Constructing an Adversary Solver for Equihash

Xiaofei Bai, Jian Gao, Chenglong Hu and Liang Zhang
School of Computer Science, Fudan University
Shanghai Key Laboratory of Data Science, Fudan University
Shanghai Institute of Intelligent Electronics & Systems
{16210240001, 16210240009, 17210240104, lzhang}@fudan.edu.cn

***Abstract*—Blockchain networks, especially cryptocurrencies, rely heavily on proof-of-work (PoW) systems, often as a basis to distribute rewards. These systems require solving specific puzzles, where Application Specific Integrated Circuits (ASICs) can be designed for performance or efficiency. Either way, ASICs surpass CPUs and GPUs by orders of magnitude, and may harm blockchain networks. Recently, Equihash is developed to resist ASIC solving with heavy memory usage. Although commercial ASIC solvers exist for its most popular parameter set, such solvers do not work under better ones, and are considered impossible under optimal parameters. In this paper, we inspect the ASIC resistance of Equihash by constructing a parameter-independent adversary solver design. We evaluate the product, and project at least 10x efficiency advantage for resourceful adversaries. We contribute to the security community in two ways: (1) by revealing the limitation of Equihash and raising awareness about its algorithmic factors, and (2) by demonstrating that security inspection is practical and useful on PoW systems, serving as a start point for future research and development.**

I. Introduction

Proof-of-work (PoW) systems were initially designed to combat spam and some denial of service (DoS) attacks. The basic idea, where anyone requesting service has to solve a computationally-hard puzzle before being verified and served, was proposed as early as 1992 [8], and was later formalized and named in 1999 by Jakobsson [14]. These systems providing quantized fairness are easy to depend on, and quickly gained popularity since the adoption in Bitcoin [18].

Public blockchain applications typically utilize PoW systems in their consensus protocols, where users compete solving PoW puzzles to create blocks (and get corresponding rewards). For an honest user, more computing power (often referred to as hash power by cryptocurrency miners) will bring more chance to create blocks, therefore more rewards. In the same way, adversaries can accumulate hash power to gain control on the consensus, eventually launching attacks or even taking over the system [18].

By design, massive hash power should come with significant cost, dominated by the power consumption of solvers (often called mining rigs). With commodity hardware like CPUs and GPUs, the unit price of hash power stays roughly the same, and the system is fair to everyone. However, dedicated ASIC solvers are more energy-efficient, yielding much higher profitability for their owners. Products like [23] work faster by orders of magnitude but draw very little power. When these products are available to some users but not to others, the system is no longer fair, and is exposed to various risks.

Existing studies [1], [5], [19], [20] have issued the problem and provided decent solutions [10] for cryptographic hashing, but additional factors divert PoW systems from the track, namely (1) PoW systems require stronger ASIC-resistance than message hashing: a 2-3 times efficiency advantage can considerably centralize and weaken blockchain systems, but is too small to cause any problem elsewhere; and (2) PoW systems can utilize additional puzzles that do not have hash-like semantics.

Once open to the public, PoW schemas in blockchain networks are hard to modify, and applying fixes becomes especially difficult. Almost any change to the system would require a successful hard fork, demanding support from most its users and/or miners. Therefore any vulnerability within a PoW schema has to be disclosed to the public before being patched, which is the opposite to common security practices. To this end, PoW schemas must be carefully inspected and hardened beforehand.

Equihash [4] is one of the state-of-the-art PoW schemas achieving considerable ASIC-resistance. It uses the memory-hard approach [7], and when parameters are properly set, is sufficient to eliminate single-chip ASIC solvers. However, we have found a set of weaknesses, allowing efficient multi-chip solvers to be built under all parameters. In this paper, we analyze its software solving algorithm from an adversary's viewpoint, and construct a solver design to exploit the weaknesses. We also work around limitations to make the design as practical as possible, but skipping fine-grained engineering decisions not relevant to the subject.

In general, Equihash provides reasonable memory requirements on both capacity and bandwidth, but its memory usage can be dissected into subroutines with different characteristics and handled accordingly. These subroutines can be separately optimized and then implemented onto a small chip, making full use of its connected memory but consuming very little power. Simulation results with a 28nm library show up to 90% reduction in computation power compared to latest GPUs (under 12nm process), making it safe to project at least 10x

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23337
www.ndss-symposium.org

efficiency advantage for resourceful adversaries.

As Equihash is already deployed in practical applications [1] [2], we decide to present and evaluate our solver design without actually building any miner. Going through the production cycle might allow commercial products to harm the systems before maintainers have time to understand the risks. These applications are more or less moving (or have already moved) to larger parameter sets, where existing ASIC solvers fail [15] but our construction remains effective.

The direct goal of our study is to explore the interaction between PoW schemas and efficient ASIC solver designs, so future systems can achieve better fairness and thus better protection. We also intend to raise the awareness that efficient multi-chip ASIC solvers are possible for Equihash, and that the assets protected by its deployments should be carefully valued. However, this paper is *not* to criticize Equihash. In fact, Equihash is one of the most ASIC-resistant PoW schemas already put into production, and remains so even with our projected ASIC efficiency advantage.

We list our methodology and contributions below:

- We propose a method to inspect PoW systems, by analyzing the algorithm and constructing an adversary solver. Introducing hardware design strategies to the process reveals hidden factors.
- We apply the method on Equihash, especially targeting parameter sets with large memory capacity requirements. At the time of writing, there is no witness of similar designs or implementations.
- We evaluate the produced solver design, estimating its efficiency using simulation results of its core components and power usage of commodity hardware. We then compare it to the best results on CPUs and GPUs, projecting a 10x ASIC advantage as a reference for system deployments and further research.
- We list the factors encountered when constructing the design and discuss their effect on ASIC-resistance.

We start this paper by briefing existing studies including the theories of PoW, ASIC solvers, ASIC-resistant approaches, and Equihash itself. In section III we propose our strategy as an adversary. We apply it to Equihash in section IV and construct a design, using multiple on-chip modules to complete Equihash subroutines. In section V we evaluate the performance and resource usage of our product, proving its practicality and projecting an advantage of resourceful adversaries. We conclude the paper with discussion on our methodology and algorithmic factors affecting the design construction.

II. Background

In this section, we summarize the current status of PoW systems, its major applications, some revelant facts and the dynamics within. We hope to reach a sound basis, concluding some research efforts and known facts, before proceeding with our method and construction.

A. PoW systems and blockchains

The principle of PoW systems is that users requesting something need to demonstrate some computational efforts in a (sometimes implicitly) specified interval of time. The results can be quickly verified by server-side programs to fairly distribute resources, based on the computational cost guaranteed by the PoW schema. Through quantitative control of this cost, service providers can have fine-grained control over the resource distribution, making PoW an effective approach to preventing resource abuse.

