

RESEARCH ARTICLE

Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem

Alex Biryukov,* Dmitry Khovratovich†

Abstract. Proof-of-work is a central concept in modern cryptocurrencies and denial-of-service protection tools, but the requirement for fast verification so far has made it an easy prey for GPU-, ASIC-, and botnet-equipped users. The attempts to rely on memory-intensive computations in order to remedy the disparity between architectures have resulted in slow or broken schemes. In this paper we solve this open problem and show how to construct an *asymmetric* proof-of-work (PoW) based on a computationally-hard problem, which requires a great deal of memory to generate a proof (called a "memory-hardness" feature) but is instant to verify. Our primary proposal, Equihash, is a PoW based on the generalized birthday problem and enhanced Wagner's algorithm for it. We introduce the new technique of *algorithm binding* to prevent cost amortization and demonstrate that possible parallel implementations are constrained by memory bandwidth. Our scheme has tunable and steep time-space trade-offs, which impose large computational penalties if less memory is used. Our solution is practical and ready to deploy: a reference implementation of a proof-of-work requiring 700 MB of RAM runs in 15 seconds on a 2.1 GHz CPU, increases the computations by a factor of 1000 if memory is halved, and presents a proof of just 120 bytes long.

1. Introduction

Request of intensive computations as a countermeasure against spam was first proposed by Dwork and Naor in 1992,³³ and as denial of service (DoS) protection in the form of TLS client puzzle by Dean and Stubblefield in 2001.²⁹ The amount of work is certified by a proof, thus called *proof-of-work*, which is feasible to get by an ordinary user, but at the same time slows down multiple requests from the single machine or a botnet. Perhaps the simplest scheme is Hashcash,¹⁶ which requires a hash function output to have certain number of leading zeros and is adapted within the Bitcoin cryptocurrency. At present, to earn 25 Bitcoins a miner must make an average of 2^{68} calls to a cryptographic hash function.

Long before the rise of Bitcoin it was realized that dedicated hardware can produce a proof-of-work much faster and cheaper than a regular desktop or laptop.³² Thus the users equipped with such hardware have an advantage over others, which has led the Bitcoin mining industry to concentrate in a few hardware farms of enormous size and high electricity consumption.

*A. Biryukov (alex.biryukov@uni.lu) is a full professor of Cryptography and Information Security in the Computer Science and Communications Research Unit at the University of Luxembourg:

16B1873NxbzzX2xNhW2GzYbCBjtjxzF23j

†D. Khovratovich is a post-doctoral researcher at the CryptoLUX cryptography research group at the University of Luxembourg:

1PbreZUDQjpVWFLBSZp3aimaqSyoilLa5eT

An advantage of the same order of magnitude is given to “owners” of large botnets, which at present often accommodate hundreds of thousands of machines. For practical DoS protection, this means that the early TLS puzzle schemes are no longer effective against the most powerful adversaries.^{16, 28}

1.1. Memory-hard computing—In order to remedy the disparity between the ASICs and regular CPUs, Dwork *et al.* first suggested memory-bound computations,^{11,32} where a random array of moderate size is accessed in pseudo-random manner to get high bandwidth. In later work they suggested filling this memory with a memory-hard function (though this term was not used) so that the memory amount can be reduced only at the large computational cost to the user.^{34, 1} As memory is a very expensive resource in terms of area and the amortized chip cost, ASICs would be only slightly more efficient than regular x86-based machines. Botnets remain a problem, though on some infected machines the use of GBytes of RAM will be noticeable to users. One can also argue that the reduced ASIC advantages may provide additional incentives for botnet creators and thus reduce the security of an average Internet user.^{42, 26}

No scheme in either the 2003 or 2005 work by Dwork *et al.* has been adapted for the practical use as a PoW.^{32, 34} Firstly, they are too slow for reasonably large amount of memory, and must use too little memory if required to run in reasonable time (*i.e.*, seconds). The first memory-hard candidates were based on superconcentrators, and similar constructions explored in the theory of pebbling games on graphs.^{34, 56, 40} To fill N blocks in memory a superconcentrator-based functions make $N \log N$ hash function calls, essentially hashing the entire memory dozens of times. Better performance is achieved by the scrypt function and memory-hard constructions among the finalists of the Password Hashing Competition, but their time-space tradeoffs have been explored only recently.^{55, 10, 24, 12, 13}

For example, a superconcentrator-based function using $N = 2^{25}$ vertices of 32 bytes each (thus taking over 1 GB of RAM) makes $O(N \log N)$ or cN with large c calls to the hash function (the best explicit construction mentioned in Dziembowski *et al.* makes $44N$ calls,³⁵ thus hashing the entire memory dozens of time.

Secondly, the proposed schemes (including the PHC constructions) are *symmetric* with respect to the memory use. To initialize the protocol in Dwork *et al.*,^{32, 34} a verifier must use the same amount of memory as the prover. This is in contrast with the Bitcoin proof-of-work: whereas it can be checked instantly (thus computational asymmetry) without precomputation, virtually no memory-hard scheme offers *memory asymmetry*. Thus the verifier has to be almost as powerful as the prover, and may cause DoS attacks by itself. In the cryptocurrency setting, fast verification is crucial for network connectivity. Even one of the fastest memory-hard constructions, scrypt,⁵⁵ had to be taken with memory parameter of 128 KB to ensure fast verification in Litecoin. As a result, Litecoin is now mined on ASICs with 100x efficiency gain over CPU.⁴⁸

Finally, these schemes have not been thoroughly analyzed for possible optimizations and amortizations. To prove the work, the schemes should not allow any optimization (which adversaries would be motivated to conceal) nor should the computational cost be amortizable over multiple calls.³³

A reasonably fast and memory-asymmetric scheme would become a universal tool and could be used as an efficient DoS countermeasure, spam protection, or a core for a new egalitarian cryptocurrency.

1.2. *Recent asymmetric candidates*—There have been two notable attempts to solve the symmetry and performance problems. The first one by Dziembowski *et al.* suggests an interactive protocol, called a *proof-of-space*, where the prover first computes a memory-hard function and then a verifier requests a subset of memory locations to check whether they have been filled by a proper function.³⁵ The verification can thus be rather quick. However, the memory-hard core of the scheme is based on a stack of superconcentrators and is quite slow: to fill 1 GB of memory it needs about 1 minute according to the performance reports in Park *et al.*⁵⁴ The scheme in Dziembowski *et al.* is not amortization-free: producing N proofs costs as much as producing one.³⁵ As a result, a memory-hard cryptocurrency built on proofs-of-space, Spacecoin, requires miners to precommit the space well before the mining process, thus making the mining process centralized.⁵⁴ We also note that the time-space tradeoff is explored for these constructions for memory reductions by a logarithmic factor (*e.g.* 30 for 1 GB) and more, whereas the time increases for smaller reductions are unknown.

A more promising scheme was proposed by Tromp as the *Cuckoo-cycle* PoW.⁶¹ The prover must find a cycle of certain length in a directed bipartite graph with N vertices and $O(N)$ edges. It is reasonably efficient (only 10 seconds to fill 1 GB of RAM with 4 threads) and allows very fast verification. The author claimed prohibitive time-memory tradeoffs. However, the original scheme was broken by Andersen: a prover can reduce the memory by the factor of 50 with time increase by the factor of 2 only.¹⁴ Moreover, Andersen demonstrated a simple time-memory tradeoff, which allows for the constant time-memory product (reduced by the factor of 25 compared to the original). Thus the actual performance is closer to 3-4 minutes per GB.² Apart from Andersen's analysis, no other tradeoffs were explored for the problem in Tromp, there is no evidence that the proposed cycle-finding algorithm is optimal, and its amortization properties are unknown.⁶¹ Finally, Andersen's tradeoff allows one to parallelize the computations independently, thus reducing the time-memory product and the costs on dedicated hardware.

Finally, the scheme called Momentum simply looks for a collision in 50-bit outputs of the hash function with 26-bit input.⁴⁹ The designer did not explore any time-space tradeoffs, but apparently they are quite favourable to the attacker: reducing the memory by the factor of q imposes only a \sqrt{q} penalty on the running time (more details in Appendix 1).⁶²

1.3. *Our contributions*—We propose a family of fast, memory-asymmetric, optimization/amortization-free, limited parallelism proofs of work based on hard and well-studied computational problems. First we show that a wide range of hard problems (including a variety of NP-complete problems) can be adapted as an asymmetric proof-of-work with tunable parameters, where the ASIC and botnet protection are determined by the time-space tradeoffs of the best algorithms.

Our primary proposal, Equihash, is a PoW based on the generalized birthday problem, which has been explored in a number of papers from both theoretical and implementation points of view.^{63, 22, 51, 23, 44} To make it amortization-free, we develop the technique called **algorithm binding** by exploiting the fact that Wagner's algorithm carries its footprint on a solution.

In our scheme a user can independently tune time, memory, and time-memory tradeoff parameters. In a concrete setting, our 700 MB-proof is 120 bytes long and can be found in 15 seconds on a single-thread laptop with 2.1 GHz CPU. An adversary trying to use 250 MB of memory would pay 1000-fold in computations using the best tradeoff strategy, whereas a memoryless algorithm would require prohibitive 2^{75} hash function calls. These properties and performance are unachievable by existing proposals. We have implemented and tested our scheme

in several settings, with the code available at request. Equihash can be immediately plugged into a cryptocurrency or used as a TLS client puzzle.

To increase confidence in our proposal, we review and improve the best existing tradeoff strategies for the generalized birthday problem.

We also show how to build a PoW from two different hard problems generically and get the best of their tradeoffs when the algorithm binding technique is not applicable. In this context we explore the hard knapsack problem, for which the best algorithms have been scrutinized in the recent papers.^{41, 18, 31}

1.4. Outline—This paper is structured as follows: first, we review the properties required from an asymmetric proof-of-work and show how to adapt a computationally-hard problem for a PoW (Section 2); we review the generalized birthday problem and Wagner’s algorithm in Section 3 and outline our primary proposal in Section 4; and the new results on the time-space tradeoffs and parallelism are proven in Sections 5 and 6. Generic problem composition is left for Appendices.

2. Proofs-of-Work and Hard Computational Problems

In this section we list the properties that we require from a proof-of-work and explain in a generic way how to turn a hard computational problem into a proof-of-work, as well as what the necessary conditions are for such a problem. A reader interested in a concrete proposal with technical details may immediately proceed to Section 4.

2.1. Properties—We define a *problem*

$$\mathcal{P} : \mathcal{R} \times \mathcal{I} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}.$$

as a hard predicate, where \mathcal{R} is the set of parameters that determine the hardness, \mathcal{I} is the set of inputs conforming to \mathcal{R} , and \mathcal{S} is the set of possible solutions. We assume that there is an algorithm (or a family of algorithms) \mathcal{A}_R that solves \mathcal{P}_R on any I , *i.e.* finds S such that $\mathcal{P}(R, I, S) = \text{true}$.

A *proof-of-work scheme* based on \mathcal{P} (and implicitly on the algorithm \mathcal{A} for it) is an interactive protocol, which operates as follows:

- (1) The Verifier sends a challenge input $I \in \mathcal{I}$ and parameters $R \in \mathcal{R}$ to the Prover.
- (2) The Prover finds solution S such that $\mathcal{P}(R, I, S) = \text{true}$ and sends it to the Verifier.
- (3) The Verifier computes $\mathcal{P}(R, I, S)$.

