Short Paper: The Proof is in the Pudding
Proofs of Work for Solving Discrete Logarithms

Marcella Hastings[1], Nadia Heninger[2], and Eric Wustrow[3]

[1] University of Pennsylvania
[2] University of California, San Diego
[3] University of Colorado Boulder

Abstract. We propose a proof of work protocol that computes the discrete logarithm of an element in a cyclic group. Individual provers generating proofs of work perform a distributed version of the Pollard rho algorithm. Such a protocol could capture the computational power expended to construct proof-of-work-based blockchains for a more useful purpose, as well as incentivize advances in hardware, software, or algorithms for an important cryptographic problem. We describe our proposed construction and elaborate on challenges and potential trade-offs that arise in designing a practical proof of work.

Keywords: Proofs of work, discrete log, Pollard rho

1 Introduction

We propose a proof of work scheme that is useful for cryptanalysis, in particular, solving discrete logarithms. The security of the ECDSA digital signature scheme is based on the hardness of the elliptic curve discrete log problem. Despite the problem's cryptographic importance, the open research community is small and has limited resources for the engineering and computation required to update cryptanalytic records; recent group sizes for elliptic curve discrete log records include 108 bits in 2002 [11], 112 bits in 2009 [10], and 113 bits in 2014 [30].

Our proposition aims to harness the gigawatts of energy spent on Bitcoin mining [29] to advance the state of the art in discrete log cryptanalysis. Jakobsson and Juels [16] call this a *bread pudding* proof of work. Just as stale bread becomes a delicious dessert, individual proofs of work combine to produce a useful computation. While memory-hard functions aim to discourage specialized hardware for cryptocurrency mining [21], we hope for the exact opposite effect. Just as Bitcoin has prompted significant engineering effort to develop efficient FPGAs and ASICs for SHA-256, we wish to use the lure of financial rewards from cryptocurrency mining to incentivize special-purpose hardware for cryptanalysis.

2 Background

Let G be a cyclic group with generator g of order q. We represent the group operation as multiplication, but every algorithm in our paper applies to a generic group. Every element $h \in G$ can be represented as an integer power of g, $g^a = h$, $0 \leq a < q$, and also has a unique representation as a sequence of bits. The discrete logarithm $log_g(h)$ is a, $0 \leq a < q$ satisfying $g^a = h$. Computing discrete logs is believed to be difficult for certain groups, including multiplicative groups modulo primes and elliptic curve groups. The conjectured hardness of discrete log underlies the security of multiple important cryptographic algorithms, including the Diffie-Hellman key exchange [12, 4] and the Digital Signature Algorithm [20]. Efficient computation of a discrete log for a group used for Diffie-Hellman key exchange would allow an adversary to compute the private key from the public key exchange messages; for DSA signatures, such an adversary could compute the private signing key from the public key and forge arbitrary signatures.

2.1 Discrete Log Cryptanalysis

There are two main families of algorithms for solving the discrete log problem. The first family works over any group, and includes Shanks's baby step giant step algorithm [24], and the Pollard rho and lambda algorithms [22]. These algorithms run in time $O(\sqrt{q})$ for any group of order q. It is this family of algorithms we target in this paper. A second family of algorithms is based on index calculus [15, 3]; these algorithms have sub-exponential running times only over finite fields.

Current best practices for elliptic curves are to use 256-bit curves [4], although 160-bit curves remain supported in some implementations [27]. Bitcoin miners currently perform around 2^{90} hashes per year and consume 0.33% of the world's electricity [29]. If this effort were instead focused on discrete log, a 180-bit curve could be broken in around a year[4]. Scaling this to discrete logs in 224-bit groups would require all current electricity production on Earth for 10,000 years. Alternative cryptocurrencies such as Litecoin, Ethereum, and Dogecoin achieve lower hash rates of about 2^{72} hashes per year[5].

2.2 Pollard Rho with Distinguished Points

The protocols we study in this paper compute the discrete log of an element h by finding a collision $g^a h^b = g^{a'} h^{b'}$ with $b \not\equiv b' \mod q$. Given such an equivalence, the discrete log of h can be computed as $(a' - a)/(b - b') \bmod q$.