Blockchain networks deploy PoW as a critical component, but with some minor changes. In blockchain networks, results are checked by all users instead of servers. We take Bitcoin [18] as an example throughout this section, as it is currently the most significant PoW-protected system both in headcount and in computing resources.

PoW activities within the Bitcoin network can be summarized into the following sequence:

- Miners (a majority of users) try to find a string $nonce$, such that hash result $Hash(block||nonce)$ is less than a dynamic threshold, forming a PoW proof, where $||$ represents string concatenation.
- A valid PoW proof allows a miner to generate a *block*, contributing to Bitcoin's functionality and earning a reward (ie. newly generated Bitcoins).
- The network adjusts the threshold (referred to as difficulty) to control global block generation rate, binding the reward to the computing resources consumed in mining, therefore backing Bitcoin's value.

Naturally miners mine for profit, and seek energy efficiency rather than performance. If presented with a slow but efficient configuration and a performant but less efficient one, they would prefer the former, and deploy more to compensate for the performance loss. To this end, energy efficiency becomes the dominant parameter for mining rigs, as it is directly connected to overall profitability.

In the Bitcoin scenario, the PoW schema consists of massive amounts of SHA256 calls. These simple and repetitive operations are not the intended workloads of CPUs, so dedicated mining rigs soon take over with much higher efficiency. The more specialized they are, the better they are at repeating the hashing task, and the less energy they take to complete the same PoW proof: GPUs overtake CPUs by reducing control flow; Field Programmable Gate Arrays (FPGAs) overpower GPUs by improving parallelism; and the mining business is eventually dominated by ASICs, driving all its competitors unprofitable. For example, decent software produces 24 Mhash/s on CPUs using 100 Watts[3], and around 150 Mhash/s on GPUs at around 300 Watts[4], while a typical ASIC mining rig can perform 18 Thash/s but only drawing 1620 Watts [23].

With more users and powerful mining rigs, Bitcoin's mining difficulty eventually reached a point where it is virtually impossible for individual miners to create blocks alone. Miners

[1]Zcash, https://z.cash/

[2]Bitcoin Gold, https://bitcoingold.org/

[3]Pooler's cpuminer on Intel Core i7-2600.

[4]Cgminer on Nvidia GTX580.

join mining pools to gather hash power and split the awards according to their shares. Mining pools have two major forms: (1) physical pools funded economically, and (2) online pools that process PoW puzzles into easier ones, to be solved by individual miners. In either form, the pool is presented to the network as a single user, and controls all the blocks it creates.

Here, a vulnerability arise within the network's economy. Users tend to switch to more profitable mining pools, which can gradually accumulate hash power to launch attacks. These attacks have already been stated in the original Bitcoin paper [18], but their practicality has greatly increased due to the massive efficiency advantage of ASIC solvers. In fact, a Bitcoin mining pool literally reached 51% hash power of the whole network [3], enough to alter the blockchain to its own interest. Bitcoin eventually mitigated the risk by holding a conference and having the top mining pools discuss a plan, distributing hash power. Most other blockchain systems are still vulnerable, but are not mature enough to have this option.

B. *ASIC solvers and ASIC resistance*

The blockchain communities have been debating about the effects of efficient ASIC solvers and whether or not to resist them. In this section we list major claims from both sides, to show that ASIC resistance is a reasonable concern for blockchain systems. Note that the list may not be complete and some claims may be biased.

In all, ASIC solvers give users more hash power and profitability at higher difficulties, eventually raising network difficulties. Miners with commodity hardware (CPUs and GPUs) tend to quit, and manufacturers optimize their products for better efficiency.

ASIC mining rigs are highly profitable but costly to design and produce, so they sell at incredibly high prices, and are beyond the reach of most individual users from the very beginning. To make matters worse, manufacturers often limit their production and sales to maintain mining profitability. Some models are even deployed directly and never sold at all.

Those supporting ASIC solvers think high difficulties lead to high attack costs, and can prevent botnets from impacting blockchain applications. Those against ASIC solvers state that these products are beyond the reach of many, making blockchains more or less controlled by ASIC miners and solver designers.

In this paper, we do not directly support either of the two sides. For now, ASIC resistance is a reasonable security factor, but its importance is yet to be studied. However, it is safe for us to base our impact on the following facts:

- Efficient ASIC solvers *can* be used to carry out attacks, and it is believed that some products are involved in past attacks [17].
- Claiming to resist ASIC solvers but failing to do so (in this case Equihash under $(144, 5)$ and larger parameter sets) is a vulnerability on its own.

C. *Memory hardness and Equihash*

Later PoW systems use memory-bound functions [1], [5], [19], [20] to achieve ASIC resistance, by requiring large memory capacity and intense memory access. Because fast, on-chip memory has limited capacity; and large, off-chip memory is relatively slow, memory restricts the mass parallelism of ASICs and limits their advantage [19].

Equihash is one of the state-of-the-art approaches in this category, guarding multiple cryptocurrency networks including Bitcoin Gold and Zcash, with a billion-dollar market. At the time of writing, it is one of the two ASIC-resistant PoW schemes [5] that, under any parameter set practically deployed, has the ability to eliminate efficient single-chip ASIC solvers [12].

There are ASIC solvers available, but they are all designed for the popular (yet suboptimal) $(n, k) = (200, 9)$ parameter set. None of them claim or have been witnessed to work on better parameters like $(144, 5)$, and the most known one has even been proved otherwise. Tromp physically inspected the product [15], and did not find enough memory to handle $(144, 5)$, either (1) on a large enough solver core chip, or (2) off-chip memory components.

Equihash uses a modified version of Wagner's Generalized Birthday Problem (GBP) [21] as its puzzle, and then binds solvers to Wagner's algorithm by asking for intermediate results [4]. Alcock [2] detailed the difference between the Equihash puzzle and Wagner's GBP and analyzed its effect.

For convenience, we'll refer to the Equihash puzzle as Single List Generalized Birthday Problem (SLGBP) because it has only one list as input.

Single List Generalized Birthday Problem: Alcock defined the single list generalized birthday problem as follows [2]: Given a list L of (pseudo-)random n-bit strings $\{x_i\}$, find 2^k distinct indices i such that $\bigoplus_i x_i = 0$, where $\bigoplus$ is the XOR operator.

Equihash's algorithm: Equihash's algorithm can be seen as a single-list variation of Wagner's algorithm, where k rounds of **join** operations are performed, each taking the list and produces another, canceling out the next $\frac{n}{k+1}$ bits in every element in every output element. The algorithm in figure 1 shows the exact algorithm steps before applying any optimizations. The **join** operation is the core subroutine of Equihash. It is typically implemented by first sorting the items and then going through the result, generating further index sets and calculating XOR values on the fly [6].