A non-interactive version (*e.g.*, for cryptocurrencies) can be derived easily. In this setting I contains some public value (last block hash in Bitcoin) and the prover’s ID. The prover publishes S so that every party can verify the proof.

Informally, \mathcal{A} should be moderately hard to impose significant costs on the prover. We also want all the provers, equipped with sufficient memory, to be in equal position so that no secret optimization or amortization can be applied. We summarize these requirements to \mathcal{P} and \mathcal{A} , and the other properties it must satisfy in order to become ASIC- and botnet-resistant, in the following list (*cf.* Dwork *et al.* 1992; Jakobsson and Jules).^{33, 43}

2.1.1. Progress-free process—In Bitcoin-like cryptocurrencies the mining process is usually a race among the miners to see who finds the proof first. To avoid centralization and mitigate the network delays, the mining must be a stochastic process, where the probability of the proof generation at any given time is non-zero and independent of the previous events. Therefore,

the mining must resemble the Poisson process, with the number of proofs found in a given timeframe following the Poisson distribution and the running time of the algorithm \mathcal{A}_R following the exponential distribution:

$$T(\mathcal{A}_R) \sim \text{Exponential}(\lambda(R)).$$

The Poisson process is often emulated by a difficulty filter: a fixed-time algorithm \mathcal{B} , which additionally takes some nonce N as input, is concatenated with a hash function G , whose output should have a certain number of trailing zeros. In this case, the algorithm \mathcal{B} must also be *amortization-free*, i.e. producing q outputs for \mathcal{B} should be q times as expensive.

A scheme that requires noticeable initialization time is not truly progress-free, although after the initialization the mining could become Poisson again such as in Park *et al.*⁵⁴ An informal statement could be that the shorter the initialization is, the more decentralized the mining will be.

2.1.2. Large AT cost—We expect that the ASIC or FPGA implementation of algorithms with large area requirements and high area-time product (AT) would not be much better than desktop implementations by the Price-Performance parameter. The area is maximized by the memory requirements. Therefore, for a regular user, the optimal implementation of \mathcal{A}_R should require sufficiently large memory M . Most desktops and laptops can handle 1 GB of RAM easily, whereas 1 GB of memory on chip is expensive.³

2.1.3. Small proof size and instant verification—The solution must be short enough and verified quickly using little memory in order to prevent DoS attacks on the verifier. We assume that some verifiers may use lightweight hardware such as smartphones with limited RAM and network bandwidth.

This requirement effectively cuts off straightforward use of memory-hard functions such as scrypt or the faster Argon2.²⁵ Even though a prover can be required to make exponentially more calls to these functions than the verifier, the latter still has to make at least one call and use a great deal of memory, which would motivate denial-of-service attacks on verifiers.

2.1.4. Steep time-space tradeoffs—Memory requirements are worthwhile as long as the memory reduction disproportionately penalizes the user. Many memory-intensive algorithms can run with reduced memory. Suppose that \mathcal{A}_R is a family of algorithms using different amounts of memory. Then we can think of it as a single algorithm taking the available memory amount as a parameter.

Let $T_R(M)$ be the average running time of \mathcal{A}_R with memory M . We also fix some standard implementation and its default memory requirements M_0 . Let us consider the running time growth when only M_0/q memory is used, $q > 1$:

$$C_R(q) = \frac{T_R(M_0/q)}{T_R(M_0)}.$$

We say that the time-space tradeoff for \mathcal{A}_R is *polynomial with steepness s* if $C_R(q)$ can be approximated by a polynomial of q of degree s . We say that the tradeoff is *exponential* if $C_R(q)$ is exponential of q . We note that polynomial steepness with $s = 1$ (which we call *linear*) implies that the memory can be equally traded for time. This keeps the AT cost constant, but reduces the design and production cost of a potential chip. Thus higher steepness is desirable. Finding time-space tradeoffs for most hard problems is a non-trivial task, as the best algorithms are usually optimized for computational complexity rather than for space requirements.^{17, 36}

2.1.5. Flexibility—To account for further algorithm improvements and architecture changes, the time, memory, and steepness of the PoW must be tunable independently. For cryptocurrencies this helps to sustain a constant mining rate. We recommend the following procedure (Fig. 1). To adjust M , we change R and get a new complexity (T', M') . To increase T by the factor 2^d , we harden \mathcal{P} in the style of the Bitcoin difficulty filter: $H(S)$ must have d leading zero bits, where H is a cryptographic hash function.

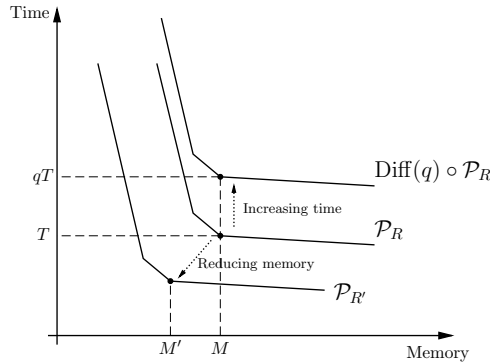


Fig. 1. Time and memory requirements for proof-of-work. The problem with parameters R can be solved in time T and memory M . In order to change M to M' , we replace R with R' . To increase time by the factor of q , we add the difficulty filter q in addition to R .

2.1.6. Parallelism-constrained—When a large portion of an ASIC is occupied by memory, adding a few extra cores does not increase the area. If \mathcal{A} can be parallelized, then the total time may be reduced and thus the AT cost can be lowered. However, if these cores have to communicate with each other, then the parallelism is limited by the network bandwidth. Thus if a specific PoW allows parallelism, the parallel version of algorithm \mathcal{A}_R should be *bandwidth-hard*, *i.e.* it quickly encounters the bottleneck in the network or RAM bandwidth.

2.1.7. Optimization-free—To avoid a clever prover getting advantage over the others, \mathcal{A} must be the most efficient algorithm to date, already employing all possible optimizations and heuristics, and it should be hard to find better algorithms.

We can now identify the problems with the Cuckoo-cycle PoW.⁶¹ It can be parallelized, which lowers the AT cost; dramatic optimizations have been identified;¹⁴ its time-space tradeoff has steepness 1; and finally, it has not been explored for amortization.

2.2. Memory-hard proof-of-work based on a hard problem—Several hard computational problems suitable for a proof-of-work (PoW) were studied by Dwork and Naor and later by Back.^{33, 16} These were PoWs with computational shortcuts: a verifier spends much less time than the prover. One could hope for memory shortcuts as well, *i.e.* verification requiring much less memory than the generation. However, a memory-hard PoW with a memory shortcut has been an open problem for quite a long time, as the existing proposals implicitly require both the verifier and the prover to allocate the same amount of memory.

Nevertheless, almost any hard problem can be turned into a proof-of-work in the framework outlined in Section 2.1. Any reader with an expertise in a particular problem is thus encouraged to create his own PoW. The well-known NP(-complete) problems (including their search and optimization analogs) are the most natural candidates, since the best algorithms for them run in exponential time, *i.e.* $\log T = O(|I|)$. On the other hand, the verification is polynomial in

$|I|$, so it is polylogarithmic in T . Thus the verification for NP-complete-based PoW should be fast compared to the running time. Moreover, the best algorithms for NP-complete problems usually require a non-negligible amount of memory and exhibit non-trivial time-space tradeoffs.³⁶ SAT solving, cliques, and Hamiltonian paths are all candidates for a PoW. The problems not demonstrated to be NP-complete but with best algorithms running in exponential time like factoring and discrete logs are natural proposals as well.

2.3. Inevitable parallelism—It is interesting to explore whether parallelism can be completely avoided; in other words if there exists a hard problem-based PoW that is *inherently sequential*.

We want for the PoW algorithm, which runs in time T and memory $M \approx T$ on a single processor (thus implying the time-area cost of $O(T^2)$), not to be significantly sped up using $O(T)$ processors. We believe that this is impossible in the following PoW-building paradigms.

A first approach could be to formulate the problem entirely for the short original input without explicit expansion. However, as we require fast verification and short input, *i.e.* $|I| = O(\log T)$, the problem we solve is implicitly in NP. Therefore, it can be solved in $\log T$ time using T processors (which in essence try all possible solutions).

A second approach (which we undertake in Equihash) is to expand the input from I to \hat{I} (so that $|\hat{I}| = \log |I|$) using some PRNG (*e.g.*, a hash function in the counter mode as we do) and apply a problem from P to \hat{I} . Here we also have an obstacle: many problems from P can be solved in polylogarithmic time using a polynomial number of processors, *i.e.* they actually belong to the class NC. It is conjectured though that $\text{NC} \subsetneq \text{P}$, which would imply the existence of P-problems outside of NC. If they exist, they belong to the class of P-complete problems, which reduce to each other with a polylogarithmic number of parallel processors. Unfortunately, the known P-complete problems and those assumed to be inherently sequential (such as the GCD problem) are not known to be verifiable in logarithmic time, even if we somehow manage to generate their inputs from logarithmically shorter ones.

To conclude, there is little foundation to prevent parallelism in a hard problem-based PoW, so we cope with it in a different manner – by showing that any parallel solution would enlarge the chip prohibitively or require enormous memory bandwidth.

2.4. Choosing a hard problem for PoW—It turns out that the properties that we listed in Section 2.1 are hard to satisfy simultaneously. A great difficulty lies in the optimization-free requirement, as the complexity of the most algorithms for hard problems is not evaluated with sufficient precision. Many algorithms are inherently amortizable. The existing implementations contain a number of heuristics. We conclude that the problem and the best algorithm must be very simple. So far we have identified three problems, for which the best algorithms are explored, scrutinized, and implemented:

- The generalized-birthday, or k -XOR problem, which looks for a set of n -bit strings that XOR to zero. The best existing algorithm is due to Wagner with minor modifications in Minder and Sinclair.^{63, 51} The algorithm's time-space tradeoffs were explored in Bernstein (2007) and it was implemented in Bernstein *et al.* (2009).^{22, 23} This problem is the most interesting, as we can manipulate the tradeoff by changing k .
- The hard-knapsack problem, which looks for a subset of signed integers summing to 0. Whereas earlier instances of the knapsack problem can be solved in polynomial time, certain parameters are considered hard.⁶⁰ For the latter the best algorithms are given in

Howgrave-Graham and Joux, and Becker *et al.* (2011).^{41, 18} This problem is appealing, as its solution is likely to be unique, and the time and memory complexity are roughly the same.

- The information set decoding problem, which looks for a codeword in random linear code. Many algorithms were proposed for this problem, and many were implemented so we expect them to be well scrutinized.¹⁹ However, in the typical setting the memory complexity is significantly lower than the time complexity.

Among these, the generalized birthday problem appears the most suitable, as its tradeoff steepness is tunable. In the next sections we introduce the problem and build our primary PoW proposal on it.

3. Equihash: Generalized-Birthday Proof-of-Work

In this section we expose the generalized birthday problem and the algorithm for it by Wagner.⁶³

3.1. Problem—The generalized birthday problem for one list is formulated as follows: given list L of n -bit strings $\{X_i\}$, find distinct $\{X_{i_j}\}$ such that

$$X_{i_1} \oplus X_{i_2} \oplus \dots \oplus X_{i_k} = 0.$$

We consider the setting where X_i are outputs of some (non-keyed) PRNG, *e.g.* a hash function H in the counter mode. Thus we have to find $\{i_j\}$ such that

$$H(i_1) \oplus H(i_2) \oplus \dots \oplus H(i_k) = 0. \quad (1)$$

For $k = 1$ this problem is the collision search, and can be solved trivially by sorting in $2^{n/2}$ time and space complexity if $|L| > 2^{n/2}$. However, for $k > 1$ and smaller lists the problem is harder. For instance, from the information-theoretic point of view we expect a solution for $k = 2$ in a list of size $2^{n/4}$, but no algorithm faster than $2^{n/3}$ operations is known.

