libeqh documentation

David A. Dalrymple
libeqh (at) dalrymple.co

(Draft produced October 27, 2016)

Contents

1 Defining the Problem

This library was conceived as a submission to the contest announced at https://zcashminers.org/ (archived) for MIT-licensed[1] Equihash solvers. But what is an "Equihash solver"? Here we aim to put together the information on the problem statement from the official contest rules [1], the primary reference on Equihash [2], and other information inferred from existing implementations of the specific version of Equihash at hand (Zcash-equihash) [3–7].

1.1 Contest API

Let's begin with the API specified by [1], which outlines the inputs and outputs of the Zcash-equihash problem.

```
int SolverFunction(const unsigned char* input,
                   bool (*validBlock)(void* validBlockData, const unsigned char* solution),
                     void* validBlockData,
                   bool (*cancelled)(void* cancelledData),
                     void* cancelledData,
                   int numThreads,
                   int n, int k);
```

[1]The Free Software Foundation points out that this term is ambiguous — not only has MIT used many licenses for various projects over the years, but the term is actually in common use to refer to both the Expat license and the X11 license. The version specified by https://zcashminers.org/rules is the Expat license.

1.1.1 Asynchrony & Threading

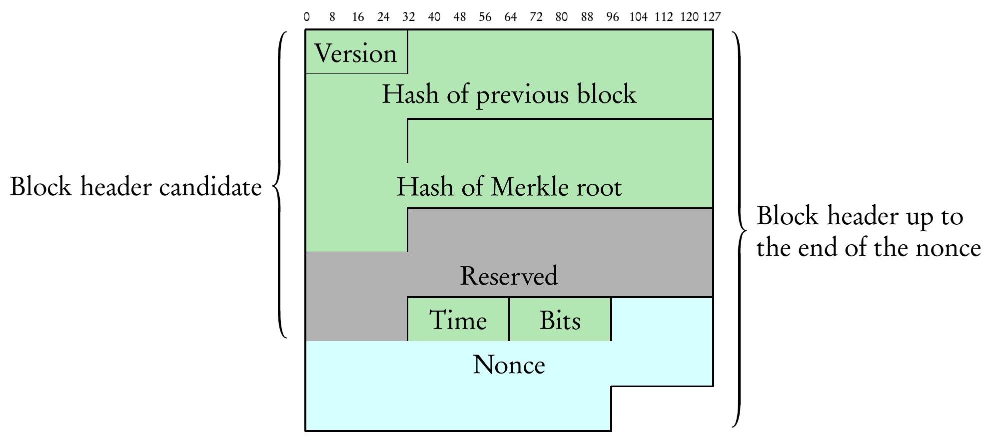
This appears to be an *asynchronous* API: `SolverFunction` is expected to spin up `numThreads` new threads, then quickly return as those threads begin to do the actual work. Upon finding a `solution`, the callback `validBlock` should be invoked. At various points during the solving process, the `cancelled` callback should be invoked to check whether to abort. `validBlock` and `cancelled` are both given with associated **`void*`** closures (`validBlockData` and `cancelledData` respectively), which should be passed along in the natural way.

1.1.2 Return value

One problem with this interpretation is that the API specifies that `SolverFunction`'s *return value* is the number of solutions found (or −1 on error). Yet, the number of solutions is not ready until the work is completed, so this is incompatible with the fast-returning asynchronous model. Our reconciliation of this is that we return `0` on success and −1 on error; after all, at the time success is signaled, `0` solutions have been found. The actual information about how many solutions `libeqh` finds is conveyed by the number of times `validBlock` is invoked.

1.1.3 Input & Output

`input` points to a 140-byte block of memory, the "block header up to end of the nonce", which is laid out as follows:



Our goal is to compute a `solution` which completes the block header, like so:

