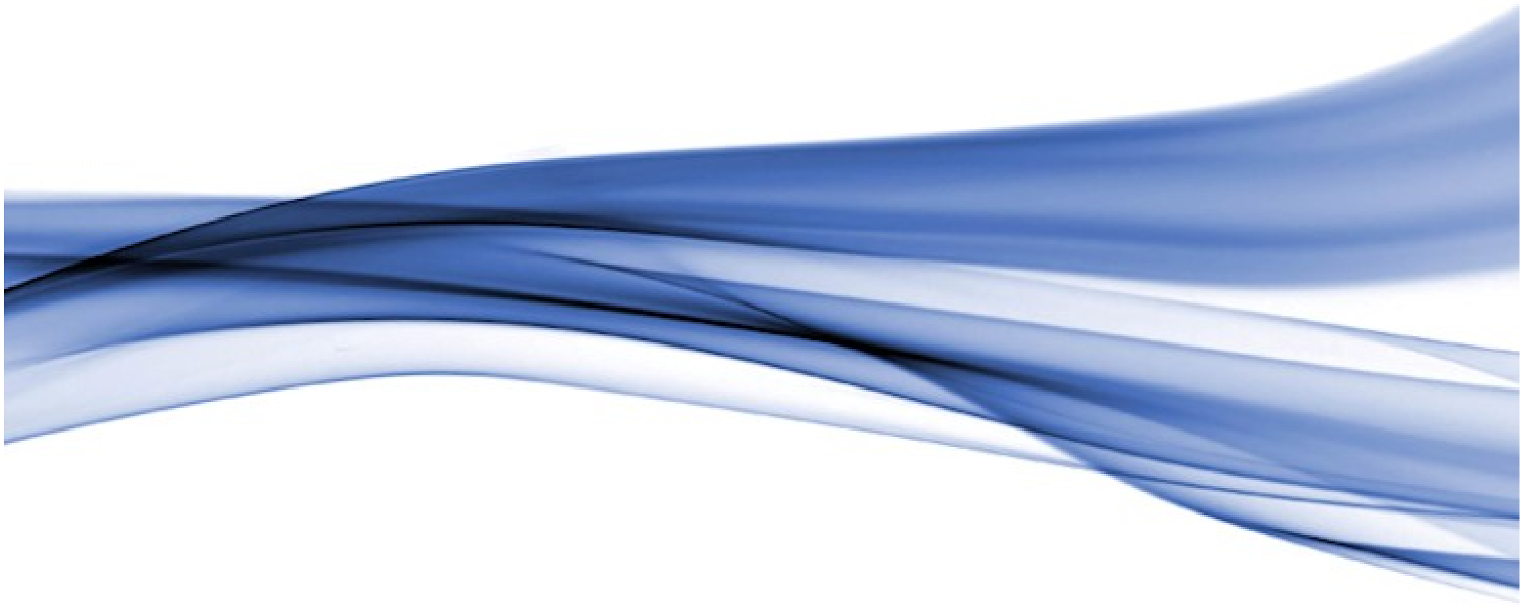
# Report:

VDF proof feasibility study



|  |  |
|---|---|
| For | **Justin Drake** |
| Of | **Ethereum Asia Pacific Limited<br>80 Robinson Road #08-01<br>Singapore<br>068898** |
| Reference | **P0137-R-004b** |
| Date | **October 12, 2018** |
| Prepared by | **Rupert Swarbrick** |

Argon Design Ltd.
St John's Innovation Centre
Cowley Road, Cambridge
CB4 0WS

Tel: +44 (0)1223 422355
Fax: +44 (0)1223 422356
www.argondesign.com

## Contents

# 1 Introduction

This study investigates the high-level design of an ASIC to compute VDF proofs for the next Ethereum protocol. The calculation needed for the proof is to compute $h^E$ where $E = \lfloor 2^\tau / B \rfloor$ in the group, $G$, of units in $\mathbb{Z}/N$ for some $h$, $\tau$, $N$ and $B$. For concreteness, elements of $G$ are about 2048 bits in size and $B$ is a prime, around 128 bits in size.

