the prover chooses is arbitrary – perhaps chosen by the prover itself – and does not come from any pre-specified distribution. Soundness requires that any (malicious) prover running sufficiently faster than the honest prover cannot fool the verifier with high probability, even if it fully controls the task it solves.

We associate the proof of work with a function $f\colon \{0,1\}^n \to \{0,1\}^n$ reflecting the computation being performed, and assume a canonical algorithm $\textsc{Alg}$solving f with worst-case complexity $t(n)$. All of our protocols are public-coin, so by the Fiat–Shamir heuristic [16] we may work in the random oracle (RO) model, where all parties have query access to a public random function $\mathcal{O}$. We let $\lambda \in \mathbb{N}$ be a security parameter and think of $n \gg \lambda$. Syntactically, a PoUW consists of two oracle-aided algorithms:

1. $\mathsf{Solve}(\sigma, \mathsf{x})$: a probabilistic algorithm taking a seed $\sigma \in \{0,1\}^\lambda$ and an instance $\mathsf{x} \in \{0,1\}^n$, outputting a solution $\mathsf{y} \in \{0,1\}^n$ and a proof $\pi \in \{0,1\}^*$.
2. $\mathsf{Verify}(\sigma, \pi)$: a deterministic algorithm taking a seed and a proof, outputting 0 or 1.

The role of σ is to guarantee that the work is "fresh," i.e., executed after σ became public, to avoid preprocessing.

▶ **Definition 3** (Proof of useful work). *An* ε-difficulty *and* α-overhead *PoUW for* f *relative to* $\textsc{Alg}$ *consists of two algorithms* $\mathsf{Solve}, \mathsf{Verify}$ *satisfying:*

Efficiency. *for any* σ *and any* $\mathsf{x} \in \{0,1\}^n$, $\mathsf{Solve}(\sigma, \mathsf{x})$ *runs in time* $t(n) \cdot (1 + \alpha(n))$ *and* $\mathsf{Verify}(\sigma, \pi)$ *runs in time* $c \cdot t(n)$ *for a constant* $c \in (0,1)$.

Completeness. *for every* λ, n*, every* σ*, and every* x*,*

$$\Pr[f(\mathsf{x}) = \mathsf{y}] = 1 \quad \textit{and} \quad \Pr[\mathsf{Verify}(\sigma, \pi) = 1] \geq \varepsilon(n),$$

where the probabilities are over $\mathsf{y}, \pi \leftarrow \mathsf{Solve}(\mathsf{x}, \sigma)$ *and the random oracle.*

Hardness. *there is a constant* $C > 0$ *such that for any algorithm* $\mathsf{Solve}^\star$ *running in time* $t' \leq t(n)$ *(with polynomial-time preprocessing),*

$$\Pr[\mathsf{Verify}(\sigma, \pi) = 1] < \max\left\{C \cdot \tfrac{t' \cdot \varepsilon(n)}{t(n)},\, 2^{-\lambda}\right\},$$

where the probability is over $\sigma \leftarrow \{0,1\}^\lambda$*,* $\pi \leftarrow \mathsf{Solve}^\star(\sigma)$*, and the random oracle.*

One can introduce stronger notions considering amortization across parallel instances, or streaming challenges where the adversary need only show a slight advantage over the naïve process; the latter is what is needed for Bitcoin-style blockchains, and we formalize it in Appendix B. We conclude with three remarks.

▶ Remark 4 (Tunable difficulty). In applications one must tune hardness. This can be done by fixing ε and varying the size of x, or by fixing n and tuning ε (requiring "more instances" for the same success probability). In our final protocol both n and ε are tunable.

▶ Remark 5 (Prover's efficiency). The overhead α should be as small as possible, to incentivize running Solve rather than solving f and searching for π separately. The latter already gives a PoUW with constant multiplicative overhead (see Section 3.1); the interesting regime is $\alpha(t(n)) = o(1)$, which our protocol achieves.

3.1 A (Trivial) PoUW for Any Function

A PoUW exists for any f with an associated Alg running in time $t(n)$, albeit with large overhead. The scheme runs computation and proof of work independently: $\mathsf{Solve}(\sigma, \mathsf{x})$ runs Alg to get y and, independently, chooses $T = c \cdot t(n)$ nonces $\omega_1, \ldots, \omega_T$, computes $z_i = \mathcal{O}(\sigma, \omega_i)$, and sets π to be the ω_i whose z_i has the most leading zeros; $\mathsf{Verify}(\sigma, \omega)$ accepts iff $\mathcal{O}(\sigma, \omega)$ has $\log(t(n))$ leading zeros.

▷ Claim 6 (A PoUW for f). The above is a PoUW with overhead $\alpha = c$ and difficulty $\varepsilon = 1 - e^{-c}$.

