Abstract

The following is a proposal for Zcash to adopt an alternate proof-of-work algorithm - “ProgPoW” - tuned for commodity hardware in order to close the efficiency gap available to specialized ASICs. The grant proposal is not to fund the initial implementation, but rather to fund qualified independent review, adoption, and maintenance of the code.

The security of proof-of-work is built on a fair, randomized lottery where miners with similar resources have a similar chance of generating the next block.

For Zcash - a community based on widely distributed commodity hardware - specialized ASICs enable certain participants to gain a much greater chance of generating the next block, and undermine the distributed security.

ASIC-resistance is a misunderstood problem. FPGAs, GPUs and CPUs can themselves be considered ASICs. Any algorithm that executes on a commodity ASIC can have a specialized ASIC made for it; most existing algorithms provide opportunities that reduce power usage and cost. Thus, the proper question to ask when solving ASIC-resistance is “how much more efficient will a specialized ASIC be, in comparison with commodity hardware?”

This proposal presents an algorithm that is tuned for commodity GPUs where there is minimal opportunity for ASIC specialization. This prevents specialized ASICs without resorting to a game of whack-a-mole where the network changes algorithms every few months.

Motivation

Since proof-of-work will continue to underpin the security of the Zcash network, it is important that this consensus isn’t dominated by a single party in control of a large portion of the compute power.

The existence of the Z9 ASIC miner proves that the current Equihash algorithm allows significant efficiency gains from specialize hardware. The ~150mb of state in Equihash is large but possible on an ASIC. Furthermore, the binning, sorting, and comparing of bit strings could be implemented on an ASIC at extremely high speed.

Replacing the current Equihash with ProgPoW will allow Zcash to be resistant to domination by specialized hardware and avoid further PoW code forks.

Technical Approach

ProgPoW comprises:

- A memory hard component that is latency tolerant, and has high throughput and high capacity requirements.
- A GPU optimized math component that is highly programmatically variable (based on a block header), while saturting all GPU architecture (register files, cache, math units, logic units). This component is balanced between popular GPU architectures, so one architecture isn't dominant.

Prog-PoW utilizes almost all parts of a commodity GPU, excluding:

- The graphics pipeline (displays, geometry engines, texturing, etc);
- Floating point math.

Making use of either of these would have significant portability issues between commodity hardware vendors and across programming languages.

Since the GPU is almost fully utilized, there’s little opportunity for specialized ASICs to gain efficiency. Removing both the graphics pipeline and floating point math could provide up to 1.2x gains in efficiency, compared to the 2x gains possible in Dagger Hashimoto, 50x gains possible for CryptoNight, and ~100x gains possible in Equihash.

Prog-PoW preserves the 256-bit nonce size for Zcash and thus does not require any changes to light clients. However, it entirely replaces the rest of the Equihash PoW algorithm.

For the memory hard component, Prog-PoW follows the same general structure as Dagger Hashimoto. Dagger Hashimoto was selected as a base due to its simple algorithm. Dagger Hashimoto has also withstood a significant amount of real-world “battle testing.” Starting from a simple algorithm makes the changes easy to understand and easy to assess for strengths and weakness. Unnecessary complexity tends to imply a larger attack surface.

The name of the algorithm comes from the fact that the inner loop between global memory accesses is a randomly generated program based on the block number. The random program is designed to both run efficiently on commodity GPUs and also cover most of the GPU's functionality. The random program sequence prevents the creation of a fixed pipeline implementation as seen in a specialized ASIC. The access zisee has also been tweaked to match contemporary GPUs.

The algorithm has five main changes from Dagger Hashimoto, each tuned for commodity GPUs while minimizing the possible advantage of a specialized ASIC. In contrast to Dagger Hashimoto, the changes detailed below make Prog-PoW dependent on the core compute capabilities in addition to memory bandwidth and size.

Changes keccak to blake2s.

Zcash already uses BLAKE2 in various locations, so it makes sense to continue using BLAKE2. GPUs are natively 32-bit architectures so blake2s is used. Both blake2b and blake2s provide the same security, but are tuned for 64-bit and 32-bit architectures respectively. Blake2s runs roughly twice as fast on a 32-bit architecture as blake2b.

Increases mix state.

A significant part of a GPU's area, power, and complexity is the large register file. A large mix state ensures that a specialized ASIC would need to implement similar state storage, limiting any advantage.

Adds a random sequence of math in the main loop.