Wagner demonstrated an algorithm for $k > 1$ and the lists are large enough to contain numerous solutions. It has time and space complexity of $O(2^{\frac{n}{k+1}})$ for lists of the same size. Wagner's algorithm generalizes easily to some operations other than XOR (*e.g.*, to the modular addition). We also note that for $k \geq \log_2 n$ a XOR solution can be found by the much faster Gaussian elimination with complexity of $O(2^k)$ string operations.²⁰

3.2. *Wagner's algorithm*—The basic algorithm to find a solution to Equation (1) is described in Algorithm 1.

Algorithm 1 Basic Wagner's algorithm for the generalized birthday problem.

Input: list L of N n -bit strings ($N \ll 2^n$).

- (1) Enumerate the list as $\{X_1, X_2, \dots, X_N\}$ and store pairs (X_j, j) in a table.
- (2) Sort the table by X_j . Then find all unordered pairs (i, j) such that X_i collides with X_j on the first $\frac{n}{k+1}$ bits. Store all tuples $(X_{i,j} = X_i \oplus X_j, i, j)$ in the table.
- (3) Repeat the previous step to find collisions in $X_{i,j}$ on the next $\frac{n}{k+1}$ bits and store the resulting tuples $(X_{i,j,k,l}, i, j, k, l)$ in the table.
- (4) [...] Repeat the previous step for the next $\frac{n}{k+1}$ bits, and so on until only $\frac{2n}{k+1}$ bits are non-zero.
- (5) $[k+1]$ At the last step, find a collision on the last $\frac{2n}{k+1}$ bits. This gives a solution to the original problem.

Output: list $\{i_j\}$ conforming to Equation (1).

3.3. *Analysis*—For the further text, assume that sorting $l = O(N)$ elements is computationally equivalent to l calls to the hash function H .⁴ Let a single call to H be our time unit.

Proposition 1. For $N = 2^{\frac{n}{k+1}+1}$ and $k^2 < n$ Algorithm 1 produces two solutions (on average) using $(2^{k-1} + n)N/8$ bytes of memory in time $(k+1)N$.

Proof. Suppose we store $N = 2^{\frac{n}{k+1}+1}$ tuples at the first step. Then after collision search we expect

$$(N(N-1)/2)/(N/2) = N-1$$

entries for the second table, then $N-3$ entries for the third table, and so on. Before the last (k -th) collision search we expect $N - 2^{k-1} + 1 \approx N = 2^{\frac{n}{k+1}+1}$ entries, thus on average we obtain two solutions after the last step.

The computational complexity is dominated by the complexity of list generation (N hash calls) and subsequent k sortings of N elements. Therefore, the total computational complexity is equivalent to

$$(k+1)N = (k+1)2^{\frac{n}{k+1}+1}$$

hash function calls. This ends the proof.

We have not computed the variance of the number of solutions, but our experiments demonstrate that the actual number of solutions at each step is very close (within 10%) to the expected number.

If larger lists are used, the table will grow in size over the steps. We have taken the list size exactly so that the expected number of solutions is small and the table size does not change much.

3.4. *Algorithm binding*—The generalized birthday problem in its basic form lacks some necessary properties as a proof-of-work. The reason is that Wagner's algorithm can be iterated to produce multiple solutions by selecting other sets of colliding bits or using more sophisticated techniques.²³ If more memory is available, these solutions can be produced at much lower amortized cost (Proposition 3). Since this property violates the non-amortization requirement for the PoW (Section 2.1), we suggest modifying the problem so that only two solutions can be produced on average.

Our modification is inspired by the fact that a solution found by Wagner’s algorithm carries its footprint. Namely, the intermediate 2^l -XORs have leading $\frac{nl}{k+1}$ bits, for example $X_{i_4} \oplus X_{i_5} \oplus X_{i_6} \oplus X_{i_7}$ collide on certain $\frac{2n}{k+1}$ bits. Therefore, if we pre-fix the positions where 2^l -XORs have zero bits, we bind the user to a particular algorithm flow. Moreover, we can prove that the total number of possible solutions that conform to these restrictions is only 2 on average, so that the problem becomes amortization-free for given input list L . We only have to take care of duplicate solutions which appear if we swap 2^{l-1} -XORs within the 2^l -XOR, for any l . We simply require that every 2^l -XOR is ordered as a pair, *e.g.* with lexicographic order. We stress that a certain order is a prerequisite as otherwise duplicate solutions (produced by swaps in pairs, swaps of pairs, etc.) would be accepted.

With this modification the Gaussian elimination algorithm does not apply any longer, so we can use larger k with no apparent drop in complexity.²⁰

3.5. Time-space tradeoffs—The time-space tradeoffs for Wagner’s algorithm are explored in details in Section 5.2. Here we report the main results. First, we consider optimizations, which are based on methods from.²³

Proposition 2. *Optimized Wagner’s algorithm (Algorithm 2) for $N = 2^{\frac{n}{k+1}+1}$ runs in $M(n, k) = 2^{\frac{n}{k+1}}(2^k + \frac{n}{2(k+1)})$ bytes of memory and $T(n, k) = k2^{\frac{n}{k+1}+2}$ time.⁵*

The next proposition is a corollary from results in Bernstein (2007).²²

Proposition 3. *Using $qM(n, k)$ memory, a user can find $2q^{k+1}$ solutions with cost $qT(n, k)$, so that the amortized cost drops by q^{k-1} .*

Our novel result is the following tradeoffs for standard and algorithm-bound problems.

Proposition 4. *Using $M(n, k)/q$ memory, a user can find 2 solutions in time $C_1(q)T(n, k)$, where*

$$C_1(q) \approx \frac{3q^{\frac{k-1}{2}} + k}{k+1}.$$

Therefore, Wagner’s algorithm for finding 2^k -XOR has a tradeoff of steepness $(k-1)/2$. At the cost of increasing the solution length, we can increase the penalty for memory-reducing users.

Proposition 5. *Using constant memory, a user can find one algorithm-bound solution in time*

$$2^{\frac{n}{2}+2k+\frac{n}{k+1}}.$$

Proposition 6. *Using $M(n, k)/q$ memory, a user can find 2 algorithm-bound solutions in time $C_2(q)T(n, k)$, where*

$$C_2(q) \approx 2^k q^{k/2} k^{k/2-1}.$$

Therefore, the algorithm-bound proof-of-work has higher steepness $(k/2)$, and the constant is larger.

3.6. Parallelism—Thus far we have equalized the time and computational complexity, whereas an ASIC-equipped user or one with a multi-core cluster would be motivated to parallelize the computation if this reduces the AT cost. The following result, also stated in Bernstein (2007), is explained in details in Section 6.²²

Proposition 7. *With $p \ll T(n,k)$ processors and $M(n,k)$ shared memory a user can find 2 algorithm-bound solutions in time $\frac{T(n,k)}{p}(1 - \log_N p)$. Additionally, the memory bandwidth grows by the factor of p .*

For fixed memory size, memory chips with bandwidth significantly higher than that of typical desktop memory (such as DDR3) are rare. Assuming that a prover does not have access to memory with bandwidth higher than certain Bw_{max} , we can efficiently bound the time-memory (and thus the time-area) product for such implementations.

Corollary 3.0.1. *Let the reference memory of size M have bandwidth Bw , and let the prover be equipped with memory chips of bandwidth Bw_{max} . Then the time-area product for the prover can be reduced by the factor $\frac{Bw_{max}}{Bw}$ using $\frac{Bw_{max}}{Bw}$ parallel sorting processors.*

To the best of our knowledge, the highest reported bandwidth in commercial products does not exceed 512 GB/s (Radeon R9 variants), whereas the desktop DDR3 can have as high as 17 GB/s bandwidth.³⁰ Thus we conclude that the highest advantage a prover can get from parallel on-stock hardware does not exceed the factor of 30. This may mean that our proof-of-work is GPU- and desktop-friendly, whereas it is also ASIC-resistant in the sense that an ASIC implementation does not yield smaller time-area product.

4. Primary Proposal

4.1. Specification—To generate an instance for the proof protocol, a verifier selects a cryptographic hash function H and integers n, k, d , which determine time and memory requirements as follows:

- Memory M is $2^{\frac{n}{k+1}+k}$ bytes.
- Time T is $(k+1)2^{\frac{n}{k+1}+d}$ calls to the hash function H .
- Solution size is $2^k(\frac{n}{k+1} + 1) + 160$ bits.
- Verification cost is 2^k hashes and XORs.

Then he selects a seed I (which may be a hash of transactions, block chaining variable, etc.) and asks the prover to find 160-bit nonce V and $(\frac{n}{k+1} + 1)$ -bit x_1, x_2, \dots, x_{2^k} such that

$$\left\{ \begin{array}{l} /* Gen.birthday - condition */ \\ H(I||V||x_1) \oplus H(I||V||x_2) \oplus \dots \oplus H(I||V||x_{2^k}) = 0, \\ /* Difficulty - condition */ \\ H(I||V||x_1||x_2||\dots||x_{2^k}) \text{ has } d \text{ leading zeros,} \\ /* Alg.binding - condition */ \\ H(I||V||x_{w2^l+1}) \oplus \dots \oplus H(I||V||x_{w2^l+2^l}) \\ \text{has } \frac{nl}{k+1} \text{ leading zeros for all } w, l \\ (x_{w2^l+1}||x_{w2^l+2}||\dots||x_{w2^l+2^l-1}) < \\ < (x_{w2^l+2^l-1+1}||x_{w2^l+2^l-1+2}||\dots||x_{w2^l+2^l}). \end{array} \right. \quad (2)$$

Here the order is lexicographical. A prover is supposed to run Wagner's algorithm and then H (Fig. 2).

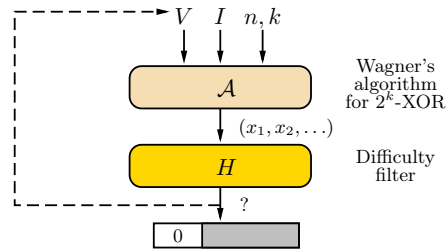


Fig. 2. Equihash: proof-of-work based on the generalized birthday problem .

The analysis in Section 1fsec:birth shows that Wagner’s algorithm produces 2 solutions per call on average for each V . Thus to produce a hash with d leading zeros it must be called 2^{d-1} times with distinct V , which yields the time complexity $(k + 1)2^{\frac{n}{k+1}+d}$. The memoryless verifier checks all the conditions. Note also that computations for $V = V_0$ can not be reused for another $V \neq V_0$. Note that the order of two (or more solutions) produced by Wagner’s algorithm is not important; we do not have to enumerate them; only the one that passes the d -zero test is needed. Furthermore, also note that shorter ($2^l, l < k$) solutions are not allowed neither full solutions based on them (the order prohibits this).

Our proposal fulfills all the properties from Section 2.1. The large AT cost is ensured by $M \geq 2^{30}$. The implementation can be made fast enough to use $M = 2^{29}, n = 144, k = 5$ in 15 seconds with 1 thread. The verification is instant, as it requires only 2^k hashes. The tradeoff has steepness $(k - 1)/2$ and a large constant. Parallelism is restricted due to the memory bandwidth growth in parallel implementations. Optimizations are explored, and amortization does not reduce costs as the number of solutions is small on average. Finally, time, memory, and steepness can be adjusted independently.