Proof. Since $\mathcal{O}$ is a random oracle, its output on fresh points is uniform in $\{0,1\}^\lambda$. Overhead is $\alpha = c$ by construction; the success probability of a single query is $p = 1/t(n)$, so all T attempts fail with probability $(1-p)^{c \cdot t(n)} \leq e^{-c}$, giving completeness. Hardness follows from the RO property: "guessing" a nonce whose output has $\log T$ leading zeros amounts to guessing a random oracle output without querying it. ◁

This protocol is not useful: either P has linear overhead over Alg, or a malicious $P^\star$ (which only searches nonces and multiplies nothing) gains a linear advantage. Concretely $t_{P^\star}(n) = c \cdot t(n)$, a factor $(1 + 1/c)$ faster than $t_P(n)$. Setting $c = \omega(1)$ makes the advantage sub-linear but blows up the honest overhead. This tension is exactly what our matrix-multiplication scheme resolves.

4 A PoUW for Matrix Multiplication

Here the task consists of two $n \times n$ matrices A, B with entries in $\mathbb{F}_q$, and the solution is $C = A \cdot B$.[3] We first present the canonical algorithm we assume (MatMul), then abstract the required primitive (a *transcript-unpredictable encoding*), and finally give two instantiations – the second of which is an optimal PoUW under a new algorithmic conjecture.

4.1 The Canonical MatMul Algorithm

We describe the textbook algorithm, generalized to operate on $r \times r$ blocks ("tiles") instead of single entries; see Algorithm 1.

Algorithm 1. *Canonical Matrix Multiplication* $\textsc{MatMul}_r(A, B)$

// $A \in \mathbb{F}_q^{n \times k}$, $B \in \mathbb{F}_q^{k \times m}$, $r \in [n]$, and $r \mid n, k, m$.

1. Initialize an all-0 matrix $C^{(0)} \in \mathbb{F}_q^{n \times m}$.
2. Partition A, B, C into $r \times r$ blocks; the (i, j)th block of X is $X_{i,j}$.
3. For each $i \in [n/r]$, $j \in [m/r]$, $\ell \in [k/r]$, compute $C_{i,j}^{(\ell)} := C_{i,j}^{(\ell-1)} + A_{i,\ell} \cdot B_{\ell,j}$, where $A_{i,\ell} \cdot B_{\ell,j}$ is computed by any classical algorithm.
4. Output $C^{(k/r)}$.

The complexity of $\textsc{MatMul}_r$ is governed by the nkm/r^3 intermediate values. Computing them naïvely gives $t_{\textsc{MatMul}}(n, k, m) = O(nkm)$; for $n = k = m$ we write $t_{\textsc{MatMul}}(n)$. The transcript of an execution consists of all intermediate blocks.

▶ **Definition 7** (Transcript). *The transcript of* $\textsc{MatMul}_r(A, B)$ *is* $\mathsf{Tr}(\textsc{MatMul}_r, A, B) = \{C_{i,j}^{(\ell)}\}_{i \in [n/r],\, j \in [m/r],\, \ell \in [k/r]}$.

For $n = m = k$: if $r = n$ there is a single intermediate (the output); if $r = n/2$ there are 8; if $r = 1$ there are n^3, each a single field element.

Algorithms for the Intermediates

One can compute all nkm/r^3 intermediates directly ($O(r^3)$ each, $O(nkm)$ total), but FMM does better. Using fast *square* multiplication, each intermediate costs r^ω (with $\omega = 2.371552$ [28]), for total $O(nmk/r^{3-\omega})$. Using fast *rectangular* multiplication is often better: for fixed ℓ, the matrix $\{C_{i,j}^{(\ell)}\}$ is the product of a column-strip of A and a row-strip of B; letting ω_s denote the exponent of multiplying $n \times s$ by $s \times n$ matrices, we get $(n/r) \cdot n^{\omega_r} = n^{\omega_r + 1}/r$, which can be $o(n^3)$. Concretely, an $n \times n^s$ by $n^s \times n$ product for $s \leq 0.321334$ takes $n^{2+o(1)}$ time [17, 28], so for $r = n^{0.3}$ all intermediates can be computed in $n^{2.7+o(1)}$ time. In practice, however, the direct method remains fastest for realistic n, as the alternatives have large constants or heavy memory/sequentiality costs.

4.2 A Transcript-Unpredictable Encoding Scheme

Our template relies on a generic invertible operation we call a *transcript-unpredictable encoding*. The encoding injects noise so that the transcript of the perturbed matrices has non-trivial entropy; the inverse "peels off" the noise to recover the original output. Both operations are parameterized by the same $r \in [n]$ as $\textsc{MatMul}_r$.