The random math changes every 50 blocks to amortize compilation overhead. Having a random sequence of math that reads and writes random locations within the state ensures that the ASIC executing the algorithm is fully programmable. There is no possibility to create an ASIC with a fixed pipeline that is much faster or lower power.

Adds reads from a small, low-latency cache that supports random addresses.

Another significant part of a GPU's area, power, and complexity is the memory hierarchy. Adding cached reads makes use of this hierarchy and ensures that a specialized ASIC also implements a similar hierarchy, preventing power or area savings.

Increases the DRAM read from 128 bytes to 256 bytes.

The DRAM read from the DAG is the same as Dagger Hashimoto's, but with the size increased to 256 bytes. This better matches the workloads seen on commodity GPUs, *preventing a specialized ASIC from being able to gain performance by optimizing the memory controller for abnormally small accesses.*

The DAG file is generated according to traditional Dagger Hashimoto specifications, with an additional `ProgPoW_SIZE_CACHE` bytes generated that will be cached in the L1.

Prog-PoW can be tuned using the following parameters. The proposed settings have been tuned for a range of existing, commodity GPUs:

- `ProgPoW_LANES`: The number of parallel lanes that coordinate to calculate a single hash instance; default is `32`.
- `ProgPoW_REGS`: The register file usage size; default is `16`.
- `ProgPoW_CACHE_BYTES`: The size of the cache; default is `16 x 1024`.
- `ProgPoW_CNT_MEM`: The number of frame buffer accesses, defined as the outer loop of the algorithm; default is `64` (same as Dagger Hashimoto).
- `ProgPoW_CNT_CACHE`: The number of cache accesses per loop; default is `8`.
- `ProgPoW_CNT_MATH`: The number of math operations per loop; default is `8`.

Prog-PoW uses FNV1a for merging data. The existing Dagger Hashimoto uses FNV1 for merging, but FNV1a provides better distribution properties.

Prog-PoW uses KISS99 for random number generation. This is the simplest (fewest instruction) random generator that passes the TestU01 statistical test suite. A more complex random number generator like Mersenne Twister can be efficiently implemented on a specialized ASIC, providing an opportunity for efficiency gains.

```
uint32_t fnv1a(uint32_t &h, uint32_t d)
{
   return h = (h ^ d) * 0x1000193;
}

typedef struct {
   uint32_t z, w, jsr, jcong;
} kiss99_t;

// KISS99 is simple, fast, and passes the TestU01 suite
// https://en.wikipedia.org/wiki/KISS_(algorithm)
// http://www.cse.yorku.ca/~oz/marsaglia-rng.html
```

```
uint32_t kiss99(kiss99_t &st)
{
    uint32_t znew = (st.z = 36969 * (st.z & 65535) + (st.z >> 16));
    uint32_t wnew = (st.w = 18000 * (st.w & 65535) + (st.w >> 16));
    uint32_t MWC = ((znew << 16) + wnew);
    uint32_t SHR3 = (st.jsr ^= (st.jsr << 17), st.jsr ^= (st.jsr >> 13), st.jsr ^= (st.jsr <<
5));
    uint32_t CONG = (st.jcong = 69069 * st.jcong + 1234567);
    return ((MWC^CONG) + SHR3);
}
```

The `LANES*REGS` of mix data is initialized from the hash's seed.

```
void fill_mix(
    uint64_t hash_seed,
    uint32_t lane_id,
    uint32_t mix[ProgPoW_REGS]
)
{
    // Use FNV to expand the per-warp seed to per-lane
    // Use KISS to expand the per-lane seed to fill mix
    uint32_t fnv_hash = 0x811c9dc5;
    kiss99_t st;
    st.z = fnv1a(fnv_hash, seed);
    st.w = fnv1a(fnv_hash, seed >> 32);
    st.jsr = fnv1a(fnv_hash, lane_id);
    st.jcong = fnv1a(fnv_hash, lane_id);
    for (int i = 0; i < ProgPoW_REGS; i++)
            mix[i] = kiss99(st);
}
```