4.2. *Implementation and concrete parameters*—Varying n and k we can reach a wide range of the memory and time complexity of our proof-of-work proposal. From the implementation point of view, it is convenient to have $\frac{n}{k+1}$ as multiples of 8 so that we can work with integer numbers of bytes. The solution size in bits is computed by formula

$$L = 2^k \left(\frac{n}{k+1} + 1 \right) + 160.$$

We suggest a wide range of parameters, which cover different memory requirements and tradeoff resistance (Table 1). The memory and time requirements are taken from Proposition 2 with $\epsilon = 0$ and indices trimmed to 8 bits. As a proof of concept, we have implemented and tested Equihash with various parameters. Our implementation is written in C++ with STL without assembly/intrinsic optimizations.⁶ We used bucket sort and extra memory to store the resulting collisions. The performance is reported in Table 2. We see that Equihash runs in a few seconds up to hundreds of MBytes, and can be called progress-free if we consider periods of one minute or longer.

n	k	Complexity			Solution size
		Memory-full		Memoryless	
		Peak memory	Time	Time	
96	5	2.5 MB	$2^{19.2}$	2^{74}	88 B
128	7	8.5 MB	2^{20}	2^{94}	292 B
160	9	32.5 MB	$2^{20.3}$	2^{114}	1.1 KB
176	10	64.5 MB	$2^{20.4}$	2^{124}	2.2 KB
192	11	128.5 MB	$2^{20.5}$	2^{134}	4.4 KB
96	3	320 MB	2^{27}	2^{78}	45 B
144	5	704 MB	$2^{27.5}$	2^{106}	120 B
192	7	4.2 GB	2^{28}	2^{134}	420 B
240	9	16.4 GB	$2^{28.2}$	2^{162}	1.6 KB
96	2	82 GB	$2^{34.5}$	2^{84}	37 B
288	8	131 GB	2^{36}	2^{192}	1.1 KB

Table 1. Concrete parameters and their security level for Equihash. Memory-full complexities are taken from the analysis of Algorithm 2 (Proposition 2). Memoryless complexity is taken from Proposition 5. Time is counted in hash calls.

n	k	Complexity		
		Minimum memory	Time	Solution size
96	5	2.5 MB	0.25 sec	88 B
102	5	5 MB	< 0.5 sec	92 B
114	5	20 MB	< 2 sec	100 B
80	4	1.5 MB	0.2 sec	58 B
90	4	6 MB	1 sec	62 B
100	4	26 MB	4 sec	66 B
96	3	320 MB	10 sec	45 B
144	5	704 MB	15 sec	120 B
200	9	522 MB	10 sec	2.5 KB

Table 2. Performance of our Equihash solver on 2.1 GHz machine with a single thread. Minimum memory is the optimized amount of memory given by Proposition 2; our implementation takes about 4 times as much.

5. Time-Space Tradeoffs, Optimizations for Generalized Birthday Problem

There can be two types of time-space tradeoffs for the generalized birthday algorithm. First, there could be small optimizations in storing the indices, hash outputs, and sorting algorithms. We will show that the straightforward implementation of the generalized birthday algorithm allows a memory reduction by a small factor (2 or 3 depending on the parameters) with about the same increase in the time complexity. However, these optimizations are limited.

If the prover wanted to save further memory, he would have to reduce the total number of tuples. We will show that this approach would cause him harsh computational penalties.

5.1. Optimizations—In Algorithm 1 the index size doubles in size at each step, whereas we can trim $\frac{n}{k+1}$ of intermediate sum per step. There can be two types of optimizations, partly explored in Bernstein *et al.*:²³

- Not storing the intermediate XOR value but recomputing it at each step from the indices. This approach was taken in Bernstein *et al.*²³ However, for large k this approach becomes too expensive.
- Storing only a fraction of index bits, *e.g.* t bits only per index. Then after the last step we have to figure out the missing bits for all 2^k -XOR candidates. For large t this can be done by simply checking the 2-XOR of all subpairs of the 2^k -XOR. For smaller t we essentially have to repeat our algorithm with 2^k different lists, which gives an overhead time factor about k , and $2^{\frac{n}{k+1}+1-t}$ values in each list.

These two optimizations are illustrated in Fig. 3 for $n = 144, k = 5$. It appears that the combination of both optimizations yields the best results, where we recompute the hashes from at first steps, and trim the indices at later steps (Algorithm 2).

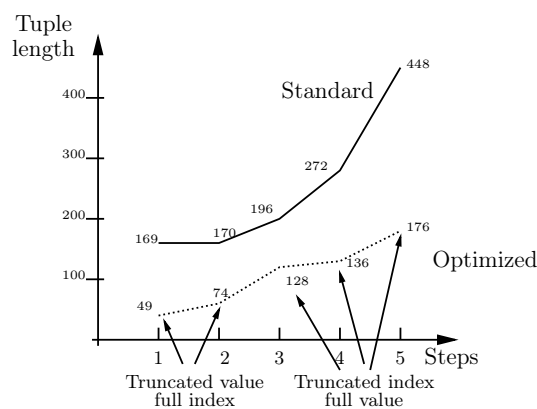


Fig. 3. Tuple length in bits over the steps of Wagner's algorithm in the basic and the optimized implementations.

Algorithm 2 Optimized Wagner’s algorithm for the generalized birthday problem.

Input: list L of N n -bit strings.

- (1) Enumerate the list as $\{X_1, X_2, \dots, X_N\}$ and store pairs (j) in a table.
- (2) Sort the table by X_j , computing it on-the-fly. Then find all unordered pairs (i, j) such that X_i collides with X_j on the first $\frac{n}{k+1}$ bits. Store all such pairs (i, j) in the table.
- (3) Repeat the previous step to find collisions in $X_{i,j}$ (again recomputing it on the fly) on the next $\frac{n}{k+1}$ bits and store the resulting tuples (i, j, k, l) in the table.
- (4) [...] Repeat the previous step for the next $\frac{n}{k+1}$ bits, and so on. When indices trimmed to 8 bits plus the length $X_{i,j,\dots}$ becomes smaller than the full index tuple, switch to trimming indices.
- (5) $[k + 1]$ At the last step, find a collision on the last $\frac{2n}{k+1}$ bits. This gives a solution to the original problem.

Output: list $\{i_j\}$ conforming to Equation (1).

The optimal pattern depends on n and k , so we checked it manually for all suggested parameters assuming that we trim the indices to 8 bits. For all parameters the peak memory use is in the last step, and for all parameters except $n = 96, k = 2, 3$ it is optimal to switch to index trimming before the last step. Therefore, the peak memory is upper bounded by 2^{k-1} 8-bit indices per tuple at the last step. Therefore, at the last step the tuple length is $\frac{2n}{k+1} + 8 \cdot 2^{k-1}$ bits, or $2^{\frac{n}{k+1}} (2^k + \frac{n}{2(k+1)})$ bytes in total. To recover the trimmed bits, we generate $2^{\frac{n}{k+1}-7}$ hash values for each i_j and run the algorithm again, now with 2^k lists, each 2^8 times as small. In the multi-list version of the algorithm, it suffices to keep only k lists in memory simultaneously.⁴⁴ The time complexity of the multi-list step is dominated by generating the 2^k values, as later lists are much smaller. Thus the multi-list phase runs in $2^{\frac{n}{k+1}-7+k}$ time, which is smaller than $(k-1)2^{\frac{n}{k+1}+1}$ for all our parameters. This implies Proposition 2.

5.2. Generic tradeoffs—If a prover wanted to save even more memory than the indices allow, he would have to store fewer tuples at each step. The first time-space tradeoffs of this kind were explored in Bernstein (2007).²² Suppose that the adversary stores only $2^{\frac{n}{k+1}+1}/q$ elements at each step, $q > 1$. Then Bernstein suggests truncating H to $n - (k+1) \log q$ bits and apply Wagner’s algorithm to $n' = n - (k+1) \log q$. After the last step we check if the remaining $(k+1) \log q$ bits are zero, which succeeds with probability q^{-k-1} . Therefore, the algorithm must be repeated q^{k+1} times, but each step is cheaper by the factor of q . Thus the computational penalty is computed as $C(q) = q^k$. We note that the same reasoning applies to the case where more memory is available: if we have $qM(n, k)$ memory for $q > 1$ then we obtain $2q^{k+1}$ solutions in the end, but we spend q times as many computations. This implies Proposition 3.

In the later paper Bernstein *et al.* suggested applying the memoryless collision search method at the last step of the algorithm.²³ They viewed the first $(k-1)$ steps as a single function \mathcal{H} that outputs $\frac{2n}{k+1} + (k+1) \log q$ bits. The complexity of the memoryless collision search is about the square root of the output space size, with more precise estimate of $4 \cdot 2^{l/2}$ for l -bit function.^{62, 53} Therefore, the total computational complexity is

$$4 \cdot 2^{\frac{n}{k+1} + \frac{(k+1) \log q}{2}} \cdot k \cdot 2^{\frac{n}{k+1} + 1 - \log q} = 4k2^{\frac{2n}{k+1} + 1} \cdot q^{(k-1)/2}$$

and the total computational penalty is computed as

$$C(q) \approx 4 \cdot 2^{\frac{n}{k+1}} q^{\frac{k+1}{2}}.$$

Later, Kirchner suggested using available memory for the last step, getting a slightly better tradeoff.^{44, 7} The drawback of both approaches is that they require multiple iterations of the algorithm and thus are very demanding in terms of memory and network bandwidth in practical implementations.

5.3. *Parallel collision search*—The following result is of great importance for our tradeoff exploration. The parallel collision search algorithm for the function that maps m bits to n bits, $m \geq n$, due to van Oorschot and Wiener,⁶² finds T collisions using $2wm$ bits of memory in time

$$\frac{2.5T \cdot 2^{n/2}}{\sqrt{w}}. \quad (3)$$

Here the memory is occupied by w distinguished points, and about $2^{n/2} \sqrt{w}$ calls must be spent to generate those points. Every point is a pair of inputs plus some additional information.

Proof of Proposition 4 Our idea to improve Bernstein's tradeoffs is to increase the number of colliding bits at the first step to $\frac{n}{k+1} + k \log q$ and generate $2^{\frac{n}{k+1}+1}/q$ collisions. The next $(k-2)$ steps require collisions on $\log q$ fewer bits each, and the final step — on $2 \log q$ fewer bits. Then on average we obtain the same 2 collisions at the last step, thus being as lucky as in the memory-full approach. Therefore, we run Wagner's algorithm only once, as soon as we get the necessary amount of inputs. However, the first step is more expensive.