- $A', B' \Leftarrow \textsc{Encode}_r(\sigma, A, B)$: given a seed and two matrices, output two matrices A', B'.
- $C \Leftarrow \textsc{Decode}_r(\sigma, C')$: given a seed and a matrix C', output a matrix C.

[3] The protocol extends to rectangular matrix multiplication; we favor simplicity of presentation.

▶ **Definition 8** (Transcript-unpredictable encoding). *A pair* $(\textsc{Encode}, \textsc{Decode})$ *is a transcript-unpredictable encoding if:*

Completeness. *For all* A, B *and auxiliary* r, $\Pr_\sigma[\textsc{Decode}_r(\sigma, A' \cdot B') = A \cdot B \mid A', B' = \textsc{Encode}_r(\sigma, A, B)] = 1$.

ε-transcript unpredictability. *There is a constant* $C > 0$ *such that for all* $n \times n$ *matrices* A, B*, every* r*, and every iterative numerical algorithm* $\mathcal{A}$ *running in time* $t = t(n)$*, the probability that* $\mathcal{A}(A', B')$ *outputs the full transcript* $\{C^{(\ell)}_{i,j}\}$ *of* $\textsc{MatMul}_r(A', B')$ *(where* $A', B' = \textsc{Encode}_r(\sigma, A, B)$*) is at most* $C \cdot \frac{t(n)}{t_{\textsc{MatMul}}(n)} \cdot \varepsilon(n)$*, over* $\sigma \leftarrow \{0,1\}^\lambda$ *and the internal randomness of* $\mathcal{A}$.

Satisfying either property alone is trivial; the challenge is to have both simultaneously. We give an even more general abstraction – *unpredictable encodings*, of independent interest – in Appendix C.

4.3 From Transcript Unpredictability to a PoUW

Transcript unpredictability implies that computing the whole transcript essentially requires performing the direct computation in time $t_{\textsc{MatMul}}(n)$, which directly yields a PoUW (Algorithm 2).

Algorithm 2. *A PoUW for Matrix Multiplication with parameter* r

$\underline{\mathsf{Solve}(\sigma, A, B)}$: // $A, B \in \mathbb{F}_q^{n \times n}$

1. Compute $A', B' = \textsc{Encode}_r(\sigma, A, B)$.
2. Compute $C' = \textsc{MatMul}_r(A', B')$; let $z = \{C^{(\ell)}_{i,j}\}_{i,j,\ell \in [n/r]}$ be the intermediate blocks.
3. Compute $C := \textsc{Decode}_r(\sigma, C')$.
4. Output (C, π) where $\pi = (A, B, z)$.

$\underline{\mathsf{Verify}(\sigma, \pi)}$:

1. Parse $\pi = (A, B, z)$.
2. Recompute from σ, A, B the correct value of z; output 1 iff z is correct.

Correctness and Hardness

Correctness is immediate: $C = A \cdot B$ by correctness of $(\textsc{Encode}, \textsc{Decode})$, and the verifier repeats the honest computation, ending with the same z. For hardness, any $P^\star$ running in time strictly less than $t_{\textsc{MatMul}}(n)$ fools the verifier with probability at most ε, directly by the ε-transcript unpredictability of $(\textsc{Encode}, \textsc{Decode})$.

4.4 A First Instantiation

Our first instantiation uses $r = 1$ and the classical random self-reducibility of matrix multiplication (Algorithm 3).

Algorithm 3. *First implementation of* Encode *and* Decode

$\underline{\textsc{Encode}_r(\sigma, A, B)}$: parse $\sigma = E, F$ with $E, F \in \mathbb{F}_q^{n \times n}$; output $A' = A + E$, $B' = B + F$.

$\underline{\textsc{Decode}_r(\sigma, C')}$: parse $\sigma = E, F$; compute $C'' = A \cdot F + E \cdot (B + F)$ (two MatMuls and an addition); output $C = C' - C''$.

Completeness holds since $C' - C'' = (A + E)(B + F) - \big(A \cdot F + E \cdot (B + F)\big) = A \cdot B$. Since $r = 1$, the transcript consists only of the output C', and transcript unpredictability reduces to the assumption that multiplying random matrices is hard.

▶ **Assumption 9.** *There is no algorithm computing the product of two random $n \times n$ matrices in time $o(n^\omega)$.*

The downside is that a malicious solver can skip Decode altogether, saving two matrix multiplications: the honest solver does at least three times the necessary work ($\alpha \geq 3$), while $\mathsf{Solve}^\star$ does essentially $t_{\text{MatMul}}(n)$. Hence this scheme is not useful.

4.5 A Second Instantiation

Our second instantiation uses low-rank noise and a more novel hardness assumption (Algorithm 4).