Equihash sets the initial list size at $N = 2^{1+\frac{n}{k+1}}$, to maintain the expectation of list size $|L^{(i)}|$ before the final round.

In the original Equihash paper [4], the authors implemented and tested $(n, k) = (144, 5)$ as their proof of concept. However, this parameter set is not widely used, due to the longer time it takes to solve a single puzzle. The most popular Equihash adopter, Zcash, uses $(200, 9)$ instead [13].

[5]The other one is EtHash.

[6]Some software solvers accomplish this step with hashing, but they are somewhat equivalent to bucket sort.

```
Input : list $L$ of $N$ $n$-bit strings ($N \ll 2^n$)
begin
    Enumerate $L^{(0)}$ as $\{(x_i, \{i\})|i = 1, 2, ..., N\}$
    $r \leftarrow 1$
    while $r < k$ do
join    Sort $L^{(r-1)}$, finding all unordered pairs
          $((x_i, S_i), (x_j, S_j))$ such that $x_i$ collides
          with $x_j$ on the first $\frac{rn}{k+1}$ bits, and that
          $S_i \cap S_j = \varnothing$
        $L^{(r)} \leftarrow \{(x_i \oplus x_j, S_i \cup S_j)|((x_i, S_i), (x_j, S_j))$
          is a found pair $\}$
        $r \leftarrow r + 1$
join Sort $L^{(k-1)}$, finding all unordered pairs
      $((x_i, S_i), (x_j, S_j))$ such that $x_i = x_j$, and that
      $S_i \cap S_j = \varnothing$
    $R \leftarrow \{(S_i \cup S_j)|((x_i, S_i), (x_j, S_j))$ is a found
      pair $\}$
Output: list $R$ of sets of distinct indices
```

Fig. 1: Equihash algorithm for SLGBP

Throughout this paper, we will be presenting our solver and comparing it to software solvers under both parameter sets. However, mining software products are more optimized and statistics have significantly better coverage under $(200, 9)$, so the results are more convincing.

Under $(200, 9)$, an Equihash puzzle takes 2M binary strings of 200 bits as input. In each of the first 8 rounds, an 20-bit substring is taken from each string to find collisions. These bits are discarded, and the remaining strings of each colliding pair are XORed to enter the next round. In the final round, all 40 bits are taken, and 1.879 colliding pairs (or solutions) can be expected [2], [6].

Following memory-hardness theories [1], [7]–[9], Equihash is designed to force intense memory usage, both about capacity and about bandwidth. Even when analyzed with the method proposed later in [19], we find that it achieved moderate bandwidth hardness. These two aspects greatly impact the design and implementation of ASIC solvers.

However, Equihash is not bullet-proof, with various factors limiting its ASIC-resistance:

- Being more of a real-life algorithm (ie. containing real-life subroutines like sorting) rather than cryptographic routine, it is subject to optimizations like the index pointer technique we discuss below.
- CPUs and GPUs are general purpose devices, and under PoW workload, have many components drawing power but not contributing to puzzle solving. At the very least, ASIC solvers can remove them for a small advantage.
- As mentioned in section II-A, the mining business is extremely volatile. Even minor advantages can enable attacks, which already happened to Bitcoin Gold, an Equihash adopter [17].

Index pointer: Index pointer is a technique observed in software solvers like [22] and analyzed in [2].

In the original algorithm, index sets S are carried within the **join** step, and their size expand exponentially with each round. With this technique applied, index sets are stored incrementally as a tree, greatly reducing the memory footprint of Equihash. We notice in section IV-B that this technique can greatly reduce logic and memory access width as well, increasing the efficiency advantage of adversaries.

To summarize this section, blockchain networks are vulnerable to monopoly formation and attacks, as adversaries can reach several times efficiency in PoW solving using ASIC solvers against software. It is hard to fix for deployed blockchain networks, therefore PoW schemas must be carefully chosen, and thoroughly inspected for limitations.

III. Adversary Strategy

Empirically, computing devices tend to consume less power when optimized for area, and will likely be more energy-efficient, though sometimes sacrificing performance. The cryptocurrency mining industry has long been adopting this idea, and has created solver products with incredible efficiency. Optimizing for area brings other benefits as well, like lowered per-chip investment and increased manageability, but in this paper, we focus solely on the most decisive criterion, energy efficiency, as discussed in section II-A.

It is widely agreed that two major methods exist to reduce chip area: (1) apply advanced production technologies to shrink all components, and (2) complete the design using less logic. The first approach does not change the game for anyone, because CPUs and GPUs are manufactured in the same way as ASICs, and always share production technologies. Adversaries are therefore forced to take the latter track, to refine their designs and reduce logic usage.

Recall in section II-C that ASIC solvers (including proposals) and deployments with parameters disabling them [12], [15] both exist for Equihash. An optimal parameter set will eventually be found, and adversaries can not rely on memory-saving tweaks forever. Implementing the whole solver within one chip will sooner or later become impossible, as required in [19]. In this case, an ideal strategy for adversaries is to go through the following design methodology:

1) Analyze the solving algorithm (in this case Wagner's algorithm for SLGBP) and decide on the data to offload from the core solver chip (or solver core for short).
2) Implement the solver core, optimizing for area.
3) Based on data access characteristics (frequency, word length, burst, cost of latency, etc), pick an adequate memory configuration, including type, amount, topology and device parameters like timing.
4) Pick a clock speed for the core, precisely using up memory bandwidth.
5) Tweak non-bottleneck components, trading performance for lower power.

Given a set of PoW puzzles with concrete parameters, this process should always produce a resonable solver. Security inspection can then be performed by evaluating its products and projecting adversary advantage.

IV. Solver construction

In this section we apply the above methodology on Equihash and construct an efficient solver design. As mentioned in section II-C, existing (single-chip) ASIC solvers cannot output valid solutions for parameter sets requiring more memory. Our method uses off-chip memory so the same limitation does not hold. As Equihash adopters are discussing parameter changes (if not already done so like [12]), our construction represents a new type of risk and has to be treated with care.

We perform memory usage analysis and assign an ASIC solution to each subroutine accordingly. We (as adversaries) encountered limitations and resolved some by adjusting Wagner's algorithm. These interactions are good indicators of ASIC-resistance and can be observed similarly in other schemas.

A. Memory usage analysis

To construct the top-level design of an adversary solver, we have to understand how the algorithm accesses data. As Equihash does not include significant lookahead opportunities, we dig directly into the **join** step in figure 1, inspecting its subroutines including sorting/hashing, pair generation, and XOR computation.