A straightforward approach to generate that many collisions would be to carry Bernstein's idea to the first step of Algorithm 2 and keep only those inputs that yield $k \log q$ leading zero bits. However, this gives the computational penalty factor of q^{k-1} . A better approach is to use the parallel collision search method, where the average cost of one collision grows sublinearly as a function of memory (Equation (3)). Each distinguished point is smaller than the output of H . Using $2^{\frac{n}{k+1}+1}/q$ distinguished points to generate $2^{\frac{n}{k+1}+1}/q$ collisions on $\frac{n}{k+1} + k \log q$ bits, we make

$$5 \cdot 2^{\frac{n}{2(k+1)} + \frac{k \log q}{2} + \frac{n}{2(k+1)} + \frac{1 - \log q}{2}} \approx 3 \cdot q^{\frac{k-1}{2}} \cdot 2^{\frac{n}{k+1}+1}$$

calls to H . The next steps calls the equivalent of $\frac{k}{q} 2^{\frac{n}{k+1}+1}$ hash calls. The success rate is the same. Thus the total computational penalty is estimated as

$$C(q) \approx \frac{3q^{\frac{k-1}{2}} + k}{k+1}.$$

5.4. *Algorithm-bound tradeoffs*—In our proof-of-work proposal we explicitly specify that the solution must carry the footprint of Wagner's algorithm, in particular all intermediate 2^l -XORs must have $\frac{nl}{k+1}$ zeros at certain positions. First we show that the expected total number of solutions that conform to this property is 2. Indeed, there are $2^{\frac{2^k n}{k+1} + 2^k}$ ordered 2^k -tuples of $\frac{n}{k+1}$ -bit values. There are $2^k - 1$ intermediate 2^l -XORs that must have lexicographic order, which reduces the total number of tuples to $2^{\frac{2^k n}{k+1} + 1}$. The restriction of having zeros at certain position contains one $\frac{2n}{k+1}$ -bit condition at the last step, two $\frac{n}{k+1}$ -bit conditions at the step before last, ...,

$2^{k-1} \frac{n}{k+1}$ -bit conditions at the first step, or $2^k \frac{n}{k+1}$ filtering bits in total. Thus there are 2 possible solutions left on average.

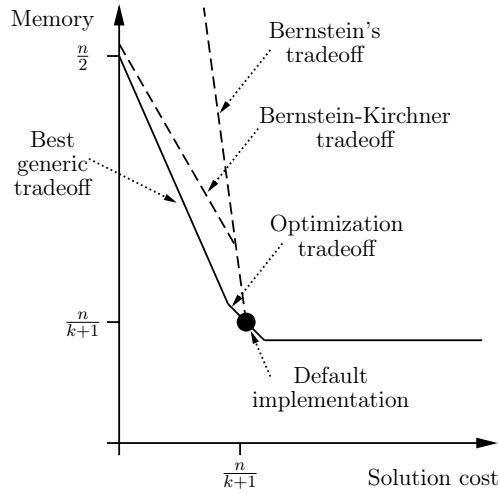


Fig. 4. Computation-memory tradeoffs for the generalized birthday algorithm: ours (Proposition 4), Bernstein’s,²² and “Bernstein-Kirchner”.^{23,44} The complexities are given in \log_2 .

Let us now explore the time-space tradeoffs. It is not evident that the tradeoffs that we obtained in Section 5.2 can be carried out to the algorithm-bound setting. Since the inputs to H are limited to $2^{\frac{n}{k+1}+1}$ values, it is not trivial even to find a memoryless attack with complexity smaller than 2^n . Surprisingly, the tradeoffs for algorithm-bound adversaries are only slightly worse than the original ones.

Proof of Proposition 5 First we show how to find an algorithm-bound solution with very low memory. Recall that the memoryless collision search for f works as the ρ -method: we iterate f till we detect a cycle so that the two different entry points to the cycle constitute a collision. A cycle is detected by either iterating $f()$ and $f(f())$ simultaneously or by storing a few distinguished points along the iteration. The time complexity is about $4 \cdot 2^{l/2}$ for l -bit f for the success rate close to 1.⁵³ However, we might have to truncate f first to ensure that its domain is as large as the range.

It is important to know that the basic memoryless algorithm is seeded, *i.e.* it starts at some point and the eventual complexity is determined by this point. We can imagine an oracle \mathcal{O}_f that takes S as seed and outputs a collision for f .

Consider Equation (2). The algorithm binding requires us to find a solution such that the intermediate 2^l -XORs collide on certain $\frac{nl}{k+1}$ bits. Let us denote such XORs by separate functions:

$$f^{2^l} = H(I||x_1) \oplus H(I||x_2) \oplus \dots \oplus H(I||x_{2^l})$$

Therefore for the original problem we have to find a collision in $f^{2^{k-1}}$, and in the algorithm-bound setting each of the colliding inputs must itself be a collision for $f^{2^{k-2}}$ on fewer bits and so on. At each subsequent step we require a collision on $\frac{n}{k+1}$ bits only, as the colliding bits accumulate from the nodes of the recursion to the root. Note that $f^{2^{k-1}}$ has only $2^{\frac{n}{k+1}+1}$ inputs but a $\frac{2n}{k+1}$ -bit output, *i.e.* it is an expanding function.

This suggests the memoryless solution search as a recursive memoryless collision search. The last step makes $9 \cdot 2^{\frac{3n}{2(k+1)}}$ calls to $\mathcal{O}_{f^{2^{k-1}}}$ (Appendix 1) where the seeds are intermediate inputs to f^{2^k} . Oracle $\mathcal{O}_{f^{2^{k-1}}}$ makes $2^{\frac{n}{2(k+1)}+2}$ calls to $\mathcal{O}_{f^{2^{k-2}}}$, and so on. In total we make $2^{\frac{n}{2}+2k+\frac{n}{k+1}}$ calls to f to find one collision. This ends the proof.

Proof of Proposition 6 The memoryless solution search can be adapted for the reduced memory settings. The idea is to replace the memoryless collision search at each level but the last one with the collision search with distinguished points. In contrast to Section 5.2, we apply the method recursively, and do not store the solutions. However, we have to store distinguished points for each recursion level simultaneously, which affects the computation complexity as follows.

Suppose we have memory sufficient to store only $2^{\frac{n}{k+1}+1}/q$ tuples for some $q > 1$. We split the available memory evenly between k sets of distinguished points (for the sake of simplicity assume that each point is about the same size as the tuple), so that each set contains $\frac{2^{\frac{n}{k+1}+1}}{qk}$ points. The sets are available at all levels. The amortized collision cost using w distinguished points is $2.5 \cdot \frac{2^{n/2}}{\sqrt{w}}$ (Equation (3)). Thus we obtain that oracle $\mathcal{O}_{f^{2^l}}$ makes on average

$$2.5 \cdot 2^{\frac{n}{2(k+1)} - \frac{n}{2(k+1)} - 0.5} \sqrt{qk} \approx 2\sqrt{qk}$$

calls to $\mathcal{O}_{f^{2^{l-1}}}$ to produce a solution. The final oracle makes $2.5 \cdot 2^{\frac{3n}{2(k+1)}+1.5} / \sqrt{w} \approx 2^{\frac{n}{k+1}+2} \sqrt{qk}$ calls to $\mathcal{O}_{f^{2^{k-1}}}$. Therefore, the total computational complexity is

$$2^{\frac{n}{k+1}+1} \cdot 2^k q^{k/2} k^{k/2},$$

and the penalty should be computed as

$$C(q) \approx 2^k q^{k/2} k^{k/2-1}.$$

This ends the proof.

Let us take one of our concrete parameters as an example. Let $n = 144, k = 5$, *i.e.* we suggest finding 32-XOR on 144 bits. A straightforward implementation of the generalized birthday algorithm would require 1.6 GBytes of RAM and about 1 minute of a single CPU core. Recomputing the hash values for the first two steps and truncating the indices to 8 bits at the last two steps, we can decrease the peak tuple length to 176 bits, thus in total requiring 704 MBytes, or aggressively trim to 4 bits, reaching 500 MBytes. However, further reductions are more expensive. Using 2^{24} instead of 2^{25} tuples would cause the computational penalty factor of 2^{10} , and factor of 2^{20} for using 2^{20} tuples ($q = 1/32$). We summarize that for large memory reductions the computational penalty would be prohibitive even for adversaries equipped with a number of parallel computational cores.

5.5. Summary—Here we summarize the security results on Equihash and our security claims.

It is known that the list must contain at least 2^{2^k} tuples for the solution to the 2^k -XOR problem to exist with probability close to 1.⁶³ Since the list entries in Equihash are efficiently generated, this is only a *lower bound on the computational complexity* of any Equihash-solving algorithm.

The lowest computational complexity for the 2^k -XOR problem is still given by Wagner's algorithm.⁶³ We *conjecture* that no faster algorithm will be found in the near future. We also

expect that in the random oracle model the Equihash problem can not be solved with fewer operations than the 2^k -XOR problem if we count memory access as an operation with cost 1, which forms the base of the *optimization-free*. A straightforward implementation of Wagner’s algorithm for Equihash requires more memory for the last steps, and there are several possible optimizations to make the memory use more even. One of them is given in Proposition 2, but we admit that some other modifications with constant factor of memory reduction are possible.

If we ignore the small optimizations and consider a list element as an atomic memory cell, then there is a family of algorithms that trade the memory required by the Wagner’s algorithm for computational complexity. For Wagner’s the reduction in memory results in the tradeoff given by Propositions 4 and 5, and the increase in memory allows many more solutions in the reduced amortized cost (Proposition 3). The algorithm-binding requirement makes the first tradeoff steeper (Proposition 6) and the second one impossible. We conjecture that all these algorithms are optimal, which forms the base of our *tradeoff steepness* requirement.

Compared to proofs-of-space, the time-space tradeoff for Equihash comes from the public scrutiny but the time penalty is significant even for small reductions.³⁵ In contrast, the time penalties in Dziembowski *et al.* have provable lower bounds, but they apply only after reductions by the $O(\log N)$ factor and more, with N being the graph size.³⁵

We do not claim security against parallel attacks, as the sorting procedure within Wagner’s algorithm admits a number of parallel implementations (see next section for details). However, we expect that any practical parallel implementation would have to have very high memory bandwidth, which is currently out of reach for GPU and FPGA. The question whether an efficient parallel ASIC implementation exists remains open.

5.6. *Future cryptanalysis*—New tradeoffs for Wagner’s algorithm may change the security level of our scheme. In our model, we consider it a tradeoff attack if it becomes possible to significantly reduce $C(q)$ for any q . We note that the model does not distinguish the different types of memory, thus a practical speed up coming from using less RAM but much more (say) SSD memory would not be considered a break.

6. Parallelism

6.1. *Parallelized implementations on CPU and GPU*—It is rather easy to analyze Wagner’s algorithm from the parallelism point of view, since it consists of well-known procedures: batch hashing and collision search via sorting.²² Suppose we have p processors with $M(n, k)$ shared memory. The hashing step is straightforward to parallelize: the processors merely fill their own block of memory.

Parallel sorting algorithms have been explored for decades, and full exposition of these results is beyond the scope of this work. Whereas the quicksort is traditional choice for single-thread applications, a number of its variations as well as that of bucket sort, radix sort, sample sort, and many others have been proposed, as they all differ in scalability, computational time, communication complexity, and memory bandwidth. The implementations on a CPU, a multi-core cluster, a GPU, and even FPGA have been reported.^{39, 64}

Proof of Proposition 7 For our purpose a modification of bucket sort, also called a sample sort, suffices. The idea is the following (for the sake of simplicity assume that p is a power of 2). Let

us denote the total number of tuples by N . We partition the entire memory into p^2 equal cells and represent it as a $(p \times p)$ matrix $M[i, j]$. At the first step the processors operate row-wise. They generate the hashes and place them in one of p cells according to the leading $\log p$ bits. This takes time N/p . Thus column j has only entries starting with j (in the bit representation), so there is no collision between the entries from different columns. Then the processors operate column-wise and sort the columns simultaneously. Then each processor goes over the sorted column, identifies collisions, and overwrites the column with placing the collisions into different buckets, so now row j has only entries starting with j . At the next step the processors operate row-wise, and so on. This method requires a small buffer to store the collisions, but due to uniformity of entries it can be rather small (1% in our experiments).