The main search algorithm uses blake2s to generate a seed, expands the seed, does a sequence of loads and random math on the mix data, and then compresses the result into a final blake2s for target comparison.

```
bool ProgPoW_search(
    const uint64_t prog_seed,
    const uint256_t nonce,
    const uint256_t header,
    const uint64_t target,
    const uint64_t *g_dag, // gigabyte DAG located in framebuffer
    const uint64_t *c_dag  // kilobyte DAG located in l1 cache
)
{
    uint32_t mix[ProgPoW_LANES][ProgPoW_REGS];
    uint32_t result[4];
    for (int i = 0; i < 4; i++)
        result[i] = 0;

    // blake2s(header..nonce)
    uint64_t seed = blake2s(header, nonce, result);

    // initialize mix for all lanes
    for (int l = 0; l < ProgPoW_LANES; l++)
        fill_mix(seed, l, mix);

    // execute the randomly generated inner loop
    for (int i = 0; i < ProgPoW_CNT_MEM; i++)
        ProgPoWLoop(prog_seed, i, mix, g_dag, c_dag);

    // Reduce mix data to a single per-lane result
    uint32_t lane_hash[ProgPoW_LANES];
    for (int l = 0; l < ProgPoW_LANES; l++)
    {
        lane_hash[l] = 0x811c9dc5
        for (int i = 0; i < ProgPoW_REGS; i++)
            fnv1a(lane_hash[l], mix[l][i]);
    }
    // Reduce all lanes to a single 128-bit result
    for (int i = 0; i < 4; i++)
        result[i] = 0x811c9dc5;
    for (int l = 0; l < ProgPoW_LANES; l++)
        fnv1a(result[l%4], lane_hash[l])

    // blake2s(header .. blake2s(header..nonce) .. result);
    return (blake2s(header, seed, result) <= target);
}
```

The inner loop uses FNV and KISS99 to generate a random sequence from the `prog_seed`. This random sequence determines which mix state is accessed and what random math is performed. Since the `prog_seed` changes relatively infrequently it is expected that `ProgPoWLoop` will be compiled while mining instead of interpreted on the fly.

```
kiss99_t ProgPoWInit(uint64_t prog_seed, int mix_seq[ProgPoW_REGS])
{
    kiss99_t prog_rnd;
    uint32_t fnv_hash = 0x811c9dc5;
    prog_rnd.z = fnv1a(fnv_hash, prog_seed);
    prog_rnd.w = fnv1a(fnv_hash, prog_seed >> 32);
    prog_rnd.jsr = fnv1a(fnv_hash, prog_seed);
    prog_rnd.jcong = fnv1a(fnv_hash, prog_seed >> 32);
    // Create a random sequence of mix destinations for merge()
    // guaranteeing every location is touched once
    // Uses Fisher–Yates shuffle
    for (int i = 0; i < ProgPoW_REGS; i++)
        mix_seq[i] = i;
    for (int i = ProgPoW_REGS - 1; i > 0; i--)
    {
        int j = kiss99(prog_rnd) % (i + 1);
        swap(mix_seq[i], mix_seq[j]);
    }
    return prog_rnd;
}
```

The math operations that merge values into the mix data are ones chosen to maintain entropy.

```
// Merge new data from b into the value in a
// Assuming A has high entropy only do ops that retain entropy
// even if B is low entropy
// (IE don't do A&B)
void merge(uint32_t &a, uint32_t b, uint32_t r)
{
    switch (r % 4)
    {
    case 0: a = (a * 33) + b; break;
    case 1: a = (a ^ b) * 33; break;
    case 2: a = ROTL32(a, ((r >> 16) % 32)) ^ b; break;
    case 3: a = ROTR32(a, ((r >> 16) % 32)) ^ b; break;
    }
}
```

The math operations chosen for the random math are ones that are easy to implement in CUDA and OpenCL, the two main programming languages for commodity GPUs.

```
// Random math between two input values
uint32_t math(uint32_t a, uint32_t b, uint32_t r)
{
    switch (r % 11)
    {
    case 0: return a + b;
    case 1: return a * b;
    case 2: return mul_hi(a, b);
    case 3: return min(a, b);
    case 4: return ROTL32(a, b);
    case 5: return ROTR32(a, b);
    case 6: return a & b;
    case 7: return a | b;
    case 8: return a ^ b;
    case 9: return clz(a) + clz(b);
    case 10: return popcount(a) + popcount(b);
    }
}
```