Pollard's rho algorithm for discrete logarithms [22] works for any cyclic group G of order q. The main idea is to take a deterministic pseudorandom walk inside of the group until the same element is encountered twice along the walk. By the birthday bound, such an element will be found with high probability after $\Theta(\sqrt{q})$ steps. The non-parallelized version of this algorithm uses a cycle-finding algorithm to discover this collision, and computes the log as above.

We base our proof of work on Van Oorschot and Wiener's [28] parallelized Pollard rho algorithm using the method of distinguished points. A distinguished

[4] Elliptic curve point multiplications take about 2^{10} times longer than SHA-256 on modern CPUs.

[5] Extrapolated from peak daily hash rates at `bitinfocharts.com`

point is an element whose bitwise representation matches some easily-identifiable condition, such as having d leading zeros. Each individual process j independently chooses a random starting point $g^{a_j}h^{b_j}$ and generates a psuedorandom walk sequence from this starting element. When the walk reaches a distinguished point, the point is saved to a central repository and the process starts over again from a new random starting point until a collision is found.

The number of steps required to compute the discrete log is independent of d, which we call the difficulty parameter below; d only determines the storage required. We expect to find a collision after $\Theta(\sqrt{q})$ steps by all processes. With m processes running in parallel, the calendar running time is $O(\sqrt{q}/m)$.

The pseudorandom walk produces a deterministic sequence within the group from some starting value. Given a group generator g and a target h, the walk generates a random starting point $x_0 = g^{a_0}h^{b_0}$ by choosing random exponents a_0, b_0. In practice, most implementations use the Teske pseudorandom walk [26]: given a disjoint partition of G with 20 sets of equal size $T_1, \ldots, T_{20}$ parameterized by the bitwise representation of an element, choose $m_s, n_s \in [1, q]$ at random and define $M_s = g^{m_s}h^{n_s}$ for $s \in [1, 20]$. Then we can define the walk $\mathcal{W}(x) = M_s * x$ for $x \in T_s$. In general, an effective pseudorandom walk updates the group representation of a point based on some property of the bitwise representation.

2.3 Proofs of Work

A proof of work [13, 16] protocol allows a *prover* to demonstrate to a *verifier* that they have executed an amount of work. We use the definition from [2].

Definition 1. *A $(t(n), \delta(n))$-Proof of Work (PoW) consists of three algorithms* $(\mathsf{Gen}, \mathsf{Solve}, \mathsf{Verify})$ *that satisfy the following properties:*

- ***Efficiency**:*
  - $\mathsf{Gen}(1^n)$ *runs in time* $\tilde{O}(n)$.
  - *For any* $c \leftarrow \mathsf{Gen}(1^n)$, $\mathsf{Solve}(c)$ *runs in time* $\tilde{O}(t(n))$.
  - *For any* $c \leftarrow \mathsf{Gen}(1^n)$ *and any* π, $\mathsf{Verify}(c, \pi)$ *runs in time* $\tilde{O}(n)$.
- ***Completeness**: For any* $c \leftarrow \mathsf{Gen}(1^n)$ *and any* $\pi \leftarrow \mathsf{Solve}(c)$, $\Pr[\mathsf{Verify}(c, \pi) = \mathit{accept}] = 1$.
- ***Hardness**: For any polynomial* ℓ, *any constant* $\epsilon > 0$, *and any algorithm* Solve^*_ℓ *that runs in time* $\ell(n)t(n)^{1-\epsilon}$ *when given as input* $\ell(n)$ *challenges* $\{c_i \leftarrow \mathsf{Gen}(1^n)\}_{i \in [\ell(n)]}$,
  $\Pr\left[\forall i\, \mathsf{Verify}(c_i, \pi_i) = \mathit{accept} \mid (\pi_1, \ldots, \pi_{\ell(n)}) \leftarrow \mathsf{Solve}^*_\ell(c_1, \ldots, c_{\ell(n)})\right] < \delta(n)$