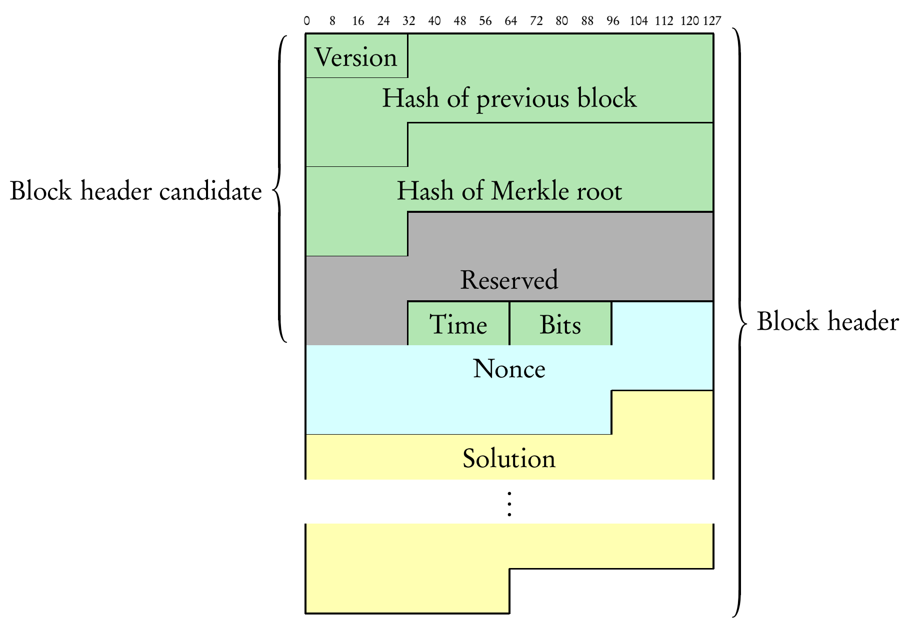
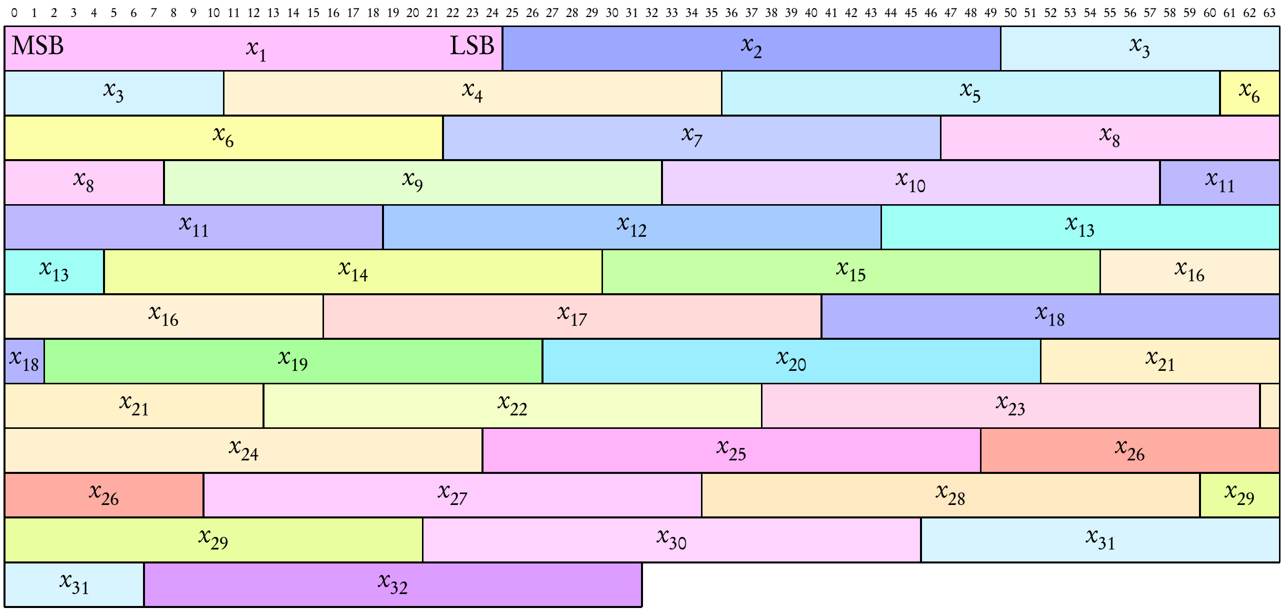


Figure 1. The bit-layout of "minimal representation" of a Zcash-equihash solution for $n = 144$, $k = 5$.



In particular, the `solution` is a solution to a slightly modified version of the Equihash proof-of-work puzzle [2], with integer parameters n and k specified as input to `SolverFunction`. The layout of a `solution` depends on those parameters: it consists of 2^k bitstrings x_i, each of length $\frac{n}{k+1} + 1$, concatenated together. For example, see Figure 1.

