A Note on the Security of Equihash

Leo Alcock
Independent scholar
leo.c.alcock@gmail.com

Ling Ren
Massachusetts Institute of Technology
renling@mit.edu

ABSTRACT

Proof-of-work (PoW) has recently become the backbone of cryptocurrencies. However, users with Application Specific Integrated Circuits (ASICs) can produce PoW solutions at orders of magnitude lower cost than typical CPU/GPU users. Memory-hard PoWs, i.e., PoW schemes that require a lot of memory to generate proofs, have been proposed as a way to reduce the advantage of ASIC-equipped users. Equihash is a recent memory-hard PoW proposal adopted by the cryptocurrency Zcash. Its simplicity, compact proof size, and tunable parameters make it a good candidate for practical protocols. However, we find its security analysis and claims are flawed. Most importantly, we refute Equihash's claim that its security is based on Wagner's algorithm for the generalized birthday problem. Furthermore, no tradeoff-resistance bound is known for Equihash, and its analysis on the expected number of solution is incorrect. Our findings do not expose any immediate threat to Equihash. The main purpose of this short note is to raise awareness that Equihash should be considered a heuristic scheme with no formally proven security guarantees.

ACM Reference Format:
Leo Alcock and Ling Ren. 2017. A Note on the Security of Equihash. In *Proceedings of CCSW'17*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3140649.3140652

1 INTRODUCTION

Dwork and Naor proposed proof-of-work (PoW) [10] as a countermeasure to spam and denial-of-service attacks. More recently, PoW has been used extensively as the backbone of cryptocurrencies, also known as blockchains. However, cryptocurrency miners soon realized that Application Specific Integrated Circuits (ASICs) can find PoW solutions orders of magnitude faster and more efficiently than commodity CPUs/GPUs. At the time of writing, a state-of-the-art ASIC Bitcoin miner [1] computes SHA256 hashes roughly 200,000× faster and 40,000× more energy efficiently than a state-of-the-art multi-core CPU. The concentration of mining power to ASIC-equipped miners calls into question the decentralization promise of Bitcoin and other cryptocurrencies.

Memory-hard PoWs have been proposed as a promising way to address the above challenge. These PoW schemes require a lot of memory to find solutions efficiently. The reasoning behind this line of work is that since RAMs and storage devices are inherently general-purpose, use of memory should narrow the efficiency gap between ASICs and commodity CPUs/GPUs.

Memory-hard PoW schemes fall into two broad categories based on the memory consumption in verification. If a scheme requires roughly the same amount of memory to find a solution and to verify a solution, then the scheme is called a *symmetric* memory-hard PoW, or a memory-hard function (e.g., scrypt [14]). If a scheme requires significantly less memory to verify a solution than to find a solution, then the scheme is called an *asymmetric* memory-hard PoW. This paper focuses on asymmetric schemes. For cryptocurrencies, asymmetric schemes are superior, and perhaps even necessary, since verification delay will be incurred at every hop in the peer-to-peer network.

The security guarantees that we require from memory-hard PoW schemes are space-time trade-off resistance: i.e., if an adversary wishes to save space by a factor of q when generating a proof, then it must incur a runtime penalty of $\Omega(q)$. The larger the runtime penalty is, the better security (trade-off resistance) a scheme provides. Memory-hard PoW schemes can then be divided into two classes based on whether or not a scheme has a rigorous security (trade-off resistance) proof.

Before we proceed, we need to clarify what we consider as rigorous. As is always the case in cryptography, no security proof is absolute truth. A proof always relies on some assumptions (e.g., hardness of factoring). We say a scheme has a rigorous proof if its security has been reduced to some well-established or well-understood assumptions. On the other hand, we say a scheme has no rigorous proof (yet) if we cannot (yet) reduce its security to any clearly stated assumption other than the degenerate one that simply asserts "the scheme is secure". We call a scheme with no rigorous proof a *heuristic* scheme.