In order to put numbers in our calculations, recall that the VDF evaluator module we have proposed will run at around 300 MHz, taking three cycles to perform one group operation. As such, it might perform $10^8$ group operations per second. Ethereum described a VDF evaluation phase that should take about 85 minutes: five times as long as the 17 minute RANDAO phase. In that time, the VDF evaluator could perform $85 \times 60 \times 10^8 = 5.1 \times 10^{11}$ group operations. This number gives a practical value for $\tau$. The aim is for the prover to run in around 1% of the time of the evaluator, which gives it 51 seconds.

Unlike the VDF itself, the proof calculation can be parallelised. There are two schemes described in [Wes18] (hereafter, Algorithm 5 and Algorithm 6). We at Argon (mostly Peter de Rivaz) have suggested some further algorithms, which will be included in the report.

The report starts with notes about different algorithms and their costs (in number of group operations and storage). It then describes some practical considerations that the prover has to take into account: how exactly should Wesolowski's `get_block` be implemented? How much parallelism can we wring out of each algorithm? The last main section describes different multiplier architectures the prover might use. This section is incomplete: we believe that finding the definitive answer about the best approach to use needs more work.

The final section of the report is on a slightly different topic: can the prover ASIC detect when its being overclocked? This would avoid having to discard all but the fastest chips at the fab, at a cost of complication and slight supply chain risk.

## 2 Possible algorithmic improvements

### 2.1 Schemes based on Wesolowski's paper

The first scheme in [Wes18] (called Algorithm 5 there and reproduced as Algorithm 1 below) is a starting-point for all the other algorithms considered in the report. There are two parts to the computation: raising an array of elements, $y_b$, to the power $2^k$ and multiplying successive elements by pre-computed coefficients $C_i$. The number of group operations that must be performed is $lk2^k + \tau/k$.

```
for $b \in \{0, ..., 2^k - 1\}$ do
    $y_b \leftarrow 1$;
end
for $j \leftarrow l - 1$ to 0 do
    for $b \in \{0, ..., 2^k - 1\}$ do
        $y_b \leftarrow y_b^{2^k}$;
    end
    for $i \leftarrow \lfloor \tau/kl \rfloor$ to 0 do
        $b \leftarrow$ get_block($il + j$);
        $y_b \leftarrow y_b \cdot C_i$;
    end
end
$x \leftarrow 1$;
for $b \in \{0, ..., 2^k - 1\}$ do
    $x \leftarrow x \cdot y_b^b$;
end
```

**Algorithm 1:** Wesolowski's Algorithm 5

The improved scheme in the appendix of [Wes18] (called Algorithm 6 there) improves the complexity by replacing the $lk2^k$ operations with $l2^{k+1}$ operations. Peter at Argon came up with a simplified version of the scheme (with the same complexity), reproduced as Algorithm 2 below.

```
$x \leftarrow 1$;
for $j \leftarrow l - 1$ to 0 do
    for $b \in \{0, ..., 2^k - 1\}$ do
        $y_b \leftarrow 1$;
    end
    for $i \leftarrow \lfloor \tau/kl \rfloor$ to 0 do
        $b \leftarrow$ get_block($il + j$);
        $y_b \leftarrow y_b \cdot C_i$;
    end
    $t \leftarrow 1$;
    $x \leftarrow x^{2^k}$;
    for $b \leftarrow 2^k - 1$ to 1 do
        $t \leftarrow t \cdot y_b$;
        $x \leftarrow x \cdot t$;
    end
end
```

**Algorithm 2:** A simplified algorithm with complexity $l2^{k+1} + \tau/k$

To see why this is equivalent, imagine moving the calculation of $y_b^b$ into the loop and then switching around the order of evaluation so that we first raise the new factors in $y_b$ to the power $b$ and then square repeatedly to raise just one variable to the power $2^k$. Of course, once we've raised the new factors of each $y_b$ to the power $b$, we can multiply them together before raising to the power $2^k$. The last trick is to re-order the product that we're trying to compute as

$$x = \prod_{b=0}^{2^k-1} (y_b)^b = y_{N-1}\,(y_{N-1}\cdot y_{N-2})\,(y_{N-1}\cdot y_{N-2}\cdot y_{N-3})\cdots(y_{N-1}\cdot y_{N-2}\cdots y_1)$$

where $N = 2^k$. The values of the variable $t$ in the algorithm are the successive bracketed terms in the product and the values of $x$ are partial products, working from the left.