Recall that Wagner's algorithm runs k rounds in each attempt to solve the SLGBP, each round taking input data from the previous, and the last round using different parameters. The amount of pairs produced is always random, so we perform our analysis using its mathematical expectations for now.

First of all, the random input L has to be saved. It can not be instantly consumed as the sorting key is only $\frac{n}{k+1}$ bits for the first round. We can save a portion of capacity and bandwidth usage if the key is fed directly to sorting components. A even better option is to have L ready in memory ahead of time, arranging the subroutine in a timeslice with less memory access. We apply this method in section IV-E.

Next, the solver must either sort or hash its input. If done with sorting, optimally only one pass of sequential read is needed. The hashing method need to write the hash table somewhere for use later, introducing two extra memory access passes, write and read. Software solvers use hashing [22] because optimal sorting is not available on CPUs and GPUs. As we are constructing an ASIC design, this is not a problem for us.

Table I lists ideal memory capacity and bandwidth usage for the sorting step. In section IV-B we'll discuss how it is not achievable on popular parameter sets and how we work around it. The tweaks weakens its memory advantage, but we still use it for demonstration thanks to its better flexibility.

When fed with sorted input, the pair generation step itself does not access memory at all. However, due to the index pointer technique, produced index pairs in normal rounds have to be written back to memory, causing memory usage as in table I. The technique has both been proved [2] and field-tested [22] to lower overall memory requirements, so the additional usage here is totally acceptable. Note that the utilized storage builds up incrementally with algorithm rounds, the exact opposite of L, which can have a column discarded in every round.

The XOR calculation step can always get indices from its previous step, but the operands are still in memory and thus need to be fetched. Its results can be partially consumed by the next sort/hash step, with the remaining written back to memory for further XOR rounds. As the input is random, no satisfying cache policy exist for this step, and the subroutine always use memory as indicated in table I.

The final step is to build the index set, both to eliminate groups with duplicate indices and to fulfill the algorithm bound requirements of Equihash. Practically this subroutine is to traverse index pointer trees, using the final round's output as root. We decide to defer this till the end of the algorithm, so index pointer pairs generated in non-final round don't create complex interactions, and can propagate through the pipeline at a line speed of one item or pair per cycle.

Under popular parameters like $(200, 9)$, checking for index set intersection after every round does not significantly reduce list length [2] and is safe to skip. These results can ease the final check, but there is very little to gain because the Equihash schema doesn't produce a lot of final results [4]. This step include very little memory access, and is omitted in the table.

Under some other parameters like $(192, 11)$, the expected number of solutions can drop to 10^{-7} scale. Frequent checking would cause lists to decay quickly, and would often deplete them in early rounds. In this case, it may be beneficial to always perform checks, freeing up the pipeline to process the next puzzle. We ignore these cases here because they exhibit totally different characteristics not discussed in its original paper, thus will unlikely deploy practically without further research.

B. Sorting

An optimal sorting method is actually possible with dedicated hardware, but its linear logic complexity makes it impractical for popular parameter sets. In this section we introduce our sorting peripheral and add merge step, trading memory for logic.

1) Linear sort: The linear sorting technique we use here is similar to the one used by Grozea [11] when CPU, GPU, and FPGA are compared for sorting performance. We slightly tweak its RTL design into figure 2 [7]. Table II describes the actual behavior of a smartcell, and can be directly used in the Look-Up Table (LUT) in figure 2.

A 'smartcell' here consists of two sets of flip-flops (FFs) and one digital comparator. One set of FFs drives the output network while the other stores a value internally. Every cycle, the input value is compared to the stored one. The greater value is sent to output and the lesser one is saved locally. Extra logic is added to handle the head and tail of any sequence passing

[7]The diagram is simplified for better understanding. Clock networks and irrelevant control networks are omitted, and combinational logic is represented with corresponding semantic blocks.

TABLE I: Subroutine memory usage of Wagner's algorithm on SLGBP

subroutine	round	parameterized		$(200, 9)$		$(144, 5)$		$(192, 11)$	
		capacity	bandwidth	capacity	bandwidth	capacity	bandwidth	capacity	bandwidth
storage of L	input	$n * 2^{1+\frac{n}{k+1}}$ b	n b/tick	400 Mib	200 b/tick	4608 Mib	144b/tick	24 Mib	192 b/tick
storage of L	normal	$n * 2^{1+\frac{n}{k+1}}$ b	0	400 Mib	0	4608 Mib	0	24 Mib	0
optimal sort	normal	0	$\frac{n}{k+1}$ b/tick	0	20 b/tick	0	24 b/tick	0	16 b/tick
optimal sort	last	0	$\frac{2n}{k+1}$ b/tick	0	40 b/tick	0	48 b/tick	0	32 b/tick
pair generation	normal	$\frac{(n+k+1)(k-1)}{k+1} * 2^{2+\frac{n}{k+1}}$ b	$\frac{2n+2}{k+1}$ b/tick	672 Mib	42 b/tick	6400 Mib	50 b/tick	43 Mib	34 b/tick
XOR (round i)	normal	$n * 2^{1+\frac{n}{k+1}}$ b	$\frac{3n(k+1-i)}{k+1}$ b/tick	400 Mib	$\leqslant$ 540 b/tick	4608 Mib	$\leqslant$ 480 b/tick	24 Mib	$\leqslant$ 528 b/tick

through, because comparing binary values to nothing does not make sense in our scenario. We add a bit on every set of FFs, representing whether valid data is stored. When only one value is present on a smartcell, it is saved if not, or sent to output if already saved. The other set of FFs are set to invalid.

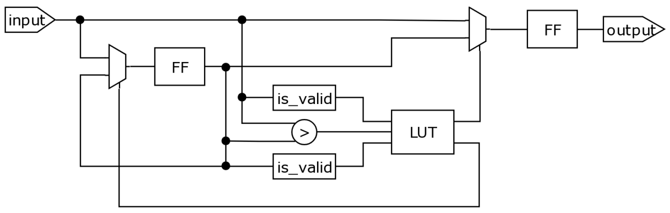


Fig. 2: Simplified RTL diagram of a smartcell

TABLE II: Behavioral truth table of a smartcell

case		behavior	
input	internal	internal	output
greater	lesser	keep	set to input
lesser	greater	set to input	set to internal
valid	invalid	set to input	set to invalid
invalid	valid	set to invalid	set to internal

Chaining N_c smartcells in series, we can accomplish the sorting task with linear time at the cost of linear logic. During the entire sort process, only one sequential pass of reading is performed on the original data. This reduces many memory accesses, thus significantly saving memory bandwidth.