To summarize, when we abstract away the asynchrony and threading concerns, `SolverFunction` is given n and k (parameters of the Equihash construction), and `input` (in the notation of [2], the seed I already concatenated with the nonce V, i.e., $I||V$), and outputs `solution` (in the notation of [2], $x_1||x_2||\ldots||x_{2^k}$).

1.2 Zcash-equihash

In this section, we outline the version of Equihash we need to solve, which we call Zcash-equihash, since it is distinct from what is presented in [2] in several ways (minor modifications, but any modification is significant here). We refer to the latter, in this section, as original-Equihash.

The list of conditions that define an original-Equihash solution according to [2] are listed in Figure 2. We now discuss the differences.

1.2.1 Hash function

Zcash-equihash's hash function, $H(I||V, x_i)$, is defined in terms of BLAKE2b [8], as follows. First, the personalization string is set to "ZcashPoW" followed by n and k in little-endian order [7]. The salt is set to `0`. Then, given an $I||V$ and an index x_i, instead of directly concatenating them as in original-Equihash, we first divide x_i by a constant (`IndicesPerHashOutput`$:= {}^{512}/_{n}$), yielding quotient x_i° and remainder x_i^*. Then we compute BLAKE2b$(I||V||x_i^\circ)$, and from the output take bits $n \cdot x_i^*$ through $n \cdot x_i^* + n - 1$, so that the resulting "hash output" is exactly n bits long.

We speculate that the rationale for this is to reduce the number of hash evaluations required to solve Zcash-equihash, since hashing is CPU-hard and Equihash aims to be primarily memory-hard.

1.2.2 Difficulty condition

In the context of Zcash, the difficulty condition is considered part of "proof of work" rather than part of Zcash-equihash [3]. Thus the difficulty condition is effectively absent from Zcash-equihash, along with its corresponding parameter, d.

Figure 2. The problem statement of ORIGINAL-EQUIHASH [2].

Given integer parameters n, k, d, and seed bytestring I, generate 160-bit nonce V and $\left(\frac{n}{k+1}+1\right)$-bit $x_1, x_2, \ldots, x_{2^k}$ such that:

- **Generalized birthday**:

$$H(I||V||x_1) \oplus H(I||V||x_2) \oplus \cdots \oplus H(I||V||x_{2^k}) = 0$$

- **Algorithm binding**:
  - Intermediate solutions:

$$\forall w, \ell \quad H(I||V||x_{w2^\ell+1}) \oplus \cdots \oplus H(I||V||x_{w2^\ell+2^\ell}) \quad \text{has } \tfrac{n\ell}{k+1} \text{ leading zeros}$$

  - Ordering:

$$\forall w, \ell \quad \left(x_{w2^\ell+1} \middle\| x_{w2^\ell+2} \middle\| \ldots \middle\| x_{w2^\ell+2^{\ell-1}}\right) < \left(x_{w2^\ell+2^{\ell-1}+1} \middle\| x_{w2^\ell+w^{\ell-1}+2} \middle\| \ldots \middle\| x_{w2^\ell+2^\ell}\right)$$

- **Difficulty**:

$$H(I||V||x_1||x_2||\ldots||x_{2^k}) \quad \text{has } d \text{ leading zeros}$$

1.2.3 NONCE

ZCASH-EQUIHASH uses a 256-bit nonce instead of a 160-bit nonce. More importantly, in ZCASH-EQUIHASH we are *provided* the nonce, already concatenated into the input, instead of being asked for a nonce as part of the output. Generating the nonce is part of "proof of work" but not part of ZCASH-EQUIHASH.

1.3 OUR PROBLEM STATEMENT

1.3.1 PARAMETERS

We are given integer parameters n and k satisfying the following conditions:

- **Positivity**: $n > 0$ and $k > 0$.
- **Hash chunk is bytestring**: n is divisible by 8, so we can take an integer number of bytes of BLAKE2b output for each index.
- **Index is bitstring**: n is divisible by $k+1$, so that the x_i, which are $\frac{n}{k+1}+1$ bits long, are integer-length bitstrings.
- **Solution is bytestring**: $k \geq 3$, so that $2^k \cdot \left(\frac{n}{k+1}+1\right)$ is divisible by 8.
- **Index fits in dword**: $\frac{n}{k+1}+1 < 32$, so we can assume each index fits in a 32-bit dword.
- **Hash chunk fits in yword**: $n < 256$, so we can assume each hash chunk fits in a 256-bit `ymm` register on a modern x64 CPU.