Note that a practical implementation wouldn't need to do the first loop (initialising $y_b$ to 1): instead it would keep $2^k$ single-bit flags denoting whether $y_b$ had been initialised. These would be used to select the input to be multiplied by $C_i$ and, if there was no block that hit $b$ for a particular $j$, to disable the update of $t$ later. Clearing these flags would require writing to $2^k$ bits, rather than $2^k$ group elements.

One potential problem with this algorithm is the tight loop to update $x$ and $t$. A practical implementation would have to split this up to allow parallelism: we expand on this in a later section.

A slight simplification of Algorithm 2 (not one that will make a significant difference to execution speed) is to remove the line that raises $x$ to the power $2^k$. We set $t$ to $x$ instead of one so that the $2^k-1$ multiplications of $x$ by $t$ calculate the power for free. This is shown below in Algorithm 3.

```
$x \leftarrow 1$;
for $j \leftarrow l-1$ to $0$ do
    for $b \in \{0,\ldots,2^k-1\}$ do
        $y_b \leftarrow 1$;
    end
    for $i \leftarrow \lfloor \tau/kl \rfloor$ to $0$ do
        $b \leftarrow$ get_block($il+j$);
        $y_b \leftarrow y_b \cdot C_i$;
    end
    $t \leftarrow x$;
    for $b \leftarrow 2^k-1$ to $1$ do
        $t \leftarrow t \cdot y_b$;
        $x \leftarrow x \cdot t$;
    end
end
```

**Algorithm 3:** Getting rid of a step in Algorithm 2

Algorithms 2 and 3 have the same complexity, with $\tau/k + l2^{k+1}$ group operations. Trying to speed things up further, note that accumulating into $y_b$ and then updating $t$ by multiplying by $y_b$ is slightly redundant: what if we could just multiply $t$ by the $C_i$'s instead, skipping the last multiplication? This would remove a multiplication and would avoid having to store all the $y_b$'s (lots of large group elements).

Of course, the problem is that then we must iterate over the values of $b$ in descending order. Finding the appropriate values of $i$ for a given $b$ essentially means inverting `get_block`. This is the approach in Algorithm 4, which inverts `get_block` by storing a list of indices for each $b$.

```
$x \leftarrow 1$;
for $j \leftarrow l-1$ to $0$ do
    for $b \in \{0,\dots,2^k-1\}$ do
        $A_b \leftarrow []$;
    end
    for $i \leftarrow \lfloor \tau/kl \rfloor$ to $0$ do
        $b \leftarrow$ get_block($il+j$);
        $A_b \leftarrow$ append($A_b$, $i$);
    end
    $t \leftarrow x$;
    for $b \leftarrow 2^k-1$ to $1$ do
        for $i \in A_b$ do
            $t \leftarrow t \cdot C_i$;
        end
        $x \leftarrow x \cdot t$;
    end
end
```

**Algorithm 4:** Avoiding the accumulation into separate $y_b$'s

The complexity in group operations is $\tau/k + l2^k$. This is because the line that updates $t$ runs once for each $i$ (albeit in a strange order), so this happens $\tau/kl$ times for each $j$ and $\tau/k$ times in total. The other group operation (updating $x$) runs $l2^k$ times.

Of course, this isn't the whole story: we imagine a chip with pipelining that can do one group operation per cycle, and the work to construct the $A_b$ lists isn't zero. However, we can pipeline this work, with a small part of the chip constructing each $A_b$ for the next $j$. The more significant cost is that this doubles the space requirement for the $A$ arrays.

## 2.2 Another approach: Lim and Lee

The basic algorithm in Wesolowski's paper seems to be a clever sparsified version of a standard algorithm from the literature, normally called "BGMW", after the authors Brickell, Gordon, McCurley and Wilson [BGMW92]. Another algorithm that was published at a similar time appears in Lim and Lee [LL94]. Interestingly, both of these algorithms were patented in the US; I think the latter expired in 2015. Bernstein points out in a preprint, [Ber02], that they are both special cases of a (complicated!) construction by Pippenger [Pip80].