Sorting each column with quicksort requires $O(\frac{N}{p} \log \frac{N}{p})$ time, and this can be done independently for each step of the algorithm. Therefore each collision search step of Wagner's algorithm is faster by the factor

$$\frac{p \log N}{\log \frac{N}{p}} = \frac{p}{1 - \log_N p}.$$

We note that each processor must have access to one entire column and one entire row, so that it is not possible to restrict a processor to its own memory. Therefore, memory conflicts are unavoidable, and the memory chip bandwidth becomes the bottleneck as p increases.

The total bandwidth is calculated as follows. Assuming that the amount of memory operations in sorting is (almost) a linear function of array length, we get that if one core needs R memory reads/writes to sort N entries, then p cores need R/p operations each. In addition, the cores must distribute the collisions into different buckets, spending $O(N)$ memory operations for that. If the running time decreases by the factor of p , then the bandwidth grows at least by the same factor.

This ends the proof.

6.2. Parallel sorting in practice—The observable speedup on multi-core CPU and GPU is not that big. The fastest GPU sortings we are aware of have been reported in Merrill and Grimshaw, where radix sort was implemented and tested on a number of recent GPUs.⁵⁰ The best performance was achieved on GTX480, where 2^{30} 32-bit keys were sorted in 1 second.⁸ The same keys on the 3.2 GHz Core-i7 were sorted with rate 2^{28} keys per second, *i.e.* only 4 times as slow.⁵⁸ Thus the total advantage of GPU over CPU is about the factor of 4, which is even smaller than bandwidth ratio (134 GB/s in GTX480 vs 17 GB/s for DDR3). This supports our assumption of very limited parallelism advantage due to restrictions of memory bandwidth (Satish also mentions high GPU memory latency as a slowing factor).⁵⁸

6.3. Parallel sorting on ASICs—We have subdivided this section into three parts: ASIC implementation that is not quite efficient, mesh-based parallel sorting, and area as prohibitive factor.

6.3.1. ASIC implementation that is not quite efficient—An anonymous reviewer suggested the following ASIC architecture for Equihash. Consider an ASIC hashing chip that performs about 20 GHash/sec. From the Bitcoin mining hardware we estimate it as an equivalent of 50 MB of DRAM.^{27, 38} The chip is supposed to solve the PoW with $n = 144, k = 5$, *i.e.* using 700 MB of RAM. Therefore the chip area is small compared to the RAM area. The reviewer wonders if it is possible to fully utilize the hashing capacity of the chip in this setting.

Our answer is no. Indeed, the Equihash parameters imply the total list length of 2^{25} entries. The total number of hash calls per solution (without the difficulty filter) is $2^{27.5}$, *i.e.* about 200

MHashes. Thus to match the hashing rate the ASIC must produce 100 solutions per second, and thus make 500 sortings of 2^{25} entries per second. The hypothetical sorting rate is about 2^{31} keys per second, or 10 times higher than GTX480 does for 32+128-bit keys. The necessary bandwidth thus can reach 1 TB/sec, which is not reported yet for any memory chip, even for most advanced GPU.⁹ Even if it were the case, the highest reported bandwidth applies to large (a few GB) memory chips, for which it is not so hard to place sufficiently many reading pins to get the bandwidth. For smaller chips (700 MB) we presume it to be much harder.

We conclude that the proposed ASIC architecture would face the memory bandwidth limit quickly. A smaller hashing chip would be more suitable and indeed would probably become a reasonable ASIC implementation. However, the memory will still be the dominating cost in producing such ASICs, as the memory will dominate the area requirements.

6.3.2. Mesh-based parallel sorting—Now we consider a more promising ASIC implementation of Equihash, which utilizes the state of the art for parallel sorting algorithms.

Parallel algorithms for sorting have been explored since late 1960s, but most of the results remained theoretical and were not connected well with potential implementations. Existing algorithms and their analysis can be partitioned into two groups: those that take into account the wire length and bandwidth, and those that do not.

We illustrate the importance of these issues on the following example. Let the array $X[]$ consist of N elements of b bits each. The most straightforward approach to sort the array would be to assign a processor to each element x and count the number of elements greater than x simultaneously for all x , then relocate x to its position in the sorted array. Assuming no memory conflicts, this can be done in time N using N processors, so the time-area product is proportional to N^2 .

This naive approach can be improved in two directions. First, the running time can be decreased significantly. Most of standard sorting algorithms (mergesort, quicksort) were found to be parallelizable due to their divide-and-conquer nature, but the internal merge step was apparently difficult to handle, and optimal algorithms were not found till the late 1980s. Eventually, it was demonstrated that fixed-length integers can be sorted in sublogarithmic time on a sublinear number of processors, thus giving the area-time complexity of order $O(N)$. In turn, the general sorting problem can be solved with a superlinear number of processors in sublogarithmic time, thus having complexity $O(N \log^{1+\epsilon} N)$ almost reaching the time-area lower bound $O(N \log N)$. These results and the survey of previous attempts can be found in Rajasekaran and Reif.⁵⁷ All these algorithms assume random memory access, which make them more suitable for GPUs than ASICs. For example, the butterfly networks that can sort an array in logarithmic time, were found to require the area of at least $O(N^2)$,¹⁵ so the area-time product is at least $O(N^2 \log N)$.

The more promising way to get an ASIC-efficient implementation is to restrict the inter-processor communication. We require that a processor does not make a cross-memory request in order to save the area for wires. There exists a group of parallel algorithms, which sort the array so that the processors work with a few local memory cells only and communicate only to its neighbors. The simplest of them is the so called odd-even transposition sort, which operates in N steps as follows:

- On odd steps, sort pairs $(X[2i], X[2i + 1])$ for every i .
- On even steps, sort pairs $(X[2i - 1], X[2i])$ for every i .

Its time-area product is $O(N^2)$, but the data exchange is local so that it can be implemented

on an ASIC with relatively little wire overhead. It is possible to do better by the quadratic-mesh algorithm by Lang *et al.*, which was advocated by Bernstein to speed up the NFS integer factorization algorithm in Bernstein (2001).^{46, 21} The algorithm needs the memory arranged as a mesh of dimension $\sqrt{N} \times \sqrt{N}$. Each memory slot is assigned with a processor, which is connected to the four neighbors at the mesh. There could be different sorting orders on the mesh: from left to right and then from upper row to lower row; from top to bottom and then from left column to right column; the snakelike order (odd rows are sorted from left to right, and even rows from right to left) and so on. The algorithm in Lang *et al.* can sort in any such order,⁴⁶ and for the former one it works recursively as follows:

- Sort each quadrant (upper ones from left to right, lower ones from right to left).
- Sort columns from top to bottom in parallel using odd-even-transposition algorithm.
- Sort rows in the snakelike order.
- Sort columns from top to bottom.
- Sort rows from left to right.

As the last four steps take \sqrt{N} time each, we get that the time $T(N)$ complexity of the algorithm fulfills the equation

$$T(N) = T(N/4) + 4\sqrt{N}.$$

It implies that $T(N) \approx 8\sqrt{N}$. Thus the time-area product of sorting is $O(N^{3/2})$. The wires are very short, so we think that the real complexity would be close to this number. Since sorting with 1 processor takes time proportional to $O(N \log N)$ (or N if we sort constant-length numbers with radix sort), the advantage ranges from $O(\sqrt{N})$ to $O(\sqrt{N} \log N)$.

These asymptotic estimates are not precise enough to answer decisively if the ASIC implementation of such sorting algorithm would worth the design costs, as the hidden constant factor may make the advantage negligible.

In order to estimate the time and area costs more precisely, we turn to the hardware cost estimates made in Geiselmann and Steinwandt and in Lenstra *et al.* regarding Bernstein's scheme in Bernstein (2001).^{37, 47, 21} For sorting 26-bit numbers and 8-transistor RAM cells they estimate the processor to take between 2000 and 2500 transistors, so that the area increase due to the use of processors is about the factor of 10 or 12. For smaller memory units (*e.g.* DRAM) the processors would be accordingly smaller too. Each step would take 2 cycles then.

For the concrete parameter set ($n = 144, k = 5, 2^{25}$ entries) we thus expect that the implementation of the parallel-mesh algorithm would require a 10-times bigger chip (equivalent of 5 GB RAM), but would finish after $8 \cdot 2^{12.5+1} = 2^{16.5}$ cycles, or in 0.1 milliseconds. We recall that GTX-480 sorts this number of elements using 500 MB of RAM in 150 milliseconds, or 1500 times as slow.⁵⁰ Thus the cost advantage of ASIC would be at most the factor of 150. In fact, producing collisions and hashing the initial counters would add additional slowdown to the ASIC performance and the actual advantage would be even smaller.

We conclude that Equihash can be implemented on ASICs with the time advantage of the factor of about 1000 over GPU and CPU, but the chip would be 10-12 times bigger than a non-parallel ASIC implementation, and for the parameters $n = 144, k = 5$ would be at least as big as 8GB-RAM chip.

6.3.3. Area as prohibitive factor—Despite a large cost reduction, we actually expect that the high design costs would prohibit such implementations. Indeed, an 8 GB single chip would

require significant area, and the probability that some logic is implemented with error becomes very high.

One may argue that even unreliable chips may still produce collisions. Indeed, an error in the hash generation does not affect the result unless this particular hash value is part of the solution (which is unlikely). However, we expect that due to the recursive nature of our sorting algorithm any *comparison error* will diffuse quickly and affect a large part of the output array. For instance, it was demonstrated that any sorting algorithm must make $\Omega(N \log N + kN)$ comparisons to cope with k comparison mistakes.⁴⁵ Therefore, we may hope to keep sorting work in the same time for a logarithmic number of corrupted chips only. For large chips this assumption may be too optimistic.

7. Further discussion

Optimizations with more memory— Proposition 2 illustrated that the memory amount can be slightly reduced at the cost of small computational overhead. From another perspective, the increase in memory use results in small computational savings.

In addition to more efficient memory access, the sorting procedure can be optimized as well. We have experimented with various counting sort algorithms and observed that increased memory use admits faster sorting and thus faster Equihash solvers, although within a small constant factor. This does not contradict the optimization-free property per se, but indicates that actual implementations may not precisely follow the memory and time requirements of Proposition 2. Also if implementors wish to support the choice n, k at run-time, the simplest and/or fastest code may not be the most memory-optimized.

8. Acknowledgement

Various people from the Bitcoin community, in particular Gregory Maxwell for some of the properties that a useful PoW should satisfy. We thank John Tromp for reporting some mistakes in the discussion on Momentum and clarifying some points in the Cuckoo cycle scheme.

9. Conclusion

We have described a general approach to construction of asymmetric proofs-of-work from hard problems. Given a list of requirements for an asymmetric and ASIC-resistant PoW we identified the generalized birthday problem as the one with a scrutinized algorithms decently studied for tradeoffs.

We showed that the running time of the generalized birthday algorithm can be amortized over multiple solutions. We have introduced a technique called *algorithm binding* that prevents solution amortization by making solutions almost unique. Moreover, we investigated the time-memory tradeoffs and demonstrated that the new technique gives time-space tradeoffs that are better for the defender at negligible verification overhead. Thanks to the solution length parameter k in the generalized birthday problem, we may vary the tradeoffs so we suggest a wide range of time, memory, and tradeoff parameters for a variety of applications.

We also demonstrated that even though Wagner's algorithm is inherently parallel, any parallel implementation of its components quickly exhaust available memory bandwidth and thus has

only limited advantage while running on a GPU or ASIC. Altogether, we get a memory-hard, ASIC- and botnet-resistant PoW with extremely fast verification and very small proof size.