Algorithm 4. *Second implementation of* Encode *and* Decode

$\underline{\text{Encode}_r(\sigma, A, B)}$: parse $\sigma = E_L, E_R, F_L, F_R$ with $E_L, F_L \in \mathbb{F}_q^{n\times r}$ and $E_R, F_R \in \mathbb{F}_q^{r\times n}$; compute $E = E_L E_R$, $F = F_L F_R$ (two MatMuls); output $A' = A + E$, $B' = B + F$.

$\underline{\text{Decode}_r(\sigma, C')}$: parse σ as above; compute $C'' = (AF_L)F_R + E_L\big(E_R(B + F_L F_R)\big)$ (five MatMuls, each involving a thin factor); output $C = C' - C''$.

Completeness holds since $C'' = A \cdot F + E \cdot (B + F)$ and hence $C' - C'' = A \cdot B$. Here r is a parameter, so the transcript consists of $(n/r)^3$ intermediate $r \times r$ matrices, and E, F are uniformly random conditioned on being rank r (Section 4.6). The relevant question is whether the transcript of $\text{MatMul}_r(A', B')$ can be computed without essentially performing the computation of Section 4.1.

▶ **Assumption 10.** *There is no algorithm computing all intermediate values when multiplying two random $n \times n$ rank-r matrices in time $o(n^{\omega_r+1}/r)$.*

The algorithm of Section 4.1 multiplies any two matrices – in particular random rank-r ones – in $O(n^{\omega_r+1}/r)$ time. One might hope to exploit the low-rank structure: writing $A = A_L A_R$, $B = B_L B_R$, one can form partial sums $T_z = \sum_{i=1}^{z} A_R B_L$ in r-blocks and then compute $A_L\, T_z\, B_R$ by fast rectangular multiplication (n^{ω_r} each), but there are n/r values of z, giving the same $O(n^{\omega_r+1}/r)$. Moreover, since $\Theta(n^3/r)$ bits are needed just to write down all intermediates and $\omega_r = 2$ for small r, the assumption holds with state-of-the-art rectangular FMM.

Efficiency of the PoUW

Unlike our previous schemes, this one is useful: the honest solver performs exactly one product of two $n \times n$ matrices, since all products in Encode and Decode involve an $n \times r$ (or $r \times n$) factor and cost $O(n^2 r)$. The overhead is

$$\alpha(n) = O(n^2 + n^2 r)/t_{\text{MatMul}}(n) = o(1)$$

(unless FMM is used as the baseline). By Assumption 10, no algorithm runs in time $o(t_{\text{MatMul}}(n))$, making the protocol a truly useful PoW.

4.6 Properties of Our Noise Matrices

The noise generation in Algorithm 4 yields a uniformly random rank-r matrix. Let $\mathcal{E}_{r,n}$ be the set of $n \times n$ rank-r matrices.

Table 1 Benchmarked layer shapes.

Layer	M	N	K
Llama70B Attn WO	32768	8192	8192
Llama70B Attn QKV	32768	10240	8192
Llama70B MLP GateUp	32768	57344	8192
DeepSeek-V3 Attn WO	32768	7168	16384
Llama70B MLP Down	32768	8192	28672

Table 2 End-to-end median latency. "vs. cuBLAS" and "vs. vLLM" give the additional latency of cuPoW relative to each baseline; positive means cuPoW is slower, negative means faster.

Layer	cuBLAS (ms)	vLLM (ms)	cuPoW (ms)	vs. cuBLAS (%)	vs. vLLM (%)
Llama70B Attn WO	3.52	3.59	4.39	+24.7	+22.2
Llama70B Attn QKV	4.79	4.53	5.46	+20.5	+13.9
Llama70B MLP GateUp	26.21	41.69	28.25	+7.8	−32.3
DeepSeek-V3 Attn WO	5.44	7.17	7.31	+34.3	+1.9
Llama70B MLP Down	10.68	15.92	13.90	+30.1	−12.7

▶ **Lemma 11.** *In the random oracle model, assuming σ is unpredictable, the induced distribution of E and F is uniform over $\mathcal{E}_{r,n}$ (up to small statistical error). In particular, every $r \times r$ submatrix of E and F is marginally close to uniform.*

The proof follows from the fact that a random $n \times r$ matrix over $\mathbb{F}_q$ is full-rank except with probability $O(q^{-(n-r+1)})$; we defer it to Appendix D.