Wagner's algorithm requires sorting with payload, adding extra bits to the FFs but not the comparators. In the original Equihash paper, a payload datum is an index set that doubles in length each round. With the index pointer technique applied, we only need to carry a single index pointer. This pointer can come from a prepending counter and does not need to be read.

This design is simple to implement at RTL level (eg. in Verilog). It works perfectly for short enough sequences, but its linear logic is too much for any reasonable parameter set deployed in production. Under the popular parameter set $(n, k) = (200, 9)$, $N = 2^{21}$. If all of the 2M items are sorted using smart cells ($N_c = N$), the module would need too much power [8] and area to be practical. $(144, 5)$ brings even bigger challenge for adversaries, as 32M items need to be sorted in every pass.

[8]More than 1 kW at 1 GHz, according to section V-B2.

2) Merge sort: Our linear sort peripheral has linear logic complexity, so the merging method in [11] isn't helpful, because it cannot reduce logic. To actually reduce the area and energy cost, the only way is to reduce smartcell instances. This forces us into building exactly one linear sort peripheral that can not handle long sequences, and we therefore have to dump its results into off-chip memory. The merge sort peripheral will not have pipelined input, and have to include a fetcher.

For every x-way merge sort, an extra full pass of sequential write and random read (including both sorting key and payload) is introduced, but reducing the number of smartcell instances to $1/x$, benefitting wider merge sort peripherals. However, off-chip memory access has significant latencies and the random access here need to include prefetch queues. For this to work, each merging way need a private queue, bringing up the cost of wide merging. Under certain parameters, multiple stages of moderately-wide merge sort might provide better overall efficiency. Unlike linear sort, the merge sort here works on processed data, so index pointers need to be read from memory, adding datum width to prefetch queues.

The depth of the prefetch buffers is also an important factor to consider. It has to cover the off-chip memory latency measured in clock cycles, and is thus influenced by both of them. To produce more practical (or pessimistic) efficiency projection and reach a sound conclusion, we set a greater depth (thus higher clock frequency) than needed. An actual adversary can correct these values in the last step of our methodology and gain further energy efficiency.

C. Pair generation

Generating index pointer pairs is a trivial task for either an automaton or a micro controller unit (MCU), as its only tasks are to store the incoming sorting payloads and enumerate pairs out of every group sharing a sorting key. Keeping it up to line speed (one pair per cycle) is also simple but proving so is not.

1) Tail cutting: As stated in the original Equihash paper, the expectation of input list length for each round is always N, so in good cases [9] the length could go well beyond N, demanding extra bits everywhere within the solver and affecting overall throughput. Devadas [6] gave the precise formula for SLGBP's expected number of solutions in 2017, and showed that this number varies greatly depending on parameters.

[9]Longer lists, which tend to produce more solutions, therefore more profit.

TABLE III: Measured effect of tail cutting

tail length	$(200, 9)$		$(144, 5)$	
	solutions	loss	solutions	loss
0	1.53	17.5%	1.80	10.0%
1024	1.64	12.5%	2.00	0
2048	1.66	11.5%	2.00	0
4096	1.66	11.5%	2.00	0

Under $(200, 9)$, 1.879 final solutions are expected. Solutions do not present great values and lengths of intermediate lists $|L^{(r)}|$ do not decay significantly. Under $(144, 5)$, fewer rounds are run. Lengths of $|L^{(r)}|$ are less influenced by the index set intersection problem [2], and the expectation of solution counts is very close to 2. We therefore choose to cut the tail at a point and discard further index pointers.

We ran software simulation on different tail lengths (see table III), and found it reasonable to cut the tail at zero length. The 17.5% loss under $(200, 9)$ may seem a lot, but is totally profitable considering its efficiency gain.

As mentioned in section IV-A, solutions are extremely scarse under some parameter sets, giving these solutions significant economic values in blockchain networks. In this scenerio, good cases in intermediate steps should be carefully preserved, requiring different techniques. The tail-cutting subroutine could significantly hurt profitability here and may not be a good idea. Instead, index checking should be performed to shorten the lists in every round. These parameter sets have not been practically deployed, so we can safely defer the evaluation of corresponding designs for now.

D. Postprocessing

The **xor** computation step takes very little logic and is easy to catch up with other components' speed, given that its memory access can be similarly buffered as in section IV-B2. Even though the design is not trivial and can affect overall efficiency, it is not a serious challenge for ASIC designers. Since we are inspecting Equihash rather than building an actual solver product, knowing its ability to be practically and profitably done is sufficient for our research.

The same holds for the index set construction step at the very end. This subroutine does not require high throughput and can be done with relatively slow MCUs.

E. Pipelining

If the above components run sequentially like software, the solver will spend most of its time in non-bottleneck subroutines, wasting memory bandwidth and power. There are two major solutions: (1) cut the power from idle components, and (2) use a pipelined design to improve performance. They should yield similar efficiency, but the pipeline method has higher profitability (higher per-chip performance), and is more likely to be accepted by adversaries. In this section we apply pipelining to reduce idling within the whole design.

The most applicable subroutine is linear sorting. For every empty cell with valid input, its output becomes valid strictly after 2 clock cycles, so a sorting peripheral with N_c cells start to produce output at precisely $2N_c$ cycles after it is presented with the first input item. It continues to emit exactly one new item every tick until the sequence is depleted. Such behavior can then be modeled and optimized as a $2N_c$-stage pipeline. To prevent two sequences from mixing, an invalid value is inserted as a separator, setting its throughput to one sequence per $N_c + 1$ cycles.

The merge sort module itself has no pipeline capability across input sequences. It ejects all items before starting to process a new sequence. Therefore, no merging unit has valid input when the end-of-sequence marker is emitted. Given that prefetching can start before a queue is depleted and that the marker does not repeat in output, the prefetcher needs a cycle for each queue to either fetch or insert markers. All merging units also need an extra tick to propagate its result, except the last stage which delivers output directly. These operations also consume items from prefetch queues, triggering memory reads, so they cannot go parallel.

A W_m-way merge sort module implemented with N_m merging units can process a sequence of length N every $N + N_m + W_m$ cycles including one used for the marker.

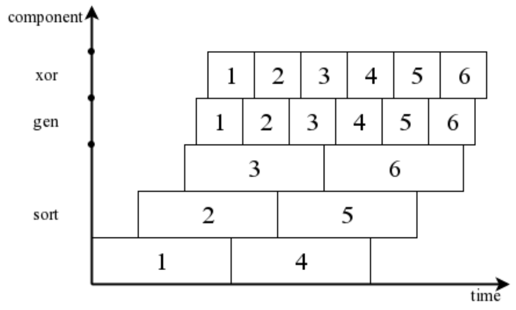
As we perform pair generation, XOR computation and intersection check using simple logic on microcontrollers, we should see line-speed performance as long as connected memory meets bandwidth requirements. The overall behavior of pipelined solvers can be characterized space-time diagrams, as shown in figure 3. Note that the pipeline characteristics of the linear sort stage changes with merge width, so subgraph (b) is not to scale.