We can describe the hash puzzle proof of work [1] used by Bitcoin [19] in this framework as follows. The challenge generated by Gen is the hash of the previous block. Solve is parameterized by a difficulty d; individual miners search for a nonce n such that SHA-256$(c, n) \leq 2^{256-d}$ when mapped to an integer. Assuming that SHA-256 acts like a random function, miners must brute force search random values of n; the probability that a random fixed-length integer is below the difficulty threshold is 2^{-d}, so the conjectured running time for Solve is $t(n) = O(2^d)$. Verify runs in constant time and accepts if SHA-256$(c, n) \leq 2^{256-d}$.

Proposals for "useful" proofs of work. Primecoin [17] proofs contain prime chains, which may be of scientific interest. DDoSCoin [31] proofs can cause a denial of service attack. TorPath [6] increases bandwidth on the Tor network. Ball et al. [2] describe theoretical proof-of-work schemes based on worst-case hardness assumptions from computational complexity theory. Lochter [18] independently outlines a similar discrete log proof of work.

3 Proof of work for discrete log

The goal of this thought experiment is to develop a proof of work scheme that, if provided with mining power at Bitcoin's annual hash rate, can solve a discrete log in a 160-bit group. We outline our proposed scheme, explain limitations of the simple model, and describe possible avenues to fix the gap.

3.1 Strawman Pollard rho proof of work proposal

In our rho-inspired proof of work scheme, workers compute a pseudorandom walk from a starting point partially determined by the input challenge and produce a distinguished point. The parameters defining the group G, group generator g, discrete log target h, and deterministic pseudorandom walk function $\mathcal{W}$, are global for all workers and chosen prior to setup. A distinguished point x at difficulty d is defined as having d leading zeros in the bitwise representation, where d is a difficulty parameter provided by the challenge generator.

In the terminology of Definition 1, Gen produces a challenge bit string c; when used in a blockchain, c can be the hash of the previous block.

To execute the Solve function, miners generate a starting point for their walk, for example by generating a pair of integers $(a_0, b_0) = H(c||n)$ where n is a nonce chosen by a miner and H is a cryptographically secure hash function, and computing the starting point $P_0 = g^{a_0}h^{b_0}$. Workers then iteratively compute $P_i = \mathcal{W}(P_{i-1})$ until they encounter a distinguished point $P_D = g^{a_D}h^{b_D}$ of difficulty d, and output $\pi = (n, a_D, b_D, P_D)$. A single prover expects to take $O(2^d)$ steps before a distinguished point is encountered.

The Verify function can check that $P_D = g^{a_D}h^{b_D}$ and has d leading zeros. This confirms that P_D is distinguished, but does not verify that P_D lies on the random walk of length ℓ starting at the point determined by (a_0, b_0). Without this check, a miner can pre-mine a distinguished point and lie about its relationship to the starting point. A verifier can prevent this by verifying every step of the random walk, but this does not satisfy the efficiency constraints of Definition 1.

A discrete log in a group of order q takes $\sqrt{q}$ steps to compute (see Section 2.2). A set of m honest miners working in parallel expect to perform $O(2^d)$ work per proof. If all miners have equal computational power, the winning miner will find a distinguished point after expected $O(2^d/m)$ individual work. This construction expects to store $\sqrt{q}m/2^d$ distinguished points in a block chain before a collision is found; the total amount of work performed by all miners for all blocks to compute the discrete log is $\sqrt{q}m$. Each distinguished point wastes $(m-1)/m$ work performed by miners who do not find the "winning" point.

We next examine several modified proof-of-work schemes based on this idea that attempt to solve the problems of verification and wasted work.

3.2 Reducing the cost of wasted work

To reduce wasted work, we can allow miners that do not achieve the first block to announce their blocks and receive a partial block reward. One technique is to use the Greedy Heaviest-Observed Sub-Tree method [25] to determine consensus, which has been adopted by Ethereum in the form of Uncle block rewards [14]. In this consensus method, the main (heaviest) chain is defined as the sub-tree of blocks containing the most work, rather than the longest chain. This allows stale blocks to contribute to the security of a single chain, and allocates rewards to their producers. In Ethereum, this supports faster block times and lowers orphan rates but we could use it to incentivize miners to publish their useful work rather than discard it when each new block is found.