5 Benchmarks and Evaluation

We benchmark cuPoW's end-to-end MatMul implementation against two common INT8 MatMul baselines, cuBLAS and vLLM, on production LLM layer shapes.[4] All measurements use `torch==2.9.1` built against the local CUDA toolkit. We report median latency over 100 timed iterations after 25 warm-up iterations, using CUDA events; to reduce cache effects, each launch is preceded by a 64 MB L2 flush and inputs are rotated across 10 independent copies. The concrete hash $\mathcal{O}$ is BLAKE3, and a few adaptations were made due to hardware constraints. We compare `cublasGemmEx` (INT8), `cutlass_scaled_mm` (tensor-wise INT8, as in vLLM), and the full cuPoW path (commitment hashing, noise generation and application, transcript hashing, and denoising), with rank $r = 64$. The benchmark uses production dimensions from Llama-3.3-70B and DeepSeek-V3 with batch dimension $M = 32768$ (Table 1).

Table 2 reports end-to-end median latency. Two patterns stand out. First, cuPoW stays reasonably close to cuBLAS even though it performs substantially more work than a plain GEMM. Second, cuPoW is clearly more efficient than vLLM on the large MLP projections: it is 32.3% faster on `Llama70B MLP GateUp` and 12.7% faster on `Llama70B MLP Down`. On the attention-style layers, vLLM is faster on two cases and essentially tied on the DeepSeek-V3 attention output.

[4] In the code base, cuPoW is called `PearlGEMM`.

Table 3 Per-stage median latencies for cuPoW. Stage medians are measured independently, so they need not sum exactly to the end-to-end median in Table 2.

Layer	Commit (ms)	NoiseGen+Apply (ms)	GEMM (ms)	Denoise+Reduce (ms)
Llama70B Attn WO	0.38	0.24	3.36	0.40
Llama70B Attn QKV	0.42	0.58	4.09	0.81
Llama70B MLP GateUp	0.76	1.07	23.99	3.19
DeepSeek-V3 Attn WO	0.66	0.97	5.53	0.71
Llama70B MLP Down	1.12	1.75	11.07	1.13

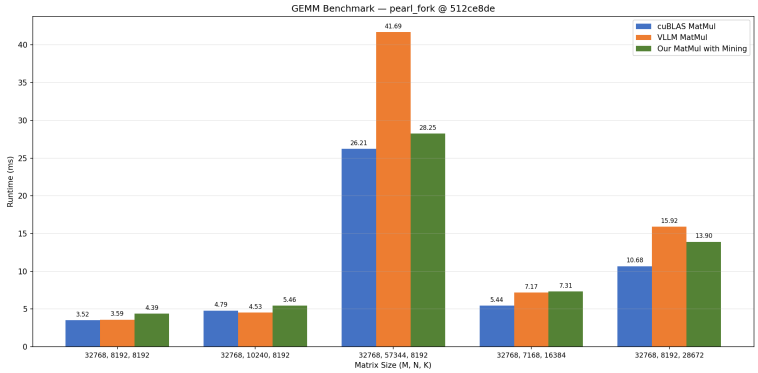

Figure 1 End-to-end benchmark comparison for cuBLAS INT8, vLLM INT8, and cuPoW.

Table 3 breaks the cuPoW pipeline down by stage. The dominant cost remains the GEMM itself, especially on the larger MLP layers; on `Llama70B MLP GateUp` the GEMM stage is 23.99 ms, while hashing/commitment is 0.76 ms, noise generation and application 1.07 ms, and denoising plus reduction 3.19 ms. The added work is visible but bounded, which is why cuPoW stays close to plain INT8 baselines even while running the full mining path.

The overall message is straightforward: cuPoW does not match raw cuBLAS on absolute latency, but it stays in the same performance regime while adding the full mining and verification-oriented pipeline (Figure 1). That tradeoff is especially favorable on the largest MLP layers, where cuPoW is substantially faster than vLLM and only modestly behind cuBLAS.

6 Open Problems and Future Directions

The core idea of this work is to perform PoW on *intermediate* computations rather than on the *output* of the task $A \cdot B$. Once one subscribes to this idea and to a direct-product security assumption, it is natural to ask how general the approach is.

▶ **Problem 12.** *Which computational tasks $f(x)$ beyond* MatMul *admit an efficient $(1 + o(1)$ overhead) PoUW? Can we characterize the functions f for which such a scheme exists?*

Natural candidates include solving linear systems, graph search and routing, and string problems (e.g., DNA sequencing). Some of these lack a natural *decomposable* structure $f(x) = \bigotimes g(x_i)$ like MatMul, so it is less clear how to apply random self-reducibility on subproblems.

▶ **Problem 13.** *Is there a better PoUW for* MatMul *than Algorithm 4, either in* computational overhead *($O(n^2 r)$) or in* security assumption *(ideally both)?*

In Appendix A we present an alternative candidate with *self-canceling noise* that *preserves* the output AB (no denoising step at all), with $O(n^2 \log n)$ overhead, but which applies only to *high-rank* matrices and relies on a subtler assumption. It nonetheless demonstrates the flexibility of random self-reductions for MatMul.