The Lim and Lee algorithm computes $g^R$ for a group element $g$ and exponent $R$. To do so, it splits up the exponent, $R$, of length $n$ into $h$ large blocks of length $a = n/h$. Each of these is further subdivided into $v$ blocks of length $b = a/v$ (to keep notation clean, we hide all the floor and ceiling operations, pretending that everything divides exactly). Imagine stacking the large blocks on top of each other to form a rectangle with $h$ rows ($a$ bits wide each) of $v$ columns ($b$ bits wide each).

Firstly, values $g_0, \dots, g_{h-1}$ are computed where $g_i = (g_{i-1})^{2^a}$. These would be computed by the evaluator and passed to the prover. The other pre-computation is the values:

$$G[0][i] = g_{h-1}^{e_{h-1}} g_{h-2}^{e_{h-2}} \cdots g_1^{e_1} g_0^{e_0}$$

for $i = 0, \dots, 2^h$ where the sequence of $e$'s is the binary expansion of $i$, followed by

$$G[j][i] = (G[j-1][i])^{2^b} = (G[0][i])^{2^{jb}}$$

for $j = 1, \dots, v$ and $i$ as before.

With these in hand, the exponentiation can be written

$$g^R = \prod_{k=0}^{b-1} \left( \prod_{j=0}^{v-1} G[j][I_{j,k}] \right)^{2^k}$$

where $I_{j,k} < 2^h$ is the index from reading off the vertical row of bits at the $k$'th bit of the $j$'th column in the rectangle. As usual, the exponentiation can be evaluated efficiently by nested squaring. The resulting algorithm is shown as Algorithm 5.

```
$z \leftarrow 1$;
for $k \leftarrow b-1$ to 0 do
    $z \leftarrow z^2$;
    for $j \leftarrow v-1$ to 0 do
        $z \leftarrow z \cdot G[j][I_{j,k}]$;
    end
end
```

**Algorithm 5:** Lim and Lee's algorithm

The only significant storage is for the pre-computed $G[j][i]$ results. There are $v2^h$ of them. The number of operations for the algorithm (not including pre-computation) is $a+b$, which is more helpfully written $(1+1/v)n/h$.

At first, the pre-computation looks challenging: given just the $g_m$ results from the evaluator, a naïve algorithm to calculate the $G[j][i]$ looks like it might take $h$ multiplications to compute each $G[0][i]$, then $bv = a$ squarings to compute each corresponding $G[j][i]$. In total, this gives $(h+a)2^h$ group operations. Although this can be done incrementally while the evaluator is at work, this is still far too much work: the evaluator only needs to do $n = ah$ group operations and, while this calculation is more parallelisable, for most $h$ it will be significantly more work.

However, this calculation is pessimistic. Instead of just passing powers $g^{2^{ma}}$, the evaluator could pass all powers $g^{2^{mb}}$ (these correspond to the small chunks of $R$). Instead of computing powers of $G[0][i]$, when the prover gets $g^{2^u}$ where $u = va + wb$, this is $(g_v)^{2^{wb}}$ and the prover can just multiply each $G[w][i]$ by the value received if bit $w$ is set in $i$. This takes $2^{h-1}$ multiplications for each checkpoint.

# 3 Practical considerations for implementing the prover

## 3.1 Getting blocks of bits from the division

In both algorithms from Wesolowski's paper and in all algorithms that we have come up with, there is the basic structure shown in Algorithm 6. The `get_block` function gets $k$ bits of $\lfloor 2^\tau/B \rfloor$, starting at the given bit index.

```
for $j \leftarrow l-1$ to 0 do
    ...;
    for $i \leftarrow \lfloor \tau/kl \rfloor$ to 0 do
        $b \leftarrow$ get_block($il+j$);
        ...;
    end
    ...;
end
```

**Algorithm 6:** Calls to `get_block`

To implement `get_block`, an implementation can use the Euclidean algorithm, which is basically the same as doing long division in base $2^k$. It would start by skipping a few bits to get to position $\tau/k + j$, then read $k$ bits and skip $k(l-1)$ bits each iteration. Since $l$ is very large (of the order of $10^6$) in all the implementations we consider, this can't work by running through the skipped bits with the Euclidean algorithm: the chip would spend all its time skipping bits through the division.