V. Solver evaluation

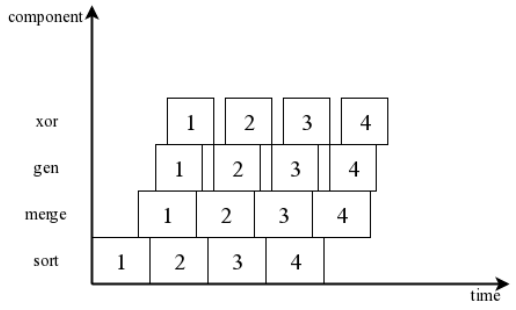
As discussed in previous sections, all underlying peripheral blocks correspond to algorithm-layer semantics and the final solver products might vary in design. In this section, we choose an acceptable arrangement with minimal logic footprint and analyze its overall efficiency. We focus on comparison of our overall efficiency (in solutions/J) to software solvers, because only comparing performance (in solutions/s) or power without mentioning each other is meaningless in the scenario, as mentioned in section II.

We use the most popular set of parameters, $(200, 9)$ so that our results can be directly compared with CPU/GPU solvers. In fact this is the *only* parameter set deployed widely enough to have highly-optimized software solvers and good statistical coverage available. Nevertheless, we still present and evaluate a valid arrangement for (144, 5), as a baseline for future research and deployments.

Under $(200, 9)$, we cut tails at offset 0 and merge sequences twice with 8*4-way merge sort modules, leaving our insertion-sort module with 2048 smartcells. Our example solver completes a single round for every puzzle, so every puzzle should go through the solver k times before producing solutions. We demonstrate the modified algorithm in figure 4 and its data flow in figure 5.



(a) without merging



(b) with one merge stage (not to scale)

Fig. 3: Space-time diagram of pipelined Equihash solvers

Under $(144, 5)$, list L contains significantly more items. As smartcells and prefetch queues are both power-hungry, increasing their numbers is not worthwhile. Adversaries can tackle this by adding a third 8*4-way merge stage and a third buffer block (limiting the insertion-sort module to 1024 cells at the same time).

A. Memory usage

Recall in section III that our design uses off-chip memories so it can handle all Equihash parameters. In table IV we list all off-chip memory usage[10] of our example design. We also use external memory as buffers before every merge sort module, writing one sorting item (both key and payload) and reading one out every cycle. These buffers need to cover two batches in the worst case.

TABLE IV: Off-chip memory usage

parameter	usage	capacity	peak read	peak write
$(200, 9)$	L	1600 Mib	400 b/tick	180 b/tick
	P	2016 Mib	21 b/tick	42 b/tick
	Buffer (each)	248 Mib	62 b/tick	62 b/tick
$(144, 5)$	L	23040 Mib	288 b/tick	120 b/tick
	P	25600 Mib	25 b/tick	50 b/tick
	Buffer (each)	4736 Mib	74 b/tick	74 b/tick

The memory utilization in table IV is far from uniform. The capacity and bandwidth requirements vary between usages, allowing manufacturers to pick suitable configurations to maximize efficiency. It is reasonable to use high-speed memory for L and lower-speed memory for P (to save energy), and possibly on-chip memory for buffers under $(200, 9)$. These options increase ASIC advantage and make our final projection pessimistic for adversaries (or optimistic for the security inspection).

[10]Listed capacity are all calculated with static allocation. No padding is taken into account.

```
Input  : list $L$ of $N$ $n$-bit strings ($N \ll 2^n$)
begin
           Enumerate $L^{(0)}$ as $\{x_i | i = 1, 2, ..., N\}$
           $r \leftarrow 1$
           while $r < k$ do
               Enumerate $X^{(r)}$ with the first $\frac{n}{k+1}$ bits of items in $L^{(r-1)}$, adding indices
               Enumerate $Y^{(r)}$ with the unused bits of items in $L^{(r-1)}$
  sort         Sort $X^{(r)}$ into multiple sequences, preserving indices
  merge        Merge sorted sequences into one, preserving indices
  gen          Find no more than $N$ unordered pairs $(i, j)$ such that $x_i = x_j$
               $P^{(r)} \leftarrow \{(i, j) | (i, j)$ is a found pair$\}$
  xor          $L^{(r)} \leftarrow \{(y_i \oplus y_j) | (i, j)$ is a found pair $\}$
               $r \leftarrow r + 1$
           Enumerate $X^{(k)}$ with all bits of items in $L^{(k-1)}$, adding indices
  sort     Sort $X^{(k)}$ into multiple sequences, preserving indices
  merge    Merge sorted sequences into one, preserving indices
  gen      Find all unordered pairs $(i, j)$ such that $x_i = x_j$
           $P^{(k)} \leftarrow \{(i, j) | (i, j)$ is a found pair$\}$
  check    Reconstruct the list $R$ of index sets using lists $P^{(r)}$ of index pointer pairs, where
           $r = k, k - 1, ..., 1$, removing any set with less than $2^k$ distinct indices
Output: list $R$ of sets of distinct indices
```

Fig. 4: ASIC-assisted Equihash algorithm for good-parameter SLGBP

Consider our pipelining configuration in section IV-E, we work with 4 puzzles at the same time in our $(200, 9)$ configuration. The memory capacity requirement is around 3x that of a single puzzle, except for L, where an extra list is used to store the results from **xor** step. The solver issues 3 reads (two by **xor** and one by **sort**) and one write (by either **xor** or input) to L, totaling 560 bits in the worst case. New puzzles can be written to L simultaneously with the **gen** step of the final round, when no more **xor** step is performed, thus should not add to bandwidth requirements. Assuming we only use memory statically (no dynamic allocations), we would be using 4112 Mib excluding overheads caused by padding. Our bandwidth requirement for L would be 290 Gb/s at 500

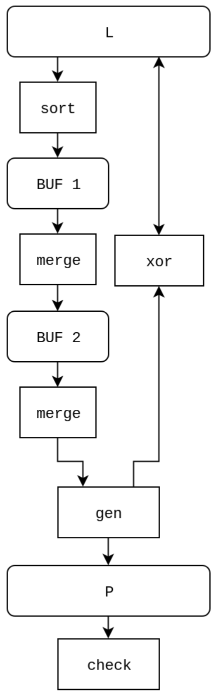


Fig. 5: Internal dataflow of an Equihash solver with two merge steps

MHz and 197 Gb/s at 333 MHz, achievable with commercially available DDR4 modules configured in multiple channels.