3.3 Limiting the length of the pseudorandom walk

We attempt to reduce the cost of the Verify function by limiting the length of the random walk in a proof to at most 2^{ℓ} steps for some integer ℓ. Individual miners derive a starting point from the challenge c and a random nonce n. They walk until they either find a distinguished point or pass 2^{ℓ} steps. In the latter case, the miner chooses another random nonce n and restarts the walk.

Solve requires miners to produce a proof $\pi = (n, \mathcal{L}, a_D, b_D)$ satisfying four criteria: (1) the walk begins at the point derived from a hash of the challenge and nonce values ($(a_0, b_0) = H(c||n)$), (2) walking from this initial point for $\mathcal{L}$ steps leads to the specified endpoint ($\mathcal{W}^{\mathcal{L}}(g^{a_0}h^{b_0}) = g^{a_D}h^{b_D}$), (3) the bitwise representation of the endpoint $g^{a_D}h^{b_D}$ is distinguished and (4) the walk does not exceed the maximum walk length ($\mathcal{L} < 2^{\ell}$). Solve runs in expected time $O(2^d)$.

Verify retraces the short walk and runs in $O(2^{\ell})$ steps. Overall, fixing a maximum walk length forces more total work to be done, since walks over 2^{ℓ} steps are never published. The probability that a length 2^{ℓ} random walk contains a distinguished point of difficulty d is $2^{\ell-d}$, so a prover expects to perform $2^{d-\ell}$ random walks before finding a distinguished point. An individual prover in a group of order q can expect to store $O(\sqrt{q}/2^{\ell})$ distinguished points before a collision is found. With 2^d work performed per distinguished point stored, the total amount of work is $O(2^{d-\ell}\sqrt{q})$. For $m \ll 2^{d-\ell}$ miners working in parallel, the work wasted by parallel mining is subsumed by that of discarded long walks.

To target a 160-bit group with mining power of around 2^{90} hashes per year, the total amount of work performed by miners should not exceed $2^{90} \geq 2^{d-\ell}2^{80}$, or $10 \geq d-\ell$, with a total of $2^{80-\ell}$ distinguished points. If we allow 1 GB $= 8 \cdot 10^9$ storage, this allows up to 2^{25} 160-bit distinguished points, so we have $\ell = 55$, and thus we set the difficulty $d = 65$. This is feasible: as of Sep 2018, Bitcoin miners produce nearly 2^{75} hashes per block and the blockchain is $\sim$183 GB.

3.4 Efficiently verifying pseudorandom walks

In theory, a SNARK [7] solves the efficient verification problem for the proof of work. Provers compute the SNARK alongside the pseudorandom walk, and include the result with the proof of work. Verification executes in constant time. Unfortunately, generating a SNARK is thousands of times more expensive than performing the original computation. A STARK [5] takes much less work to solve but slightly longer to verify and comes with a non-negligible space trade-off. In our framework, Solve finds a distinguished point and build a STARK: this takes time $O(d^2 2^{2d})$ and space $O(d2^d)$ group elements. Verify executes the STARK verify function in time $O(d)$. Verifiable delay functions [9] could also be used to solve this problem, but existing solutions appear to take advantage of algebraic structure that we do not have in our pseudorandom walk.

We attempted to emulate a verifiable delay function by defining an alternate pseudorandom walk. We experimented with several possibilities, for example a "rotating" walk that performs a set of multiplications and exponentiations in sequence. A walk of this type has the convenient algebraic property that it is simple to verify for a given start point, end point, and length $\mathcal{L}$, that the end point is $\mathcal{L}$ steps from the start. Unfortunately, it has terrible pseudorandom properties: collisions are either trivial or occur after $O(q)$ steps. There appears to be a tension between the pseudorandomness properties required for the Pollard rho algorithm to achieve $O(\sqrt{q})$ running time and an algebraic structure allowing efficient verification of the walk. Effective random walks determine each step by the bitwise representation of a given element—independent of its group element representation $g^{a_i}h^{b_i}$—but this independence makes it difficult to reconstruct or efficiently summarize the group steps without repeating the entire computation. We leave the discovery of such a pseudorandom walk to future work.