1.3.2 INPUT

Input consists of the parameters, and an opaque 140-byte block of memory M (corresponding to $I||V$ in ORIGINAL-EQUIHASH).

1.3.3 OUTPUT

Output consists of a block of memory of length $\frac{2^k}{8} \cdot \left(\frac{n}{k+1}+1\right)$ bytes, containing bitstrings $x_0, x_1, \ldots, x_{2^k-1}$ each of length $\frac{n}{k+1}+1$ bits, in "minimal representation" as shown in Figure 1. The x_i satisfy the following properties (with H defined as in section 1.2.1):

- **Intermediate solutions**:

$$\forall \ell \in (0,k) \quad \forall w \in \left[0, 2^{k-\ell}\right) \qquad H(M, x_{w2^\ell}) \oplus \cdots \oplus H(M, x_{(w+1)2^\ell-1}) \quad \text{has } \tfrac{n\ell}{k+1} \text{ leading zeros}$$

- **Generalized birthday**:

$$H(M,x_0) \oplus \cdots \oplus H(M,x_{2^k-1}) = 0 \quad \text{(i.e., has } \tfrac{n(k+1)}{k+1} \text{ leading zeros)}$$

- **Ordering**:

$$\forall \ell \in (0,k) \quad \forall w \in \left[0, 2^{k-\ell}\right) \qquad \left(x_{w2^\ell} \middle\| \cdots \middle\| x_{w2^\ell+2^{\ell-1}-1}\right) < \left(x_{w2^\ell+2^{\ell-1}} \middle\| \cdots \middle\| x_{(w+1)2^\ell-1}\right)$$

2 Our Algorithm

2.1 Overview

As intended, the output conditions suggest a natural outline for a solution algorithm:

1. Compute all $H(M,i)$ (for $i \in \left[0, 2^{\frac{n}{k+1}+1}\right)$), and note collisions on the first $\frac{n}{k+1}$ bits
2. Set $j := 1$
3. Peform all pairwise XORs on pairs which share the same first $j\frac{n}{k+1}$ bits, and note collisions on the first $(j+1)\frac{n}{k+1}$ bits.
4. Increment j. If $j < k$, repeat from the previous step.
5. Find a collision in the final XOR results.
6. Trace back the solution tree and make swaps as necessary to satisfy the ordering condition.

2.2 Probabilistic analysis

References

[1] **Zcash Open Source Miner Challenge**: *Official Rules*, Oct. 2016, URL: https://zcashminers.org/rules (cited on p. 1).

[2] Alex **Birukov** and Dmitry **Khovratovich**: "Equihash: asymmetric proof-of-work based on the generalized birthday problem," *Network and Distributed System Security Symposium*, NDSS'16, (San Diego, California), Internet Society, Feb. 21–24, 2016, URL: https://www.internetsociety.org/sites/default/files/blogs-media/equihash-asymmetric-proof-of-work-based-generalized-birthday-problem.pdf (cited on pp. 1, 3, 4).

[3] The **Zcash developers**: `zcash`, Aug. 2016, URL: https://github.com/zcash/zcash, in particular, the functions `CheckEquihashSolution`, `IsValidSolution`, and the definition of `CBlockHeader` (cited on pp. 1, 3).

[4] **aabc**: `equihash-zcash-c`, Oct. 2016, URL: https://github.com/aabc/equihash-zcash-c (cited on p. 1).

[5] **xenoncat**: `equihash-xenon`, Oct. 2016, URL: https://github.com/xenoncat/equihash-xenon (cited on p. 1).

[6] David **Jaenson**: `equihash`, Oct. 2016, URL: https://github.com/davidjaenson/equihash (cited on p. 1).

[7] John **Tromp**: `equihash`, Oct. 2016, URL: https://github.com/tromp/equihash (cited on pp. 1, 3).

[8] Markku-Juhani **Saarinen** and Jean-Philippe **Aumasson**: *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*, RFC 7693 (Informational), Internet Engineering Task Force, Nov. 2015, URL: http://www.ietf.org/rfc/rfc7693.txt (cited on p. 3).