Our $(144, 5)$ configuration uses more memory for two reasons: (1) longer L and P, and (2) an extra puzzle is needed to fill the pipeline. The acess patterns stay the same, but bandwidth requirements are actually lowered because items in L are significantly shorter.

B. *Energy efficiency*

1) Performance: Recall that the linear sorting peripheral processes a full-length sequence every $N_c + 1$ cycles, and speeds up when sequences are shorter than expected. This becomes the bottleneck within the solver, whose throughput performance is measured accordingly. In both of our example arrangements, the first merging stages are the bottlenecks, where all end-of-sequence markers are present.

Under $(200, 9)$, one batch is processed every 2.10M cycles. Every equihash puzzle has to be solved k times, so at 500 MHz our example should solve 26.5 puzzles, yielding 40.6 solutions per second. Under $(144, 5)$, one batch is processed every 33.6M cycles. At the same frequency, our design should solve 5.95 puzzles and yield 10.7 solutions per second.

2) Power consumption: Our design uses large amounts of FFs, binary comparators, and some on-chip memory, so we decide to estimate our overall power based on the count of these elements. To obtain these base data, we ran Synopsis Design Compiler on our insertion sort modules and collected power reports under SMIC 28nm HKMG with frequency set to 1 GHz, listed in table V. Although these power reports do not strictly resemble actual chips, they serve as references for IC frontend designers and should suffice in our scenario.

TABLE V: Insertion sort power usage at 1GHz

data width	index width	N_c	register	combinational
20b	21b	64	19.1011 mW	1.9151 mW
20b	21b	128	38.1958 mW	3.5137 mW
20b	21b	192	56.9064 mW	6.9055 mW
20b	21b	256	75.6878 mW	9.8749 mW
20b	21b	384	112.2805 mW	10.0351 mW
40b	21b	64	28.6188 mW	2.6302 mW
40b	21b	128	56.8771 mW	4.6216 mW
40b	21b	256	113.4787 mW	8.6371 mW

In table VI we estimate the power usage of our major modules running at 1 GHz. We assume 70 ns and 5 ns latencies for off-chip and on-chip memory, therefore the queue before each **merge** stage contains 70 items, with 5 items stored directly in FFs.

When actually scaling the clock speed, most modules' power consumption is proportional to frequency, and can be estimated accordingly. If intended for lower frequencies (like below), the prefetch queues can be shorter, thus saving extra energy, and is favorable to adversaries. However, we ignore this optimization, because its effects are minor compared to the overall efficiency advantage.

TABLE VI: Component power estimations at 1GHz

component	$(200, 9)$		$(144, 5)$	
	optimistic	pessimistic	optimistic	pessimistic
smartcell	467 μW	489 μW	562 μW	587 μW
4-way merge	406 μW	471 μW	488 μW	565 μW
8-way merge	1.10 mW	1.38 mW	1.32 mW	1.65 mW
queue (70, 5)	8.26 mW	8.51 mW	9.83 mW	10.1 mW
prefetcher	2.44 mW	2.59 mW	2.93 mW	3.11 mW
sort	956 mW	1.00 W	575 mW	601 mW
merge	272 mW	281 mW	323 mW	333 mW
total	1.50 W	1.56 W	1.54 W	1.93 W

A smartcell saves two sets of items with FFs, together with a binary comparator, and a full insertion-sort module includes N_c of them. An M-to-1 merge unit stores only one item as output and has $C(M, 2)$ comparators. If read latencies of external and on-chip memory are expressed in ticks as L_e and L_o, every prefetch queue is implemented with $L_e - L_o$ items stored in on-chip memory and L_o items in FFs, with its power close to $0.5(L_e + L_o)$ sets of FFs. A prefetcher has an extra (narrow) queue to indicate ownership for fetched data, a set of encoder and decoder, as well as an address counter for each data queue.

Under $(200, 9)$ and at 500 MHz, our solver should produce 40.6 solutions per second and use 0.75-0.78 Watts of power excluding microcontrollers, or 52-54 solutions per Joule. Under $(144, 5)$, we produce 5.36 solutions at the same frequency and using similar power, yielding 5.6-7.0 solutions per Joule. Compared to the best software solvers available [11], our method

[11]Proprietary softwares running on GPUs. Around 4 solutions per Joule under $(200, 9)$ and around 0.4 solutions per Joule under $(144, 5)$.

has already achieved more than ten times core efficiency. The advantage would increase even further if tweaks mentioned in section III and IV-B2 are applied.

Traditionally PoW solvers spend most their power in computation. In our design, computational workloads are reduced so much, the core power usage goes below previously-known overheads including memory, interconnects and microcontrollers. Actual designers might as well want to use higher-bandwidth memory solutions, bringing up the solver's frequency to mitigate the overhead. Although some inevitable deviation is introduced to our results, our goal is not defeated. multi-chip ASIC solvers are still far more efficient than CPUs and GPUs, and Equihash's ASIC resistance is not strong enough.

Ren [19] uses an estimated 0.3 nJ/B memory access cost for ASICs in the methodology, which would be an overhead around 10 W at 500 MHz if applied directly. We project a much lower consumption because:

- Malladi [16] states that modern memory systems has lower cost per access at very high utilization. It is very likely for ASIC Miner manufacturers to use up all the bandwidth to optimize performance by adjusting the operating frequency of the solver core.
- These results are to some extent outdated. Later production technologies and memory interfaces (like DDR4) have lower voltage and are more optimized, thus have lower power consumption.
- The value might have included components other than memory itself, like cache and memory controllers. Our solver does not have any [12] and demand very little from memory controllers, thus might mitigate some cost.

Even if we take the worst-case memory power usage into consideration, and utilize the most pessimistic values, adversaries still have higher efficiency than the latest GPUs. If equally up-to-date production technologies are available, the efficiency will further increase because of smaller components and lower voltages. A projection of 10x advantage is therefore reasonable for resourceful adversaries that apply mentioned optimizations. ASIC solver products, if ever produced, will therefore be profitable, making blockchain applications vulnerable to unforeseen attacks and monopoly formation.

VI. Insights

In this section we discuss our methodology of inspecting PoW schemas and constructing the solver design. We also list factors that have influence on the process, both those enabling our design and those causing difficulties. These factors serve as useful start points for future ASIC-resistance studies, especially in PoW systems.

A. Methodology

Our demonstration shows a practical and useful way to serve as a baseline for future refinements.

[12]Power consumed by buffers already included.