However, skipping a block is the same as multiplying by $2^k$ and then taking the remainder modulo B. This means we can skip all the blocks we need by multiplying by $2^{k(l-1)} \bmod B$ and taking the remainder. Since this number can be computed with $O(\log(kl))$ operations at the start of the computation, this approach is much more efficient.

This multiplication still isn't fast enough to do as we go, though: all the algorithms we consider just do one other operation per block, so the latency of the modulo multiplication would be too much. To get around this, we could store an array of remainders on-chip, one for each $i$. Each such remainder is 128 bits in size, taking the same storage as $\tau/(16kl)$ group elements. Computing the initial values for the remainder takes $O(\tau/kl)$ operations. Since this is much less than $O(\tau/k)$ for all implementations that we consider, we'll ignore this cost when calculating run times below.

## 3.2 Parallelism when multiplying by pre-computed results

The amount of work for each candidate algorithm has the form $\tau/k + f(k,l)$ operations where $f$ depends on the algorithm. Assuming that $f(k,l)$ isn't too large, the division by $k$ implies a factor of $k$ speedup. However, the largest possible values of $k$ are of the order of ten so this isn't enough for the factor 100 speedup required. To get the other factor of ten, we need to find scope for parallelism. This can come from pipelining large multipliers and increasing the clock speed and/or from using more than one multiplier.

In all the algorithms before Algorithm 4, the $\tau/k$ term in the complexity comes from multiplying some $y_b$ in place by $C_i$. While doing so, successive iterations of the loop can go in parallel if each $i$ corresponds to a different value of $b$.

To calculate the cost of stalls, note that there will usually be $p$ pipeline stages, each calculating a result for a different $b$. Suppose these values of $b$ are numbered $b_1, \ldots, b_p$. If a new input has $b = b_p$, the machine must stall for one cycle while the final pipeline stage clears. If it has $b = b_1$, the machine must stall for $p$ cycles.

Since the pipeline entries are distinct, the probability that $b$ matches just stage $i$ is $2^{-k}$ for all $i$. As such, the expected number of stall cycles is

$$\mathbb{E}(\text{stalls}) = 2^{-k}(p + (p-1) + \cdots + 1) = p(p+1)2^{-(k+1)} \tag{1}$$

This decreases with $k$ and increases with $p$ and is large enough for values of interest that collisions might defeat our parallelism. For example, if the pipeline to load a value $y_b$, multiply by $C_i$ and store the result again took 10 cycles, we would have $p = 10$ even with only one multiplier. Even with a large value of $k$ like 10, Equation 1 shows a slowdown of around 5%. With smaller values of $k$ the collision rate is higher still.

We can avoid this problem with a two-entry queue at the front of each multiplier. This would hold two pairs $(i, b_i)$. With a pair of comparators for each pipeline stage, the multiplier could make sure it fetched an entry with a $b$ that wasn't currently in flight whenever one was available. This is analogous to a reservation station in an out-of-order processor. Calculating the stall count symbolically is fiddly, but a quick simulation shows a slowdown of just 0.17% with $p = 10$, $k = 10$.

Lowering $k$ or increasing $p$ does increase the stall rate again, although this effect can be softened by using a larger buffer. For example, suppose there are two multipliers with pipeline depth 10 processing half of the $b$'s each. This corresponds to decreasing $k$ by one. With a two-entry buffer, the stall rate is about 0.5%. Increasing to three entries drops it to 0.025%.

In a later section, we explore the multiplier architecture for the chip. It seems likely that we'll have to tolerate a latency of not 10 cycles but several thousand. This means that algorithms that update $y_b$'s in place just won't work.

## 3.3 Parallelism in algorithms 2 and 3

As mentioned earlier, there is a problem with the second loop in Algorithm 2 because successive values of $t$ and $x$ depend on each other, which means a pipelined approach won't work without some changes.