3.5 Distributed verification

An alternate block chain formulation has miners accept blocks unless they see a proof that it is invalid, and incentivizes other validators to produce such proofs. This technique has been proposed for verifying off-chain transactions in Ethereum Plasma [23]. We extend this idea to allow validators to prove a miner has submitted an invalid block and offer rewards for such discoveries.

In this scheme, the Verify function accompanies a *reject* decision with a proof of falsification f, and can take as long as mining: $\tilde{O}(t(n))$. We define a function $\mathsf{Check}(c, f)$ to check whether this proof of falsification is accurate, which runs in time $\tilde{O}(n)$. In a block chain, miners Solve proofs of work and dedicated verifiers Verify. If a verifier produces a proof of falsification f (that is, finds an invalid block) it broadcasts (c, f) to all participants, who must Check the falsification.

To increase verification cost, there must be a matching increase in incentive. For example, a time-delayed bounty system requires a miner to provide a bounty with each new block, which is either collected by the miner with the block reward after a fixed amount of time, or partially poached by a verifier who produces a valid falsification. Such a scheme aims to prevent collusion between miners and verifiers to collect rewards and bounty for no useful work.

Walk summaries. A first idea modifies the proof of work π to include intermediate points $s_0, s_1, \ldots$ spaced at regular intervals along the walk. The Verify function picks a random subset of the s_i and retraces the walks from s_i to s_{i+1}. An invalid proof has the property that at least one interval does not have a valid path between the endpoints. For a walk with I intervals of length ℓ, a verifier that checks k intervals has probability k/I of detecting an invalid proof with work kI. However, checking a claimed falsification f requires ℓ work. A malicious verifier can report incorrect falsifications and force other participants to perform arbitrary work. To fix this, we need more efficiently checkable falsifications.

Bloom filters for secondary validation. One approach to efficiently checkable proof falsifications uses Bloom filters [8], a probabilistic data structure that tests set membership. It may return false positives, but never false negatives. We modify our walk summary proof of work π above to also include a Bloom filter containing every point on the walk. The Verify function chooses a random interval s_i and takes ℓ walk steps, which takes work ℓ. If an element e_i on the walk is absent from the filter, the verifier broadcasts the sequence of points $f = (e_{i-k}, \ldots, e_i)$. The Check function confirms that the points f are a correctly generated random walk and that all points except e_i are contained in the Bloom filter. This takes time k. The short sequence prevents a malicious verifier from invalidating a correct block by taking advantage of false positives in Bloom filters.

A Bloom filter containing every element in a random walk for a reasonable difficulty value will be too large (we estimate at least 150 TB for a walk of length 2^{60}). To shrink the filter, we could store hashes of short sub-walks of length ℓ', rather than every step. To Check, a participant must walk ℓ' steps for each of the k broadcast sub-walks. This increases the work to $k\ell'$, but decreases Bloom filter size by a factor of ℓ'.

Acknowledgement. Joseph Bonneau, Brett Hemenway, Michael Rudow, Terry Sun, and Luke Valenta contributed to early versions of this work. Nadia Heninger carried out this research while at the University of Pennsylvania. This work was supported by the National Science Foundation under grants no. CNS-1651344 and CNS-1513671 and by the Office of Naval Research under grant no. 568751.