If all other factors are equal, one would prefer a rigorous scheme over a heuristic one. Many heuristic schemes have been found vulnerable to space-time trade-offs or other attacks [3, 4, 16]. But heuristic schemes are often simpler, more efficient, or have other nice properties that may justify their uses in practical protocols. Also, lack of formal proof does not always mean inferior security (though it often does). The renowned RSA public-key encryption algorithm does not have a security proof, but that does not prevent it from making a tremendous impact. In fact, since RSA has stood the test of time, it is now quite reasonable to consider "hardness of RSA" a well-established assumption and reduce other schemes to it.

Among existing asymmetric memory-hard PoW proposals:

- Cuckoo Cycle [16] is a heuristic scheme and is clearly stated as such.
- Momentum [13], Dagger, Ethash [3] and MTP [11] are heuristic schemes. Although their design documents avoid explicitly admitting so, this should be obvious since the design documents never attempted any formal proofs.

CCSW'17, November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5204-8/17/11…$15.00
https://doi.org/10.1145/3140649.3140652

- A few proposals broadly named "proofs of space" [5, 15] have rigorous security proofs under the random oracle assumption [6] and the graph pebbling framework.
- Equihash [7] is a promising proposal and is adopted in Zcash. However, there seems to be some confusion and misunderstandings on whether it is a rigorous scheme or a heuristic one.

In this paper, we study the security proofs/arguments of Equihash and conclude it is a heuristic scheme.

The Equihash paper [7] claims that its security is based on a hard problem, i.e., Wagner's Generalized Birthday Problem (GBP) [17]. However, we notice that the Equihash problem formulation has a subtle yet crucial difference from Wagner's GBP [17]. As a result, the Equihash algorithm is also quite different from Wagner's algorithm [17]. While the difference may appear minor at the first glance, we have seen many examples where seemingly minor changes in problem description drastically affects its complexity class, e.g., Euler tour vs. Hamiltonian cycle, 3SAT vs. 2SAT, just to name a few [8]. In fact, we have already known optimizations that apply to Equihash but not to Wagner's GBP, or vice versa. We will also explain how these misunderstandings have led to a suboptimal parameter choice by Zcash.

At the time of writing, we do not know any trade-off resistance bound for Equihash. The Equihash paper presented and proved several propositions that looked like space-time trade-off resistance analysis. However, these propositions are all in the wrong direction: they gave upper bounds on an adversary's runtime penalty instead of lower bounds. In other words, they give examples of, rather than rule out, what an adversary is capable of. The Equihash designers probably find these propositions useful in ruling out bad parameters, but they add no value to formally reasoning security and should not be confused with trade-off resistance bounds.

While we refute Equihash's security analysis and arguments, we are not advising for or against its use in practical protocols. Our findings do not break Equihash. They simply mean that the security of Equihash, at the moment, cannot be reduced to any well-established assumption, and instead relies on the 2-year old degenerate assumption that "Equihash is secure". It is certainly a possibility that this degenerate assumption stands the test of time and becomes trustworthy. But we do believe it is important for practitioners who adopt Equihash to understand precisely what security assumptions they are relying on.

We then describe a scheme that is indeed based on Wagner's GBP. But we will not consider this new scheme rigorous either, as we find the general philosophy of "basing PoW security on hard problems" itself dubious.

The rest of the paper is organized as follows. Section 2 gives background on Wagner's GBP and Equihash. Section 3 revisits the security analysis, arguments and claims of Equihash. Section 4 describes a scheme based on Wagner's GBP. Section 5 discusses our take on basing PoW security on hard problems. Section 6 concludes.

2 BACKGROUND

Wagner's Generalized Birthday Problem (GBP). Wagner defined the generalized birthday problem as follows [17]: Given 2^k lists $L_1, \ldots, L_{2^k}$ containing (pseudo)-random n-bit strings, find one element from each list $x_i \in L_i$ such that $\oplus_i x_i = 0$.