To demonstrate the practicality of our solution, we implemented our PoW Equihash on a PC for a 700-MB proof. A reference, non-optimized implementation runs in 15 seconds with 1 thread, verification takes microseconds, and the proof length is tiny, just 120 bytes.

Notes and References

¹ The actual amount of memory in Dwork *et al.* (2005) was 32 MB, which is not that “hard” by modern standards.³⁴

² The project webpage (Tromp) claims that Andersen’s optimizations are now integrated into the miner, but the performance numbers are mainly unchanged since before the cryptanalysis appeared.⁶¹

³ An increase in the AT cost for ASICs can be illustrated as follows. A compact 50-nm DRAM implementation takes 500 mm² per GB, which is equivalent to about 15000 10 MHz SHA-256 cores in the best Bitcoin 40-nm ASICs and is comparable to a CPU size.^{38, 27} Therefore, an algorithm requiring 1 GB for 1 minute would have the same AT cost as an algorithm requiring 2^{42} hash function calls, whereas the latter can not finish on a PC even in 1 day. In other words, the use of memory can increase the AT cost by a factor of 1000 and more at the same time cost for the desktop user.

⁴ The actual ratio depends on the hash function and the sorting algorithm.

⁵ This result was independently obtained in Nikolic and Sasaki.⁵²

⁶ <https://github.com/khovratovich/equihash>

⁷ A rigorous formula is difficult to extract from Kirchner’s manuscript.

⁸ Closer to our $n = 144, k = 5$ proposal, where 48-bit keys are sorted together with 128-bit values, GTX480 sorts 32+128-bit entries with rate $2^{27.5}$ per second, but there is no such benchmark in Satish.⁵⁸

⁹ The exact value highly depends on the sorting method, so we do not give a single number for our scheme.

¹⁰ “Password Hashing Competition.” (2015) (Accessed 29 January 2017) <https://password-hashing.net/>

¹¹ Abadi, M., Burrows, M., Wobber, T. “Moderately Hard, Memory-Bound Functions.” (2003) <http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/2.pdf>

¹² Alwen, J., Blocki, J. “Efficiently Computing Data-Independent Memory-Hard Functions.” *Lecture Notes in Computer Science* **9815** (2016) 241–271

¹³ Alwen, J., *et al.* “On the Memory-Hardness of Data-Independent Password-Hashing Functions.” Cryptology ePrint Archive, Report 2016/783 (2016) (Accessed 29 January 2017) <http://eprint.iacr.org/2016/783>

¹⁴ Andersen, D. “A Public Review of Cuckoo Cycle.” (2014) (Accessed 29 January 2017) <http://www.cs.cmu.edu/~dga/crypto/cuckoo/analysis.pdf>

¹⁵ Avior, A., Calamoneri, T., Even, S., Litman, A., Rosenberg, A. L. “A Tight Layout of the Butterfly Network.” *Theory Comput. Syst.* **31.4** (1998) 475–488

¹⁶ Back, A. “Hashcash – A Denial of Service Counter-Measure.” (2002) (accessed 29 January 2017) <http://www.hashcash.org/papers/hashcash.pdf>

¹⁷ Beame, P., Borodin, A., Raghavan, P., Ruzzo, W. L., Tompa, M. “Time-Space Tradeoffs for Undirected Graph Traversal.” (1990) 429–438 doi:10.1109/FSCS.1990.89563

¹⁸ Becker, A., Coron, J., Joux, A. “Improved Generic Algorithms for Hard Knapsacks.” *Lecture Notes in Computer Science* **6632** (2011) 364–385

¹⁹ Becker, A., Joux, A., May, A., Meurer, A. “Decoding Random Binary Linear Codes in $2^{n/20}$: How $1 + 1 = 0$ Improves Information Set Decoding.” *Lecture Notes in Computer Science* **7237** (2012) 520–536 doi:10.1007/978-3-642-29011-4_31

- ²⁰ Bellare, M., Micciancio, D. “A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost.” *Lecture Notes in Computer Science* **1233** (1997) 163–192 doi:10.1007/3-540-69053-0_13
- ²¹ Bernstein, D. J. *Circuits for integer factorization: a proposal. Technical report* (2001) <https://cr.yp.to/papers/nfscircuit.pdf>
- ²² Bernstein, D. J. “Better Price-Performance Ratios for Generalized Birthday Attacks.” **7** (2007) 160
- ²³ Bernstein, D. J., Lange, T., Niederhagen, R., Peters, C., Schwabe, P. “FSBday.” *Lecture Notes in Computer Science* **5922** (2009) 18–38 doi:10.1007/978-3-642-10628-6_2
- ²⁴ Biryukov, A., Khovratovich, D. “Tradeoff Cryptanalysis of Memory-Hard Functions.” (2015) <http://eprint.iacr.org/2015/227>
- ²⁵ Biryukov, A., Khovratovich, D. “Argon2: new generation of memory-hard functions for password hashing and other applications.” (2016) (Accessed 29 January 2017) <https://www.cryptolux.org/images/0/0d/Argon2.pdf>
- ²⁶ Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J. A., Felten, E. W. “SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies.” (2015) 104–121
- ²⁷ Cawrey, D. “Avalon ASIC’s 40nm Chip to Bring Hashing Boost for Less Power.” (2014) (Accessed 29 January 2017) <http://www.coindesk.com/avalon-asics-40nm-chip-bring-hashing-boost-less-power/>
- ²⁸ Crosby, S. A., Wallach, D. S. “Denial of Service via Algorithmic Complexity Attacks.” **2** (2003)
- ²⁹ Dean, D., Stubblefield, A. “Using Client Puzzles to Protect TLS.” **42** (2001)
- ³⁰ Denneman, F. “Memory Deep Dive: Memory Subsystem Bandwidth.” (2015) (Accessed 29 January 2017) <http://frankdenneman.nl/2015/02/19/memory-deep-dive-memory-subsystem-bandwidth/>
- ³¹ Dinur, I., Dunkelman, O., Keller, N., Shamir, A. “Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems.” *Lecture Notes in Computer Science* **7417** (2012) 719–740 doi:10.1007/978-3-642-32009-5_42
- ³² Dwork, C., Goldberg, A., Naor, M. “On Memory-Bound Functions for Fighting Spam.” *Lecture Notes in Computer Science* **2729** (2003) 426–444
- ³³ Dwork, C., Naor, M. “Pricing via Processing or Combatting Junk Mail.” *Lecture Notes in Computer Science* **740** (1992) 139–147
- ³⁴ Dwork, C., Naor, M., Wee, H. “Pebbling and Proofs of Work.” *Lecture Notes in Computer Science* **3621** (2005) 37–54
- ³⁵ Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K. “Proofs of Space.” *Lecture Notes in Computer Science* **9216** (2015) 585–605 doi:10.1007/978-3-662-48000-7_29
- ³⁶ Fortnow, L. “Time-Space Tradeoffs for Satisfiability.” *J. Comput. Syst. Sci.* **60.2** (2000) 337–353 doi:10.1006/jcss.1999.1671
- ³⁷ Geiselmann, W., Steinwandt, R. “A Dedicated Sieving Hardware.” *Lecture Notes in Computer Science* **2567** (2003) 254–266
- ³⁸ Giridhar, B., *et al.* “Exploring DRAM Organizations for Energy-efficient and Resilient Exascale Memories.” (2013) 23–35
- ³⁹ Helman, D. R., Bader, D. A., JáJá, J. “A Randomized Parallel Sorting Algorithm with an Experimental Study.” *J. Parallel Distrib. Comput.* **52.1** (1998) 1–23 doi:10.1006/jpdc.1998.1462
- ⁴⁰ Hopcroft, J. E., Paul, W. J., Valiant, L. G. “On Time Versus Space.” *J. ACM* **24.2** (1977) 332–337
- ⁴¹ Howgrave-Graham, N., Joux, A. “New Generic Algorithms for Hard Knapsacks.” *Lecture Notes in Computer Science* **6110** (2010) 235–256
- ⁴² Huang, D. Y., *et al.* “Botcoin: Monetizing Stolen Cycles.” (2014)

- ⁴³ Jakobsson, M., Juels, A. “Proofs of Work and Bread Pudding Protocols.” *IFIP Conference Proceedings* **152** (1999) 258–272
- ⁴⁴ Kirchner, P. “Improved Generalized Birthday Attack.” *IACR Cryptology ePrint Archive* **2011** (2011) 377
- ⁴⁵ Lakshmanan, K. B., Ravikumar, B., Ganesan, K. “Coping with Erroneous Information while Sorting.” *IEEE Trans. Computers* **40.9** (1991) 1081–1084
- ⁴⁶ Lang, H., Schimmler, M., Schmeck, H., Schröder, H. “Systolic Sorting on a Mesh-Connected Network.” *IEEE Trans. Computers* **34.7** (1985) 652–658
- ⁴⁷ Lenstra, A. K., Shamir, A., Tomlinson, J., Tromer, E. “Analysis of Bernstein’s Factorization Circuit.” *Lecture Notes in Computer Science* **2501** (2002) 1–26
- ⁴⁸ “Litecoin: Mining hardware comparison.” (2015) (Accessed 29 January 2017) https://litecoin.info/Mining_hardware_comparison
- ⁴⁹ Lorimer, D. “Momentum – A Memory-Hard Proof-of-Work via Finding Birthday Collisions.” (2014) (Accessed 29 January 2017) <http://www.hashcash.org/papers/momentum.pdf>
- ⁵⁰ Merrill, D., Grimshaw, A. “High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing.” *Parallel Processing Letters* **21.02** (2011) 245–272 doi:10.1142/S0129626411000187
- ⁵¹ Minder, L., Sinclair, A. “The extended k -tree algorithm.” (2009) 586–595
- ⁵² Nikolic, I., Sasaki, Y. “Refinements of the k -tree Algorithm for the Generalized Birthday Problem.” *Lecture Notes in Computer Science* **9453** (2015) 683–703
- ⁵³ Nivasch, G. “Cycle Detection Using a Stack.” *Inf. Process. Lett.* **90.3** (2004) 135–140 doi:10.1016/j.ipl.2004.01.016
- ⁵⁴ Park, S., Pietrzak, K., Alwen, J., Fuchsbauer, G., Gazi, P. “Spacecoin: A Cryptocurrency Based on Proofs of Space.” *IACR Cryptology ePrint Archive* **2015** (2015) 528 <http://eprint.iacr.org/2015/528>
- ⁵⁵ Percival, C. “Stronger Key Derivation Via Sequential Memory-Hard Functions.” (2009) (Accessed 29 January 2017) <http://www.tarsnap.com/scrypt/scrypt.pdf>
- ⁵⁶ Pippenger, N. “Superconcentrators.” *SIAM J. Comput.* **6.2** (1977) 298–304 doi:10.1137/0206022
- ⁵⁷ Rajasekaran, S., Reif, J. H. “Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms.” *SIAM J. Comput.* **18.3** (1989) 594–607
- ⁵⁸ Satish, N., *et al.* “Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort.” *SIGMOD ’10* (2010) 351–362
- ⁵⁹ Schroepel, R., Shamir, A. “A $T = O(2^{n/2}), S = O(2^{n/4})$ Algorithm for Certain NP-Complete Problems.” *SIAM J. Comput.* **10.3** (1981) 456–464 doi:10.1137/0210033
- ⁶⁰ Shamir, A. “On the Cryptocomplexity of Knapsack Systems.” (1979) 118–129 doi:10.1145/800135.804405
- ⁶¹ Tromp, J. “Cuckoo Cycle: a memory bound graph-theoretic proof-of-work.” (2014) Cryptology ePrint Archive, Report 2014/059 (Accessed 29 January 2017) <http://eprint.iacr.org/2014/059>, project webpage <https://github.com/tromp/cuckoo>
- ⁶² van Oorschot, P. C., Wiener, M. J. “Parallel Collision Search with Cryptanalytic Applications.” *J. Cryptology* **12.1** (1999) 1–28 doi:10.1007/PL00003816
- ⁶³ Wagner, D. “A Generalized Birthday Problem.” *Lecture Notes in Computer Science* **2442** (2002) 288–303 doi:10.1007/3-540-45708-9_19
- ⁶⁴ Ye, X., Fan, D., Lin, W., Yuan, N., Ienne, P. “High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs.” (2010) 1–10 doi:10.1109/IPDPS.2010.5470445

Appendix

1. *Memoryless collision search for expanding functions and analysis of Momentum*—The memoryless collision search algorithm for functions that maps n bits to n bits is well known.⁵³ It runs in $O(2^{n/2})$ calls to f . However, it is much less efficient if the domain of f is smaller. For instance, suppose that f maps n bits to $2n$ bits (so that only one collision is expected), and we truncate $f(x)$ to certain n bits to iterate it in the Pollard-rho fashion. Then each found collision has only 2^{-n} chance to be the right collision, and we have to rerun the algorithm 2^n times to find the right collision. Therefore, the memoryless algorithm runs in $O(2^{3n/2})$ time in this case.