References

1. Back, A.: Hashcash-a denial of service counter-measure (2002)
2. Ball, M., Rosen, A., Sabin, M., Vasudevan, P.N.: Proofs of work from worst-case assumptions. In: CRYPTO 2018. Springer International Publishing (2018)
3. Barbulescu, R., Gaudry, P., Joux, A., Thomé, E.: A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In: EUROCRYPT'14 (2014)
4. Barker, E., Chen, L., Roginsky, A., Vassilev, A., Davis, R.: SP 800-56A Revision 3. Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography. National Institute of Standards & Technology (2018)

5. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive (2018)
6. Biryukov, A., Pustogarov, I.: Proof-of-work as anonymous micropayment: Rewarding a Tor relay. In: FC'15. Springer (2015)
7. Bitansky, N., Canetti, R., Chiesa, A., Goldwasser, S., Lin, H., Rubinstein, A., Tromer, E.: The hunting of the SNARK. Journal of Cryptology **30**(4) (2017)
8. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (Jul 1970). https://doi.org/10.1145/362686.362692
9. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Annual International Cryptology Conference. pp. 757–788. Springer (2018)
10. Bos, J.W., Kaihara, M.E., Kleinjung, T., Lenstra, A.K., Montgomery, P.L.: Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. International Journal of Applied Cryptography **2**(3) (2012)
11. Certicom ECC challenge (1997), `http://certicom.com/images/pdfs/challenge-2009.pdf`, Updated 10 Nov 2009. Accessed via Web Archive
12. Diffie, W., Hellman, M.: New directions in cryptography. IEEE transactions on Information Theory **22**(6), 644–654 (1976)
13. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Annual International Cryptology Conference. pp. 139–147. Springer (1992)
14. Ethereum Project: Ethereum white paper, `https://github.com/ethereum/wiki/wiki/White-Paper\#modified-ghost-implementation`
15. Gordon, D.M.: Discrete logarithms in GF(P) using the number field sieve. SIAM J. Discret. Math. **6**(1), 124–138 (Feb 1993). https://doi.org/10.1137/0406010
16. Jakobsson, M., Juels, A.: Proofs of work and bread pudding protocols. In: Secure Information Networks, pp. 258–272. Springer (1999)
17. King, S.: Primecoin: Cryptocurrency with prime number proof-of-work (2013)
18. Lochter, M.: Blockchain as cryptanalytic tool. Cryptology ePrint Archive, Report 2018/893 (2018), `https://eprint.iacr.org/2018/893.pdf`
19. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. White paper (2008)
20. National Institute of Standards and Technology: FIPS PUB 186-4: Digital Signature Standard (DSS). National Institute of Standards and Technology (Jul 2013)
21. Percival, C., Josefsson, S.: The scrypt password-based key derivation function. RFC 7914, RFC Editor (Aug 2016), `http://rfc-editor.org/rfc/rfc7914.txt`
22. Pollard, J.M.: Monte carlo methods for index computation (mod p). In: Mathematics of Computation. vol. 32 (1978)
23. Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts (2017)
24. Shanks, D.: Class number, a theory of factorization, and genera. In: Proc. of Symp. Math. Soc., 1971. vol. 20, pp. 41–440 (1971)
25. Sompolinsky, Y., Zohar, A.: Secure high-rate transaction processing in Bitcoin. In: FC'15. pp. 507–527. Springer (2015)
26. Teske, E.: Speeding up Pollard's rho method for computing discrete logarithms. In: ANTS-III. pp. 541–554. Springer-Verlag, Berlin, Heidelberg (1998)
27. Valenta, L., Sullivan, N., Sanso, A., Heninger, N.: In search of CurveSwap: Measuring elliptic curve implementations in the wild. In: EuroS&P. IEEE (2018)
28. Van Oorschot, P.C., Wiener, M.J.: Parallel collision search with cryptanalytic applications. Journal of cryptology **12**(1), 1–28 (1999)
29. de Vries, A.: Bitcoin's growing energy problem. Joule **2**(5), 801–805 (2018)
30. Wenger, E., Wolfger, P.: Harder, better, faster, stronger: elliptic curve discrete logarithm computations on FPGAs. Journal of Cryptographic Engineering (2016)
31. Wustrow, E., VanderSloot, B.: DDoSCoin: Cryptocurrency with a malicious proof-of-work. In: WOOT (2016)