The main loop:

```
// Helper to get the next value in the per-program random sequence
#define rnd()    (kiss99(prog_rnd))
// Helper to pick a random mix location
#define mix_src() (rnd() % ProgPoW_REGS)
// Helper to access the sequence of mix destinations
```

```
#define mix_dst() (mix_seq[(mix_seq_cnt++)%ProgPoW_REGS])

void ProgPoWLoop(
    const uint64_t prog_seed,
    const uint32_t loop,
    uint32_t mix[ProgPoW_LANES][ProgPoW_REGS],
    const uint64_t *g_dag,
    const uint32_t *c_dag)
{
    // All lanes share a base address for the global load
    // Global offset uses mix[0] to guarantee it depends on the load result
    uint32_t offset_g = mix[loop%ProgPoW_LANES][0] % DAG_SIZE;
    // Lanes can execute in parallel and will be convergent
    for (int l = 0; l < ProgPoW_LANES; l++)
    {
        // global load to sequential locations
        uint64_t data64 = g_dag[offset_g + l];

        // initialize the seed and mix destination sequence
        int mix_seq[ProgPoW_REGS];
        int mix_seq_cnt = 0;
        kiss99_t prog_rnd = ProgPoWInit(prog_seed, mix_seq);

        uint32_t offset, data32;
        int max_i = max(ProgPoW_CNT_CACHE, ProgPoW_CNT_MATH);
        for (int i = 0; i < max_i; i++)
        {
            if (i < ProgPoW_CNT_CACHE)
            {
                // Cached memory access
                // lanes access random location
                offset = mix[l][mix_src()] % ProgPoW_CACHE_WORDS;
                data32 = c_dag[offset];
                merge(mix[l][mix_dst()], data32, rnd());
            }
            if (i < ProgPoW_CNT_MATH)
            {
```

```
            // Random Math
            data32 = math(mix[l][mix_src()], mix[l][mix_src()], rnd());
            merge(mix[l][mix_dst()], data32, rnd());
         }
      }
      // Consume the global load data at the very end of the loop
      // Allows full latency hiding
      merge(mix[l][0], data64, rnd());
      merge(mix[l][mix_dst()], data64>>32, rnd());
   }
}
```

Team Background

The team that created ProgPoW has backgrounds in designing production ASICs; production hardware systems; and in optimizing miners, drivers, and firmware for mining performance.

Evaluation Plan

This algorithm should be evaluated against maximum potential ASIC efficiency gain metric described in the technical approach section.

As the algorithm nearly fully utilizes GPU functions in a natural way, a successful implementation will reflect relative GPU performance and power consumption that is similar to other gaming and graphics applications on the same architectures.

This PoW algorithm was tested against six different models from two different manufacturers. Selected models span two different chips and memory types from each manufacturer (Polaris20-GDDR5 and Vega10-HBM2 for AMD; GP104-GDDR5 and GP102-GDDR5X for NVIDIA). The average hashrate results are listed below. Additional tests are ongoing.

Model	Hashrate (MH/s)
RX580	9.4

Vega56	16.6
Vega64	18.7
GTX1070Ti	13.1
GTX1080	14.9
GTX1080Ti	21.8

Security Considerations

This algorithm is not backwards compatible with the existing Equihash implementation, and will require a fork for adoption. Furthermore, the network hashrate will change dramatically as the units of compute is re-based to this new algorithm.

The articles linked below go into more analysis on why an algorithm tied to commodity hardware provides the be incentives for decentralization and thus the best security

https://medium.com/p/da9f0512dad9

https://medium.com/p/d39078079186

Programmable hardware is the core of the question of how best to decentralize proof-of-work mining. However, it’s a balancing act of carefully administered parameters. Highly specialized hardware incentivizes protection of the value of the hardware over the value of the network. Highly generalized hardware is hard to design algorithms for, hard to optimize, and hard to protect from botnets.

Schedule

Sample code posted: 2018 May 1 (ProgPoW algorithm source)

Open review period: 2018 May 1 - 2018 Aug 15

The collaboration repository, current private at https://github.com/ifdefelse/zcash, will be made public on 2018 July 1.

First miner and client implementation for ZCash complete on 2018 Aug 1.

Review and testing complete on 2018 September 1.

Budget and Justification

The current team will remain unpaid. Beyond the current team, we will add 2 more people for a short duration to help accelerate and review the development process.

We are asking for funding for $50k for 2.5 person-months divided into:

1. $20k for one person-month on client implementation and review
2. $20k for one person-month on mining development and kernel review
3. $10k for two person-weeks as a revision buffer.

The team will disclose the address of the holding wallet in order to keep the process transparent. External persons to be funded and work progress will be documented on the associated Github repository. Expenditures will be paid 50% in advance of each work item above and 50% when the work is complete (committed to our repository and fully tested).

Work is still proceeding from the original team. If the original team is able to complete more of the work before Grant funds are first issued in July, we will reduce this funding request or return the unused funds to the Zcash Foundation.

Contact

Please contact ifdefelse@protonmail.com