Wagner's algorithm. Wagner introduced an iterative collision search algorithm to solve the above problem. The algorithm performs a **join** operation in a tree of depth k. The 2^k original lists are placed at the 2^k leaves of the tree. Each **join** operation takes two sibling lists and produces an output list at their parent node in the tree; elements in the output list have the next batch of $\frac{n}{k+1}$ bits canceled out compared to elements in the input lists. The last **join** operation cancels out the remaining $\frac{2n}{k+1}$ bits, so elements in the list at the root of the tree are solutions to the above GBP.

The join operation. The **join** operation is the central subroutine used in both Wagner's algorithm and Equihash's algorithm. The **join** operation finds partial solutions, i.e., a set of elements that cancel out a subset of bits. It takes as input two lists L and L' each containing entries in the format of (x, α), where x is the remaining partial bitstring to be canceled out, and α is the set of indices in the partial solution found so far. We refer to the two components in the entries *bitstrings* and *index sets*, respectively. Entries in the 2^k original input lists only contains bitstrings, and have to be augmented with their offsets in their respective lists. The $\mathbf{join}(L, L', B)$ operation then checks all pairs $(x, \alpha) \in L$, $(x', \alpha') \in L'$, and if $x \oplus x'$ cancels out the next batch of B bits, then it adds a new entry $(x \oplus x', \alpha \,||\, \alpha')$ to the output list, where $||$ represents concatenation.

We give an example of Wagner's algorithm using 4 lists L_1, L_2, L_3, L_4 ($k = 2$). First, place the 4 lists at the leaves a tree of depth 2. Then, obtain two lists at the two intermediate nodes: $L_{1,2} = \mathbf{join}(L_1, L_2, \frac{n}{3})$, $L_{3,4} = \mathbf{join}(L_3, L_4, \frac{n}{3})$. After this step, every element in $L_{1,2}$ or $L_{3,4}$ has its first $n/3$ bits zeroed out. Finally, $L_{1,2,3,4} = \mathbf{join}(L_{1,2}, L_{3,4}, \frac{2n}{3})$ consists of tuples that are solutions to GBP with 4-lists.

Prior analysis of Wagner's algorithm. A particularly interesting case in Wagner's algorithm is when each input list has size $|L_i| := N = 2^{\frac{n}{k+1}}$, $1 \le i \le 2^k$. In this case, the expected size of each intermediate list is preserved at $N^2 \cdot 2^{-\frac{n}{k+1}} = N$. The expected number of solutions, which is also the expected size of the list at the root, is $N^2 \cdot 2^{-\frac{2n}{k+1}} = 1$ (recall that the last join cancels $\frac{2n}{k+1}$ bits).

The **join** operation can be implemented by grouping elements with identical first B bits together, which can be done by sorting or with a hash table. In the case of $N = 2^{\frac{n}{k+1}}$, the **join** operation when implemented using a hash table has an expected time complexity of $\Theta(N)$. If the **join** operations are carried out level by level in the tree, the maximum memory occurs when at height 1, i.e., the level above the leaves. Wagner proposed a space optimization that carries out **join** operations in a postfix order. But his analysis adopted an oversimplified space model [17]. In Section 4.2, we will give more accurate analysis to these variants.

Equihash. Biryukov and Khovratovich introduced the Equihash PoW scheme which is claimed to be based on Wagner's GBP [7]. However, they defined GBP differently: Given a list L of (pseudo-)random n-bit strings, find 2^k distinct elements x_i in L such that $\oplus_i x_i = 0$. Notice the crucial difference that Equihash involves only one list rather than 2^k lists in Wagner's version.