▶ **Problem 14.** *Is there a PoUW scheme for useful operations from more standard or well-studied assumptions (e.g., program obfuscation)? And is there a PoUW for* MatMul *over the reals (accounting for rounding and floating-point effects)?*

Our scheme is defined over a field, and many LLMs (especially vision transformers, but also many open-source language models) still use integer inference; extending to floating-point is an important and challenging direction. The candidate in Appendix A is a promising starting point.

References

1 Nir Ailon and Bernard Chazelle. The fast johnson–lindenstrauss transform and approximate nearest neighbors. *SIAM Journal on Computing*, 39(1):302–322, 2009. doi:10.1137/060673096.

2 Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. *CoRR*, abs/2404.16349, 2024. doi:10.48550/arXiv.2404.16349.

3 Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in nc^0. *SIAM J. Comput.*, 36(4):845–888, 2006. doi:10.1137/S0097539705446950.

4 Adam Back et al. Hashcash-a denial of service counter-measure. Manuscript, 2002. Accessed 01.04.2025. URL: http://www.hashcash.org/papers/hashcash.pdf.

5 Marshall Ball, Alon Rosen, Manuel Sabin, and Prashant Nalini Vasudevan. Proofs of useful work. *IACR Cryptol. ePrint Arch.*, page 203, 2017. URL: http://eprint.iacr.org/2017/203.

6 Marshall Ball, Alon Rosen, Manuel Sabin, and Prashant Nalini Vasudevan. Proofs of work from worst-case assumptions. In *CRYPTO (1)*, pages 789–819. Springer, 2018. doi:10.1007/978-3-319-96884-1_26.

7 Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract]. *SIGMETRICS Perform. Evaluation Rev.*, 42(3):34–37, 2014. doi:10.1145/2695533.2695545.

8 Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, pages 549–595, 1993. doi:10.1016/0022-0000(93)90044-W.

9 Mark Braverman and Stephen Newman. Sublinear-overhead secure linear algebra on a dishonest server. *CoRR*, abs/2502.13060, 2025. doi:10.48550/arXiv.2502.13060.

10 Kenneth L. Clarkson and David P. Woodruff. Low rank approximation and regression in input sparsity time. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 81–90, New York, NY, USA, June 2013. doi:10.1145/2488608.2488620.

11 Bram Cohen and Krzysztof Pietrzak. The chia network blockchain. *White Paper, Chia. net*, 9, 2019. Accessed 1.04.2025. URL: https://www.chivescoin.org/wp-content/uploads/2021/10/ChiaGreenPaper.pdf.

12 Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987. doi:10.1145/28395.28396.

13 Maya Dotan and Saar Tochner. Proofs of useless work – Positive and negative results for wasteless mining systems. *CoRR*, abs/2007.01046, 2020. arXiv:2007.01046.

14 Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992. doi:10.1007/3-540-48071-4_10.

15 Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *CRYPTO (2)*, volume 9216 of *Lecture Notes in Computer Science*, pages 585–605. Springer, 2015. doi:10.1007/978-3-662-48000-7_29.

16 Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986. doi:10.1007/3-540-47721-7_12.

17 François Le Gall. Faster rectangular matrix multiplication by combination loss analysis. In *SODA*, pages 3765–3791. SIAM, 2024. doi:10.1137/1.9781611977912.133.

18 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. *J. ACM*, 71(4):25:1–25:49, 2024. doi:10.1145/3653445.

19 Juan A. Garay, Aggelos Kiayias, and Giorgos Panagiotakos. Consensus from signatures of work. In *CT-RSA*, pages 319–344. Springer, 2020. doi:10.1007/978-3-030-40186-3_14.

20 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. *IACR Cryptol. ePrint Arch.*, page 454, 2017. URL: http://eprint.iacr.org/2017/454.

21 Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *FOCS*, pages 294–304. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892118.

22 Kiran S. Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM Journal on Computing*, 40(6):1767–1802, 2011. doi:10.1137/08073408X.

23 Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *IEEE Symposium on Security and Privacy*, pages 157–174. IEEE, 2019. doi:10.1109/SP.2019.00063.

24 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Satoshi Nakamoto*, 2008. Accessed 1.04.2025. URL: https://bitcoin.org/bitcoin.pdf.

25 Sunoo Park, Albert Kwon, Georg Fuchsbauer, Peter Gazi, Joël Alwen, and Krzysztof Pietrzak. Spacemint: A cryptocurrency based on proofs of space. In *Financial Cryptography*, volume 10957 of *Lecture Notes in Computer Science*, pages 480–499. Springer, 2018. doi:10.1007/978-3-662-58387-6_26.

26 Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

27 Vinod Vaikuntanathan and Or Zamir. Improving algorithmic efficiency using cryptography. *CoRR*, abs/2502.13065, 2025. doi:10.48550/arXiv.2502.13065.