Writing $N = 2^k$ as before, split the range $1, \ldots, N-1$ into two halves: $1, \ldots, N/2-1$ and $N/2, \ldots, N-1$. Suppose we accumulated $x$ and $t$ over each of the halves separately, ending up with $(x_{lo}, t_{lo}), (x_{hi}, t_{hi})$ for the low and high ranges, respectively. Then the product that we actually want to compute is given by $x = x_{hi} x_{lo} (t_{hi})^{N/2-1}$. To see this is correct, expand each of the accumulators as

$$\begin{aligned}
x_{hi} &= y_{N-1}(y_{N-1}y_{N-2})\cdots(y_{N-1}y_{N-2}\cdots y_{N/2}) \\
x_{lo} &= y_{N/2-1}(y_{N/2-1}y_{N/2-2})\cdots(y_{N/2-1}y_{N/2-2}\cdots y_1) \\
t_{hi} &= y_{N-1}y_{N-2}\cdots y_{N/2}
\end{aligned}$$

and note that we are multiplying by $t_{hi}$ for each factor in $x_{lo}$ in order to extend that factor all the way up to $y_{N-1}$, as it appears in the original formula.

Splitting the range in half would work if the pipeline depth was only two. For a pipeline of depth $p$, we can split the range into $p$ parts analogously, ending up with $(x_0, t_0), \ldots, (x_{p-1}, t_{p-1})$ working from low $b$ to high $b$. For $m$ between 0 and $p-1$, write $n_m$ for the number of terms accumulated into $x_m$ and $t_m$. This will be approximately $(2^k - 1)/p$, but giving it a name lets us write a clean formula even when $p$ doesn't divide $2^k - 1$ exactly. Combining the partial products at the end gives a product of the form

$$x' = (x_{p-1}x_{p-2}\cdots x_0)(t_{p-1})^{n_{p-2}+\cdots+n_0}(t_{p-2})^{n_{p-3}+\cdots+n_0}\cdots(t_1)^{n_0}$$

and the loop finishes with $x \leftarrow x \cdot x'$.

This trick works for Algorithm 3 as well: each $x_m$ should be initialised to the previous value of $x$.

## 3.4 Parallelism in Algorithm 4

The group operations in Algorithm 4 also occur in a tight loop, but these can also be pipelined. To achieve a pipeline of length $p$, we must accumulate $p-2$ values of $b$ at once. Write $w_b = \prod_{i \in A_b} C_i$. Then the pseudo-code accumulates $t_b = \prod_{b' \geq b} w_b$ into variable $t$ at iteration $b$. The pipelined code would have a schedule that accumulated into $p-2$ different $w_b$'s. When one was complete, it would be stored into a reorder buffer. Next in the schedule, the algorithm would update $x$ if necessary with the current $t$ and finally it would take a newly computed $w_b$ (in descending order of $b$) and update $t$ with it.

While this scheme achieves the parallelism needed, the $w_b$ that was accumulated is exactly the same as the $y_b$ accumulated in earlier algorithms, and the multiplication to update $t$ has come back! For large values of $p$, this also means quite a lot of (less parallelisable) work to recombine the results. However, this work can actually be done outside the outer loop. The total memory cost probably ends up higher than earlier algorithms (you have to store the arrays, $A$, and also $p$ accumulators). However, it is very parallelisable and can tolerate any $p$ that divides $2^k$.

## 3.5 Parallelism in Lim and Lee's algorithm

Lim and Lee's paper addresses parallelism explicitly (in section 5). The basic idea is to split up the inner loop (for $j = 0, ..., v-1$) across the available processors and then accumulate at the end. The accumulation at the end is the same as we'd do in Algorithm 4 (described in the previous section). This means that we'd need $p$ to divide $v$ which in practice might mean lots of storage (since it scaled as $v2^h$).

## 4 ASIC multiplication architectures

Building a digital circuit to do modular exponentiation fast seems to be a well-studied problem: it's needed for RSA, so any cryptography co-processor ends up doing this operation. Unlike the evaluation algorithm considered in Geza's report, the proof algorithm exposes lots of parallelism so we can optimise for throughput rather than latency.