Equihash's algorithm. Due to the difference in problem definition, Equihash's algorithm is also very different from Wagner's algorithm. It also performs **join** operations iteratively, but no longer in

a tree fashion. Instead, each step of the Equihash algorithm is a self-join on a single list. Augment each element in the input list L with its offset in L, and denote the augmented list as $L^{(0)}$. The first $k-1$ steps of the Equihash algorithm computes $L^{(1)} = \mathbf{join}(L^{(0)}, \frac{n}{k+1})$, $L^{(2)} = \mathbf{join}(L^{(1)}, \frac{n}{k+1}), \cdots, L^{(k-1)} = \mathbf{join}(L^{(k-2)}, \frac{n}{k+1})$, In the last step, compute $L^{(k)} = \mathbf{join}(L^{(k-1)}, \frac{2n}{k+1})$ and elements in $L^{(k)}$ are solutions to the Equihash's version of GBP.

The self-join operation $\mathbf{join}(L, B)$ checks all pairs in the input list L, i.e., $(x_1, \alpha_1), (x_2, \alpha_2) \in L$, and if and if $x_1 \oplus x_2$ cancels out the next batch of B bits and that $\alpha_1 \cap \alpha_2 = \emptyset$, then it adds a new entry $(x_1 \oplus x_2, \alpha_1 \cup \alpha_2)$ into the output list. The pairs are unordered, i.e., $(x_1 \oplus x_2, \alpha_1 \cup \alpha_2)$ and $(x_2 \oplus x_1, \alpha_2 \cup \alpha_1)$ are the same element and is added only once.

Prior analysis of Equihash. Equihash sets the initial list size at $|L| = 2N = 2 \cdot 2^{\frac{n}{k+1}}$. Following a similar argument to Wagner's, it was argued that this would preserve the expected list size at intermediate steps, and produces 2 final solutions in expectation. However, Section 3.1 shows this is not the case. In fact, under certain parameter settings recommended by Equihash, the expected number of solutions can be orders of magnitude smaller than 2.

The Equihash paper [7] measures time complexity in n-bit word operations and measures space complexity in bits. We follow this convention. The algorithm presented in the Equihash paper [7] takes $\Theta(kN)$ time and $\Theta((2^k + n)N)$ space. An improvement that reduces space usage is later found by the Equihash designers and Zcash miners [2] independently. We call this improvement *index pointers*. We will give a simple analysis in Section 3.2 to show that index pointers reduce Equihash's space complexity to $\Theta(nN)$. The Equihash paper also suggested an optimization called "index trimming". In Section 3.2 and Section 4.2, we discuss its interactions with Wagner's GBP, Equihash and index pointers.

Equihash also introduced a technique called "algorithm binding" to prevent many optimizations such as the ones outlined by Kirchner [12]. In short, a PoW solution is considered valid only if it carries the footprint of the iterative partial collisions, i.e, at step j, the first $\frac{jn}{k+1}$ bits should XOR to 0. We believe algorithm binding is a great tool for PoW designs since it significantly reduces the attack vector. But we note in Section 5 that it is contradictory to Equihash's philosophy of "basing security on hard problems", a logic we do not find sound.

3 ANALYSIS OF EQUIHASH

The Equihash paper [7] claimed that an Equihash puzzle is an instance of Wagner's GBP, and the best algorithm to find a PoW solution is an improved version of Wagner's algorithm proposed in 2002 [17]. In Section 2, we have already pointed that the Equihash problem is different from Wagner's GBP and problems that look similar may have drastically different complexities. In this section, we first elaborate on a central difference between the two problems in Section 3.1. We then provide further evidence against claiming security based on the two problems' similarities by showing algorithmic improvements that apply to one problem but not the other Section 3.2.

3.1 Index Set Intersection

An central issue overlooked in the Equihash paper [7] is index set intersection. Here, the change from 2^k lists to a single list plays an important role. Recall from Section 2 that the self **join** operation in Equihash must check for index set intersection, i.e., $\alpha_1 \cap \alpha_2 = \emptyset$, before adding a new entry to the output list. At any step in the Equihash algorithm, if a pair of entries share an index in their index sets, this pair must be thrown out even if their next $\frac{n}{k+1}$ bits collide. This is because the problem (rightly so) asks for 2^k distinct elements from L, so the same element cannot be included twice.