28 Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *SODA*, pages 3792–3835. SIAM, 2024. doi:10.1137/1.9781611977912.134.

A A Self-Canceling Noise Scheme

There is nothing "holy" about the low-rank structure of the noise in Algorithm 4: all we need is a *low-entropy* perturbation $(A, B) \mapsto (A', B')$ whose structure can be exploited to decode the *output* AB but not the *intermediate* computations – i.e., A', B' should have marginally uniform projections on $r \times r$ tiles.

A.1 Self-Canceling Noise via Pseudorandom Rotations

Using *pseudorandom rotations* on high-rank matrices lets us *avoid the decoding step altogether*. We randomly rotate A, B by a fast pseudorandom orthonormal matrix R – e.g., the randomized Hadamard transform [1], which enables $O(n^2 \log n)$-time rotations via the FFT – and run the tile-based scheme of Algorithm 2 on

$$A' := AR, \qquad B' := R^\top B.$$

Since $RR^\top = I_n$, the noise cancels at the *output* $A'B' = ARR^\top B = AB$ (no peeling needed), yet this cancellation does *not* occur locally on the partial sums

$$P_{I,K,J} = \sum_{i\in I,\, k\in K,\, j\in J} (AR)_{ik}(R^\top B)_{kj} \tag{2}$$

of intermediate $r \times r$ tiles ($|I| = |K| = |J| = r$). This scheme is useless if a dishonest miner chooses $A = B = \mathbf{0}$; but assuming the columns of A and rows of B have high rank ($\mathsf{rk}(A), \mathsf{rk}(B) \gtrsim n$, which can be enforced by adding a random permutation of the identity), the FastJL transform R guarantees that every $\sqrt{n} \times \sqrt{n}$ tile of AR is a random subspace embedding [10], so we may as well assume $A = B = I_n$.

A.2 Local vs. Global Structure of the FFT

Assuming $A = B = I_n$, computing the partial sums $(RR^\top)_{I,K,J} \in \mathbb{R}^{r\times r}$ appears hard, since the *global* structure of the randomized Hadamard matrix seems hard to exploit *locally* on submatrices: it is not known how to multiply arbitrary $r \times r$ submatrices of the FFT matrix by general $x \in \mathbb{R}^r$ in $o(r^2)$ time without massive preprocessing [22]. To ensure full marginal entropy of each tile, Algorithm 5 composes the Hadamard transform with a block-diagonal sign matrix D; block size $d = r$ suffices but costs $O(n^2(r + \log n))$, no faster than Algorithm 2. We conjecture that even block size $d = O(1)$ remains hard.

▶ **Conjecture 15.** *Let $A, B \in \mathbb{R}^{n\times n}$ be full-rank. Then the amortized cost of computing all n^3/r^3 partial sums (2) in Algorithm 5 with $r = \sqrt{n}$, $d = O(1)$, even with $O(n^3)$ preprocessing, requires $\Omega(r^3)$ time, in the word-RAM model with word size $w = O(\log n)$.*

Algorithm 5. *Random-rotation PoUW (self-canceling noise)* $\mathsf{rrPoW}_{r,d}(\sigma, A, B)$

// $A, B \in \mathbb{R}^{n\times n}$; r is the tile size and d the Hadamard block size.

1. Parse $\sigma = D = \mathsf{diag}(B_1, \ldots, B_{n/d})$, each $B_i \in \{-1, 1\}^{d\times d}$ with i.i.d. uniform ± 1 entries.
2. Let $R \leftarrow H_n \cdot D$, where $H_n \in \{\pm 1\}^{n\times n}$ is the Hadamard matrix [1].
3. Set $A' \leftarrow AR$ and $B' \leftarrow R^\top B$.
4. Run $\mathsf{Solve}_r(A', B')$ of Algorithm 2.

We leave further theoretical and practical exploration of the (pseudo-)random rotation scheme – in particular its extension to real arithmetic – for future work.

B Proof of Useful Work as a Poisson Process

For blockchains we require that the events $\{\mathsf{Verify} = 1\}$ follow a Poisson process, as in Bitcoin (a block every 10 minutes on average).