Exploiting as an adversary has long been used by the security community to inspect and ultimately defend systems and applications. It is uncommon for blockchain networks to go through these steps and prove their unlikelihood of monopoly formation. We suspect two major explanations: (1) being unfamiliar to IC-related research effort involved in these systems, and (2) lack of responsible disclosure method. The former problem can be handled by identifying subroutines that can be efficiently implemented on hardware already (possibly via finding products with similar functionalities). It frees researchers from fine-grained details. The latter is more complex due to blockchain's natural resistance towards PoW rule patching.

As mentioned in section II-A, to apply any change onto the PoW schema (and therefore the consensus policy), one would have to launch a hard fork. This requires the fork to gain endorsement from a majority of the network, forcing exploits to be publicly disclosed beforehand. Even though ASIC solvers take months to produce, it may still not be enough to notify enough participants and fork successfully. Fortunately, the weakness we are pointing out is not so significant to need immediate patching. It is sufficient to lower attack profitability and buy time for future actions, just from raising awareness and having users value related assets with corresponding knowledge

B. ASIC-friendly factors

Clear subroutine boundary: The memory requirements of Equihash are provided by multiple subroutines with clear boundaries, exposing larger attack surface compared to other deployed PoW schemas. Multiple subroutines are more likely to be exploited separately, weakening the overall ASIC-resistance.

Real-life subroutines: The sort/hash step in Equihash is more of a real-life algorithm than a cryptographic routine, and is easily vulnerable to all kinds of tweaks and optimizations.

Sequential memory access: Both the sort/hash step and the xor step contains significant sequential memory access, creating a `memcpy`-like workload, increasing ASIC advantage and benefiting special memory configurations not available for CPU/GPUs.

C. ASIC-resistant factors

Algorithm bounding: Equihash implements the idea of algorithm bounding, but only with moderate strength. This prohibits adversaries from using alternate algorithms but not from tweaks like choosing a sort/hash method and cutting tails.

Logic complexity: The idea of logic complexity has long existed for hardware designers, especially when designing acceleration peripherals. The massive amount of logic stops adversaries from applying linear sorting, and could possibly provide alternate ASIC-resistance.

Dynamic resource usage: We mentioned in section II-C and section IV-A that L shrinks and P grows after each round. Solvers have to either (1) provide enough resources (in this case memory) for static allocation, or (2) add logic for dynamic allocation. Since software can accomplish this very well, large-scale, complex and unpredictable resource usage might limit ASIC advantage.

VII. Conclusion

In this paper, we propose a method to inspect the ASIC-resistance of PoW schemas and apply it on Equihash. By constructing a practical and efficient ASIC solver, we discover its limitation in the form of adversary advantage.

We have demonstrated the following contributions:

- It is practical and useful to carry out inspections on PoW systems, by constructing solver designs from adversaries' point of view.
- Equihash resists single-chip ASIC solvers, but is not bullet-proof. Efficient ASIC solvers using off-chip memory are still possible. Solvers following our design work under all parameters, with energy efficiency advantage around 10x.
- We list encountered algorithmic factors and comment on their influence adversary strategy, innovating future PoW schemas.

Our future plan is to refine our methodology to reveal other weaknesses within PoW schemas. We hope the construction procedure can inspire future ASIC-resistance studies, eventually securing related applications. At the same time, we would like to try out new PoW ideas, hopefully to conquer known problems and achieve better fairness.

Acknowledgment

The authors would like to thank Dr. Jinmei Lai from State Key Laboratory of ASIC & System, Fudan University for providing ASIC design suite and related resources.

This work is partially supported by Chinese National Key R&D Program under Grant No. 2017YFB1400604 and Grant No. 2018YFC0831402.

References

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," *ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 2, pp. 299–327, 2005.

[2] L. Alcock and L. Ren, "A note on the security of equihash," in *Proceedings of the 2017 on Cloud Computing Security Workshop*. ACM, 2017, pp. 51–55.

[3] Alex Kerya. (2014) GHash.IO is open for discussion. [Online]. Available: https://blog.cex.io/news/ghash-io-51-percent-5180

[4] A. Biryukov and D. Khovratovich, "Equihash: asymmetric proof-of-work based on the generalized birthday problem," *Proceedings of NDSS 2016*, p. 13, 2016.

[5] F. Coelho, "Exponential memory-bound functions for proof of work protocols." *IACR Cryptology ePrint Archive*, vol. 2005, p. 356, 2005.

[6] S. Devadas, L. Ren, and H. Xiao, "On iterative collision search for lpn and subset sum," in *Theory of Cryptography Conference*. Springer, 2017, pp. 729–746.

[7] C. Dwork, A. Goldberg, and M. Naor, "On memory-bound functions for fighting spam," in *Annual International Cryptology Conference*. Springer, 2003, pp. 426–444.

[8] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual International Cryptology Conference*. Springer, 1992, pp. 139–147.

[9] C. Dwork, M. Naor, and H. Wee, "Pebbling and proofs of work," in *Annual International Cryptology Conference*. Springer, 2005, pp. 37–54.

[10] M. J. Dworkin, "Sha-3 standard: Permutation-based hash and extendable-output functions," Tech. Rep., 2015.

[11] C. Grozea, Z. Bankovic, and P. Laskov, "Fpga vs. multi-core cpus vs. gpus: hands-on experience with a sorting application," in *Facing the multicore-challenge*. Springer, 2010, pp. 105–117.

[12] H4x3rotab. (2018) Network upgrade testnet launch. [Online]. Available: https://github.com/BTCGPU/BTCGPU/issues/324

[13] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification," Tech. rep. 2016-1.10. Zerocoin Electric Coin Company, Tech. Rep., 2016.

[14] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," in *Secure Information Networks*. Springer, 1999, pp. 258–272.

[15] John Tromp. (2018) Let's talk about ASIC mining. [Online]. Available: https://forum.z.cash/t/let-s-talk-about-asic-mining/27353/3332

[16] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 37–48.

[17] MentalNomad. (2018) Double Spend Attacks on Exchanges. [Online]. Available: https://forum.bitcoingold.org/t/double-spend-attacks-on-exchanges/1362

[18] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[19] L. Ren and S. Devadas, "Bandwidth hard functions for asic resistance," in *Theory of Cryptography Conference*. Springer, 2017, pp. 466–492.

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

[21] D. Wagner, "A generalized birthday problem," in *Annual International Cryptology Conference*. Springer, 2002, pp. 288–304.

[22] (2018) Equihash-xenon. [Online]. Available: https://github.com/xenoncat/equihash-xenon

[23] Zhejiang Ebang Communication Co., Ltd. (2018) E10.1 18T. [Online]. Available: http://miner.ebang.com.cn/goods-7.html