Expected number of solutions. An immediate consequence of index set intersection is that Equihash's claim of 2 solutions in expectation is incorrect. Wagner's calculation for list size assumes independence between entries to be joined. This holds in Wagner's GBP because the two entries are always formed from separate lists. In Equihash's case, entries in an intermediate list are correlated if they share common indices. As k increases, the likelihood of a pair of entries sharing indices also increases, and the expected number of solutions deviates further from 2. The precise formula for Equihash's expected number of solutions is derived by [9]. For the parameter choice in Zcash $(n, k) = (200, 9)$, the expected number of solutions is 1.879, which luckily is not too far from 2. But for certain parameter choices, the expected number of solutions can be orders of magnitude smaller than 2. For example, $(n, k) = (192, 11)$, which is among the recommended parameter settings by Equihash, yields 2.1×10^{-7} solutions in expectation. Had Zcash chosen this parameter setting, the actual PoW difficulty would have been orders of magnitude higher than intended.

3.2 Index Pointers and Index Trimming

Index pointers. The index pointer trick [2] is a clever way to store index sets. Recall that in the original Equihash algorithm in Section 2 and [7], each entry in an intermediate list stores a set of indices. The idea of index pointers is that, instead of storing the entire index set $\alpha_1 \cup \alpha_2$, each entry in $L^{(i)}$ stores two pointers to the two source entries in the previous list $L^{(i-1)}$. The full index set can be recovered recursively by tracing the pointers all the way to $L^{(0)}$. This way, the memory space for storing indices grows linearly in k, rather than exponentially. Below, we show that index pointers reduce Equihash space complexity from $\Theta((2^k + n)N)$ to $\Theta(nN)$.

Space reduction from index pointers. We will focus on "good" parameter settings for Equihash in which the list size remains relatively stable at around $2N$. Let us consider the space consumption of index pointers and bitstrings separately. Each pointer takes up $\log_2(2N) \approx 1 + \frac{n}{k+1}$ bits. In step h, each entry only needs to store the remaining $n - \frac{nh}{k+1}$ bits of the bitstring (the first $\frac{nh}{k+1}$ have been canceled out and do not need to be stored). Further observe that once we obtain $L^{(i)}$, the bitstrings in $L^{(i-1)}$ can be discarded, but the index pointers in $L^{(i-1)}$ are still needed. Therefore, after each step, we need about $2(\frac{n}{k+1} + 1)N$ bits to store the extra pointers, but the space taken up by the bitstrings drops by $\frac{n}{k+1}N$ bits. Overall, the total space usage increases by $(\frac{n}{k+1} + 2)N$ after each step. Initially, the space usage is nN, so the maximum space usage during the algorithm occurs at the end, and is roughly $2nN$.

Implication for parameter choices. The discovery of the index pointer technique has a big impact on the choices of Equihash parameters. For any PoW scheme, we naturally would like to increase the required space usage (for ASIC resistance) while keeping the runtime low. A quantity that captures this intuition is the time-space ratio T/S. Equihash without index pointers has $T/S = \Theta(k/(2^k+n))$. Thus, a larger k helps decrease T/S. The Zcash project chose and committed to $(n,k) = (200,9)$ before the discovery of index pointers. Note that a larger k increases the proof size and verification time. Thus, the only reason to choose a larger k is to increase the space usage while keeping runtime low, or equivalently, reducing T/S (regardless of whether the Zcash team examined this metric explicitly). In hindsight, this is almost certainly a mistake. Index pointers reduce the space complexity to $\Theta(nN)$, so $T/S = \Theta(k/n)$. Now a larger k hurts every metric, T/S included.