▶ **Definition 16** (Poisson process). *A stream of events $E_1, E_2, \ldots$ is a Poisson process with rate ρ if for every time T, interval $\overline{T}$, and k, $\Pr[\sum_{i=T+1}^{T+\overline{T}} E_i = k] = (\rho\overline{T})^k e^{-\rho\overline{T}}/k!$.*

▶ **Fact 17.** *For i.i.d. Bernoulli(p) variables $E_1, \ldots, E_N$, the sum $\sum_i E_i$ is Binomial(N, p), which converges to a Poisson distribution as $N \to \infty$ and Np tends to a finite limit.*

▶ **Definition 18** (PoUW for blockchains). *Let* $\mathsf{Solve}(\sigma, \mathsf{data}, \mathsf{x})$ *and* $\mathsf{Verify}(\sigma, \mathsf{data}, \pi)$ *be as in Section 3, augmented with a* data *field. They constitute a (ρ, α) proof of work if: (a) the honest mining process – repeatedly sampling σ_i, choosing $\mathsf{x}_i, \mathsf{data}_i$, and setting $E_i = \mathsf{Verify}(\sigma_i, \mathsf{data}_i, \mathsf{Solve}(\cdots))$ – forms a Poisson process with rate ρ; and (b) for every $T, \overline{T}$, any $(T, \overline{T}, \alpha)$-adversary that, given the honest transcript up to time T, runs for time $\alpha(n)\cdot(t(n)\cdot\overline{T})$ on fresh random seeds $\sigma_{T+1}, \ldots, \sigma_{T+\overline{T}}$, produces a stream that forms a Poisson process with rate at most ρ.*

B.1 Bitcoin's Proof of Work

Fix a time t and a polynomially long interval T. Since polynomially many random oracle evaluations are independent of one another and of any preprocessing, the number of block discoveries is Binomial(N, p) with $N = R \cdot T$ and $p = \kappa/2^\lambda$, where R is the network hash rate and κ a difficulty parameter. By Fact 17 this approximates a Poisson distribution with rate $\rho = (R \cdot T \cdot \kappa)/2^\lambda$; κ is tuned so that $\rho = 1/600$ (a block every 10 minutes).

B.2 Bitcoin from Our PoUW

A PoUW for blockchains (Definition 18), together with standard properties of the hash function, can be used to instantiate the same functionality as Bitcoin, i.e., a public ledger as defined in [18]. The proof follows [19], instantiating signatures of work with PoUW.

C Unpredictable Encodings

In Section 4.2 we introduced transcript-unpredictable encodings, which suffice for a PoUW for matrix multiplication. Here we abstract this into a more general primitive, an *unpredictable encoding* scheme (UES) – a form of "instance-optimal" randomized encoding with new and incomparable properties. A UES is associated with a function $f\colon \{0,1\}^n \to \{0,1\}^*$ with canonical algorithm Alg of worst-case running time $t(n)$, and consists of:

- $\mathsf{Encode}(x; r)$: a probabilistic procedure outputting an encoding $\tilde{x}$;
- $\mathsf{Eval}(\tilde{x})$: a deterministic procedure outputting y.

Correctness: $\mathsf{Eval}(\mathsf{Encode}(x)) = f(x)$ for every x. *Running time*: the worst-case running time of Encode plus that of Eval is $t(n) \cdot (1 + o(1))$. *Hardness*: there is a constant C such that for every x and every time-$t'(n)$ algorithm $\mathcal{A}$, $\Pr_r[\mathcal{A}(x, r) = \mathsf{Encode}(x; r)] \le C \cdot (t'(n)/t(n))$.

The transcript-unpredictability notion of Section 4.2 is the special case where $\tilde{x}$ is the transcript of the canonical matrix-multiplication algorithm applied to the encoded input matrices. The definition resembles, syntactically, randomized encodings [21, 3], but differs: we do *not* require privacy of x given $\tilde{x}$, and instead require a fine-grained hardness property together with an "optimal" efficiency property.

D Deferred Proof

▷ Claim 19. Let A be an $n \times d$ matrix ($d < n$) with entries chosen independently and uniformly from $\mathbb{F}_q$. Then $\Pr[\mathsf{rk}(A) = d] = 1 - O(q^{-(n-d+1)})$.

Proof. A has rank d iff its d columns are linearly independent. The k-th column is independent of the previous $k-1$ columns (spanning a $(k-1)$-dimensional subspace) with probability $1-q^{-(n-k+1)}$. Hence

$$\Pr[\mathsf{rk}(A)=d]=\prod_{k=0}^{d-1}\left(1-\tfrac{1}{q^{n-k}}\right)\geq\exp\left(-\sum_{k=0}^{d-1}\tfrac{1}{q^{n-k}}\right)\geq\exp\left(-2q^{-(n-d+1)}\right)\geq 1-2q^{-(n-d+1)},$$

using $\sum_{k=0}^{d-1}q^{-(n-k)}\leq 2q^{-(n-d+1)}$. ◁

Lemma 11 follows: conditioned on E_L, E_R being full rank (which holds except with the above probability), the row space of $E=E_LE_R$ equals that of E_L and is uniform over r-dimensional subspaces of $\mathbb{F}_q^n$, and likewise its column space is uniform via E_R; hence every matrix in $\mathcal{E}_{r,n}$ is equally likely.