To explore the full time-memory tradeoff, we turn to an alternative view on this collision search. Finding a collision in an expanding function f mapping n to $n + m$ bits is the same as finding the *golden collision* (i.e. one specific collision) in f truncated to n bits. The golden collision search in a n -bit function has complexity $5 \cdot 2^{3n/2} / \sqrt{w}$ if we have enough memory to store w distinguished points, where a distinguished point has size about $2wn$ bits.⁶² For very small w , this converges to an algorithm with time complexity $9 \cdot 2^{3n/2}$ for an n -bit function.

Consider now the Momentum PoW, where a single collision in F mapping n bits to $2n$ bits is the proof-of-work.⁴⁹ We immediately obtain the following results.

Proposition 8. *There is an algorithm that finds the Momentum PoW in $T_0 = M_0 = O(2^n)$ time and memory.*

Proposition 9. *The time increase in the Momentum PoW is a sublinear function of the memory reduction factor:*

$$T(M_0/q) = \sqrt{q}T(M_0); \quad C(q) = \sqrt{q}.$$

Therefore, the Momentum PoW allows a large reduction in the time-area product as the time grows slower than the area decreases.

Note that both propositions can be viewed as special cases ($k = 1$) of Propositions 5 and 6.

2. *Generic problem composition*—Our primary proposal consists of two independent steps: Wagner’s algorithm \mathcal{A} and the difficulty filter H . We achieved amortization-free and tradeoff steepness just by manipulating \mathcal{A} . Now we consider generic **problem composition** as a tool to get steeper tradeoffs and restrict the number of solutions. It can be used when the algorithm binding method is not applicable.

2.1. *Averaging tradeoffs*—Our idea is to cascade two (or more) problems so that the solution to the first is the input to the second. Interestingly, the resulting time-space tradeoff is better for the verifier than either of original tradeoffs.

Formally let \mathcal{P}_1 and \mathcal{P}_2 be two problems with the following properties:

- \mathcal{P}_1 can be solved in time T with memory M and has strong tradeoffs: any memory reduction causes a large computational penalty.
- \mathcal{P}_2 can be solved in time αT and memory M and has a small, fixed number of solutions.

Let us investigate the time-space tradeoffs for $\mathcal{P}_2 \circ \mathcal{P}_1$. Suppose that $C_1(q)T$ is the amortized cost of finding a solution for \mathcal{P}_1 given qM memory, and $\alpha C_2(q)T$ is the amortized cost of finding a solution for \mathcal{P}_2 given qM memory. Then the amortized cost for the composition $\mathcal{P}_2 \circ \mathcal{P}_1$ of problems is

$$T(q) = C_1(q)T + C_2(q)\alpha T.$$

Since for $q = 1$ the time complexity is $(1 + \alpha)T$, the computational penalty is computed as

$$C(q) = \frac{C_1(q)}{1 + \alpha} + \frac{\alpha C_2(q)}{1 + \alpha}$$

For $\alpha \approx 1$ we get $C(q) > \max(C_1(q), C_2(q))/2$, *i.e.* is at least the half of the largest penalty.

We may want to change the default memory parameters from M to M' , where M' is the maximal memory value such that all memory reductions come at cost above some threshold. This is illustrated in Fig. 5 with $\alpha = 1$: both \mathcal{P}_1 and \mathcal{P}_2 need time T and memory M to be solved but have different tradeoffs. It is worth to increase M to M' so that the decrease from M' would be penalized by \mathcal{P}_1 whereas the increase from M' would be penalized by \mathcal{P}_2 .

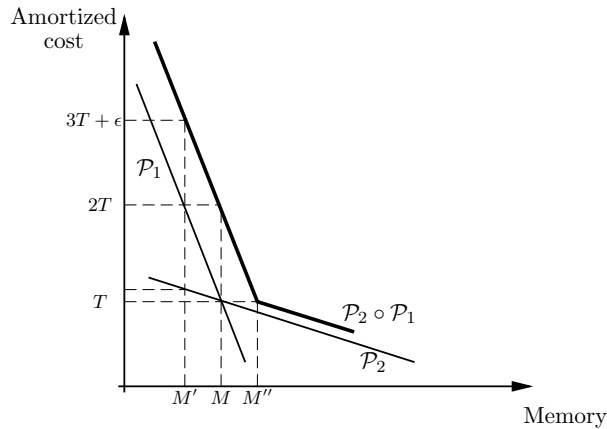


Fig. 5. Time-memory tradeoff for the composition of hard problems with different tradeoffs.

If α is too large or too small, the tradeoff will be largely determined by one of the two problems. In order to balance them, we suggest iterating the faster problem $1/\alpha$ times.

2.2. *Composition with the generalized birthday problem*—Let us investigate which problems can be composed with the generalized birthday problem. The latter with parameters (n, k) can be solved with $2^{k + \frac{n}{k+1}}$ bytes of memory and time equivalent to $2^{1 + \log k + \frac{n}{k+1}}$ calls to the hash function H . Thus the gap between the time and memory exponents is very close; in other words we need as much time as if we would hash the entire memory a few times.

For the second problem \mathcal{P}_2 we have to choose the parameters so that the memory requirements 2^l bytes would be very close to the memory needed by Wagner’s algorithm, *i.e.*

$$l \approx k + \frac{n}{k + 1}.$$

Secondly, the time complexity must be of the same order of magnitude, *i.e.* if solving \mathcal{P}_2 with 2^l memory requires $2^{\beta l}$ time, then

$$\beta \approx \frac{1 + \log k + \frac{n}{k+1}}{k + \frac{n}{k+1}}.$$

Therefore, β must be slightly smaller than 1.

We have searched over several hard problems for such ratio, but the β value is often much larger. For instance, the best information set decoding the algorithm cited in Becker *et al.* (2012) on random linear codes of length n with the full decoding setting have time complexity $2^{0.1n}$ and space complexity $2^{0.076n}$.¹⁹ If we set $n = 400$, then the memory requirements would be 2^{30}

and time would be 2^{40} , which is much higher than 2^{28} time we get for the generalized birthday problem with 2^{30} memory. A better candidate might be the McEliece decoding parameters, for which the algorithm in Becker *et al.* (2012) obtains the time complexity $2^{0.067n}$ and space complexity $2^{0.59n}$.¹⁹

Although the information set decoding algorithms are well studied, we found the hard knapsack problem a bit more scrutinized.^{59, 41} In the further text we explain the problem, show how to instantiate a proof-of-work with it, and describe existing time-memory tradeoffs.

3. *Hard knapsack problem and proof-of-work based on it*—The computational knapsack problem is described as follows. We are given positive numbers a_1, a_2, \dots, a_k, S of length n and have to find $\varepsilon_i \in \{0, 1\}$ such that

$$\varepsilon_1 a_1 + \varepsilon_2 a_2 + \dots + \varepsilon_k a_k = S.$$

The problem is known to be NP-hard, though for a subset of parameters a fast algorithm exists.⁴¹ If $k < 0.94n$ a solution can be found fast using lattice reduction algorithms. Similar algorithms apply when k is much larger than, *i.e.* there are multiple solutions. The hardest setting is $k = n$ in Howgrave-Graham and Joux, where the best algorithms are exponential.⁴¹

In order to construct a proof-of-work protocol, we reformulate the problem similarly to the PoW based on the generalized birthday. We consider the hash function output $H(i)$ as an integer of n bits. We have to find $\varepsilon_i \in \{0, 1\}$ such that

$$\varepsilon_1 H(1) + \varepsilon_2 H(2) + \dots + \varepsilon_n H(n) = H(n+1) \pmod{2^n}$$

and $\sum_i \varepsilon_i = n/2$.

The best existing algorithms so far have been proposed in Howgrave-Graham and Joux and Becker *et al.* (2011).^{41, 18} Though the latter algorithm is asymptotically better, for practical n it is outperformed by the algorithm from the former.⁴¹

The algorithm in Howgrave-Graham and Joux works as follows:

- (1) Choose integer $M \approx 2^{0.51n}$ and random $R < M$. Let us denote $H(n+1)$ by S .
- (2) Solve the original knapsack modulo M with $S = R$ with a set of solutions L_1 and $\sum_i \varepsilon_i = n/4$.
- (3) Solve the original knapsack modulo M with $S = S - R$ with a set of solutions L_2 and $\sum_i \varepsilon_i = n/4$.
- (4) Merge two solution sets and filter out pairs of solutions that activate the same ε_i .

The smaller knapsacks are solved with the algorithm from Schroepel and Shamir so that the $2^{0.31n}$ solutions are produced in time $2^{0.31n}$.⁵⁹ Then the solutions are merged with the total complexity $2^{0.337n}$ (corrected value from Becker *et al.* (2011)) and $2^{0.3}$ memory.¹⁸

The Schroepel and Shamir algorithm works similarly: it chooses $M = 2^{n/2}$, then splits the knapsack in two and solves the left part for M and the right part for $S - M$, then merges the two solutions.⁵⁹

Reducing memory by q results into smaller lists L_1, L_2 and thus the quadratic decrease in the success rate. Since the time complexity per iteration also reduces by q , we obtain the simple time-memory tradeoff $TM = 2^{0.63n}$ up to small M .¹⁸ The best memoryless algorithm found so far has the complexity $2^{0.72n}$.¹⁸

It is reported that the algorithm above runs for $n = 96$ in 15 minutes and requires 1.6 GB of RAM.¹⁸ This memory requirement corresponds to $n = 192, k = 7$ in the generalized birthday

algorithm, where the time complexity is around 2^{28} hash function calls or about 2 minutes on a single thread. Thus we conclude that these problems couple well, but to equalize the time we would have to run the problem \mathcal{P}_1 several times.



Articles in this journal are licensed under a Creative Commons Attribution 4.0 License.



Ledger is published by the University Library System of the University of Pittsburgh as part of its D-Scribe Digital Publishing Program and is cosponsored by the University of Pittsburgh Press.