Index pointers and Wagner's algorithm. While the index pointer technique can also be used in Wagner's algorithm, it does not save space. Storing full index sets in a list at level h in the tree takes $2^h \frac{n}{k+1} N$ bits. Storing index pointers for all the 2^h lists in its subtree also takes $2^h \frac{n}{k+1} N$ bits. Another way to understand the distinction is to notice that in Wagner's GBP, each index in an index set comes from a different list, meaning that the pointers would always trace back to independent entries. Thus, pointers do not save space over full index sets.

Index trimming. Index trimming refers to the technique of storing only a fraction of bits for each index. The Equihash paper credited this idea to Kirchner [12], though we could not find any mention of it in [12]. Regardless of its origin, index trimming is an effective technique to reduce space complexity of Wagner's algorithm. We will provide more details on the use and analysis of index trimming in Wagner's algorithm in Section 4.2. For now, we consider index trimming for Equihash. Contrary to the claims in the Equihash paper [7], we find it unclear how index trimming would work with Equihash. The first challenge is once again related to index set intersection. If indices are trimmed, there is no clear method of detecting possible index set intersections. Furthermore, in the second pass, there is no clear way to keep working with a single list. Instead, if we switch to Wagner's algorithm by replicating the list 2^k times (suggested in [7]), the space and time complexities of the second pass may even exceed the first pass, because the first pass can use index pointers while the second pass cannot. The index pointer technique will save much more space than index trimming and the two methods seem incompatible.

4 A POW BASED ON WAGNER'S GBP

In this section, we discuss and analyze a PoW scheme that adopts Wagner's version of GBP and Wagner's algorithm. Compared to Equihash, it has no obvious pros and cons except that it is related to the 15-year Wagner's GBP rather than the 2-year Equihash GBP. Note that we state that the scheme is related to Wagner's GBP rather than claiming security based on it. In Section 5, we explain that the assumptions PoW schemes need are not conventional "polynomial time" hardness assumptions, but rather the much stronger "exact complexity" assumptions.

4.1 Description of the Scheme

The scheme applies Equihash's algorithm binding idea to Wagner's GBP formulation. The input is 2^k lists $L_1, L_2, \cdots, L_{2^k}$, each of size $N = 2^{\frac{n}{k+1}}$, of pseudorandom n-bit strings. A valid solution of the scheme is a 2^k-tuple $(x_1, x_2, \cdots, x_{2^k})$ $(x_i \in L_i)$ that carries the collision schedule of Wagner's algorithm, i.e., $\forall u, h$, $x_{u2^h+1} \oplus \cdots \oplus x_{u2^h+2^h}$ cancels out the first $\frac{nh}{k+1}$ bits.

Since indices in Wagner's algorithm correspond to entries in separate lists, there is no notion of index set intersection. This also means the two entries to be merged are independent. The expected list size in Wagner's algorithm is thus invariant under parameters $N = 2^{\frac{n}{k+1}}$ and the expected number of solutions is 1.

4.2 Analysis of Wagner's Algorithm

Postfix order. Recall from Section 2 that Wagner's algorithm has a tree structure. Once the two lists are joined into a list at the parent node, the two lists at the child nodes can be deleted. If one performs **join** operations level by level in the tree, then the maximum space usage occurs towards the leaf level, and the space complexity is $\Theta(2^k nN)$. But this is not the most space-efficient algorithm. Wagner proposed evaluating the **join** operations in postfix order [17]. In other words, we prioritize **join** operations that are close to the root and perform them as soon as the two input lists are ready. This way, instead of having $\Theta(2^k)$ in memory, there are at most $\Theta(k)$ lists in memory at any time.

When analyzing space complexity, Wagner assumed that each entry takes $O(1)$ space, and concluded that the space complexity under postfix order is $\Theta(kN)$ [17]. We note that the entry sizes vary a lot based on their heights in the tree as the index sets double while the bitstrings shrink after each **join** operation. Below, we provide a more accurate space complexity analysis.

First, observe that the expected space consumption of a list output by a **join** operation is strictly less than the sum of the two input lists, i.e., $Mem(\mathbf{join}(L,L')) < Mem(L) + Mem(L')$. To prove this, simply note that (1) an index set in $\mathbf{join}(L,L')$ is twice as large as an index set in L or L', (2) a bitstring in $\mathbf{join}(L,L')$ is shorter than a bitstring in L or L', and (3) in expectation $|\mathbf{join}(L,L')| = |L| + |L'|$. As a result, the maximum space usage in postfix order Wagner's algorithm occurs just before the last two leaf lists are joined, at which time there are k lists in memory, one at each height. For a list at height h in the tree, its index set has $\frac{n}{k+1}2^h$ bits and its bitstring has $n - \frac{nh}{k+1}$ bits. Summing over all the lists, we get the memory complexity of postfix Wagner's algorithm

$$N \sum_{h=0}^{k-1} \left(\frac{n}{k+1} 2^h + n - \frac{nh}{k+1} \right) = \Theta((\frac{n}{k+1} 2^k + nk)N).$$

Index Trimming. A good strategy is to trim each index all the way down to 1 bit for maximum space savings. With index trimming, we essentially run Wagner's algorithm twice. In the first pass, we store only the first bit of each index (so each index takes up only 1 bit as opposed to $\frac{n}{k+1}$ bits). The output of this algorithm is the first bits of 2^k indices that correspond to a solution of Wagner's GBP. Then, one would run Wagner's algorithm for a second pass to retrieve the full indices in the solution. In the second pass, we only consider

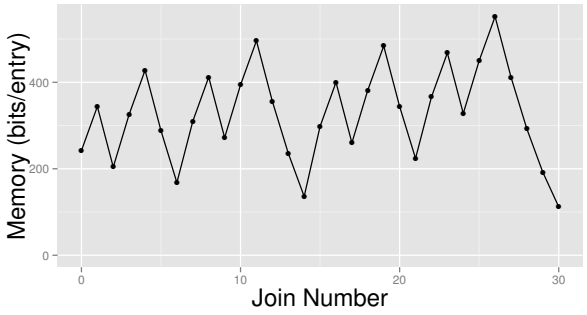

Figure 1: Memory consumption over time of Wagner's postfix order algorithm with $(n,k) = (120,5)$.

entries whose first index bits match the valid solution found by the first pass. Thus, the input lists (at the leaves) have size $N/2$. This way, the expected size of intermediate lists decrease exponentially with their heights in the tree. Thus, the time and space complexity of the second pass are insignificant compared to the first pass. To compute the space complexity after applying index trimming, we simply replace the full index size $\frac{n}{k+1}$ with 1 in the sum.

$$N \sum_{h=0}^{k-1} \left(2^h + n - \frac{nh}{k+1} \right) = \Theta((2^k + nk)N).$$

Memory consumption over time. The memory consumption of postfix order Wagner's algorithm with index trimming fluctuates a lot over time. To give a more intuitive understanding, we plot in Figure 1 the memory consumption of postfix order Wagner's algorithm with index trimming for $(n,k) = (120,5)$. This configuration gives similar maximum space usage with Zcash's configuration of $(200,9)$ with Equihash. As a comparison, Equihash's memory consumption would simply be a straight line between minimum and maximum space usage (cf. Section 3.2).

Time-space ratio. For completeness, we mention that the time-space ratio of Wagner's algorithm (the best version known to date) is $T/S = \Theta(2^k/(2^k+nk))$. When k is not too large, this is comparable to Equihash's time-space ratio $\Theta(k/n)$. In either scheme, k needs to be moderately large (around 4 or 5) to thwart time-space trade-off attacks, but there is little reason to choose k much larger than that as it hurts every efficiency metric.

5 SECURITY ASSUMPTIONS OF POW

In this section, we discuss the security assumptions that PoW schemes rely on. In previous sections, we argued that Equihash's security has only been examined for 2 years, while the scheme in Section 4 traces back to the 15-year old Wagner's GBP. However, we still consider both to be heuristic schemes, not based on the duration of 2 or 15 years, but rather because we do not see a good argument for basing PoW security on hard problems to begin with.

First and foremost, the type of assumptions PoW schemes require are very different from traditional hardness assumptions. Using factoring as an example, a traditional hardness assumption states: "there exists no probabilistic polynomial time algorithm for factoring". A cryptographic primitive relying on the above hardness assumption will remain secure even if a polynomial-factor improvement is later found for factoring. In contrast, a PoW scheme needs an assumption on the precise complexity of a problem, e.g., "the best algorithm known to date for problem X is indeed the best possible algorithm for problem X". This is clearly much stronger than a "polynomial time" hardness assumption. And any algorithmic improvement (even a sublinear or logarithmic one) to problem X constitutes a break for the PoW scheme. Therefore, strictly speaking, the discovery of the index pointer technique has already rendered Equihash broken once.

Also note that there is no reason to require problem X to be hard (e.g., super polynomial). In fact, the Equihash paper has mentioned a naïve construction that relies on a hard problem but turns out to be insecure. It then introduces the nice idea of algorithm binding, which improves security by constraining the Equihash puzzle. This turned the puzzle into an easy problem with a quasi-linear solution. Indeed, PoW schemes ideally want PoWs with matching (or close) upper and lower bounds. Hard problems are usually the opposite given our limited understanding of them. Thus, it should not be surprising that a PoW scheme achieves better security by avoiding hard problems or turning them into easier problems.

6 CONCLUSION

In this paper, we revisited the security claims and arguments of Equihash. We explained that Equihash's security does not follow from Wagner's GBP and remarked on the limitations of the general idea of basing PoW security on hard problems. Overall, we still find Equihash to be a simple and elegant memory-hard PoW heuristic scheme, and still consider it a good choice for practical protocols. But we believe it is important to be aware of its heuristic security. If provable memory hardness is desired, then one may want to look into proof of space protocols instead.

REFERENCES

[1] Antminer S9. Bitmain, https://shop.bitmain.com/market.htm?name=antminer_s9_asic_bitcoin_miner. Accessed: 2017-02-04.
[2] Equihash-xenon. https://github.com/xenoncat/equihash-xenon.
[3] Ethash. https://github.com/ethereum/wiki/wiki/Ethash. Accessed: 2017-08-04.
[4] David G. Andersen. Exploiting time-memory tradeoffs in cuckoo cycle, 2014. (Accessed August 2016) https://www.cs.cmu.edu/ dga/crypto/cuckoo/analysis.pdf.
[5] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: when space is of the essence. In *Security and Cryptography for Networks*, pages 538–557. Springer, 2014.
[6] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.
[7] Alex Biryukov and Dmitry Khovratovich. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. In *NDSS*, 2016.
[8] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
[9] Srinivas Devadas, Ling Ren, and Hanshen Xiao. On iterative collision search for lpn and subset sum. In *Theory of Cryptography*. Springer, 2017.
[10] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology–CRYPTO 1992*, pages 139–147. Springer, 1992.
[11] Alex Biryukov Dmitry Khovratovich. Egalitarian computing. In *NDSS*, 2016.
[12] Paul Kirchner. Improved generalized birthday attack, 2011.
[13] Daniel Larimer. Momentum: a memory-hard proof-of-work via finding birthday collisions.
[14] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009.
[15] Ling Ren and Srinivas Devadas. Proof of space from stacked expanders. In *Theory of Cryptography*, pages 262–285. Springer, 2016.
[16] John Tromp. Cuckoo cycle: a memory-hard proof-of-work system, 2014.
[17] David Wagner. A generalized birthday problem. In *Annual International Cryptology Conference*, pages 288–304. Springer, 2002.