There is a slew of papers about designing hardware to do Montgomery multiplication (or exponentiation). Unfortunately, most of them are quite hard to translate into numbers that will help us. There are two problems. Firstly, many of the papers concentrate on bit-lengths much less than 2048 (presumably because most people who encrypt stuff with RSA at the moment use shorter primes).

Secondly, almost all of the papers give experimental results from FPGA implementations. Converting from counts of LUTs and DSP blocks to gate counts is not really possible because one 6-input LUT might be used as an inverter, and the next as a general 6-bit lookup table. We might need to sketch out an actual implementation to get good estimates from these papers.

One recent reference with actual ASIC numbers is [KWL16]. This tabulates relative performance of different carry-save adder designs for 2048-bit group elements. Their numbers show that their multipliers will only work if we can tolerate vast parallelism: the designs have a latency on the order of 1700 cycles for 2048-bit group elements. When synthesised in the 90nm TSMC process, they claim a critical path delay of 4.4ns, which corresponds to a clock frequency of about 230MHz. The quoted area is about $1\text{mm}^2$. In a 16nm TSMC process, this might be a factor of 20 smaller[1], giving an area of about $0.05\text{mm}^2$. Estimating how the path delay would scale is harder, but we might hope for a factor of 2 or 3 frequency increase to maybe 500MHz.

Assuming a clock frequency of 500MHz and a size of $0.05\text{mm}^2$ for each multiplier, this approach yields about $6 \times 10^6\,\text{ops/sec/mm}^2$. This throughput is only marginally higher than the $5 \times 10^6\,\text{ops/sec/mm}^2$ implied by the estimates in Geza's report.

Another paper with gate numbers is [MMM04]. These report a multiplier that uses around $3.3 \times 10^5$ gates for 2048 bit group elements. This will be about $0.04\text{mm}^2$ in a 16nm process. They don't give a critical path delay for any technology, but it is very short so their design can probably run at 1GHz, taking one cycle per bit. With a bit length of 2048, this works out at $11.8 \times 10^6\,\text{ops/sec/mm}^2$. It might turn out that the critical path is so short that the design can be implemented doing two steps per cycle. This wouldn't really affect the throughput estimate, but it might not halve the clock speed, so the latency and thus the parallelism required would probably decrease.

These numbers should probably be interpreted as lower bounds: Montgomery multiplication can use a Karatsuba-style architecture to improve its asymptotic performance. It's not clear what is the minimum bit width where this starts to help, but we think that 2048 bits might be enough. Unfortunately, we couldn't find pre-existing papers with performance numbers.

[1] TSMC's 90nm libraries typically quote a gate density of around $4 \times 10^5\,\text{gates/mm}^2$. Their 28nm libraries are more variable, but have a density around $4 \times 10^6\,\text{gates/mm}^2$. Exact numbers for the 16nm libraries are even harder to find, but news articles quote a doubling in density from 28nm.

## 5 Capping the speed of an ASIC

Assuming that the high-level protocol will work by rewarding the fastest VDF evaluators, if the Ethereum foundation make ASICs available, they might want to stop users from overclocking them.

To do this, the ASIC needs to have some way to measure its clock frequency and stop working if the frequency is too high. A purely synchronous digital circuit has no notion of its clock frequency. To get such a notion, the circuit would need some sort of internal oscillator. This wouldn't need to be the actual frequency reference, but the chip could periodically compare that with the internal oscillator and disable itself if the external frequency reference appeared too fast.

We searched through the literature for information about on-chip oscillators and there seem to be two main approaches:

1. MEMS-referenced PLLs
2. Self-referenced CMOS oscillators

MEMS-referenced PLLs consist of a MEMS device that resonates at some fixed frequency, with the period divided down by PLL logic. This is a mature technology, but seems mostly to be sold as separate oscillator chips.

We don't believe that a MEMS-referenced PLL really makes sense for this application. While they might be integrated in a package, the MEMS resonators are actually produced as separate dies. See Figure 1 for a die photograph. One concern is that a sufficiently determined attacker might open a package and replace the MEMS resonator with a faster one. This seems difficult, but might be easier than building another ASIC. Also, it may not be possible to integrate the PLL circuitry on the same wafer as the main digital design, which makes the attack much easier (since now the attacker just needs to switch some bond wires and a resonator).

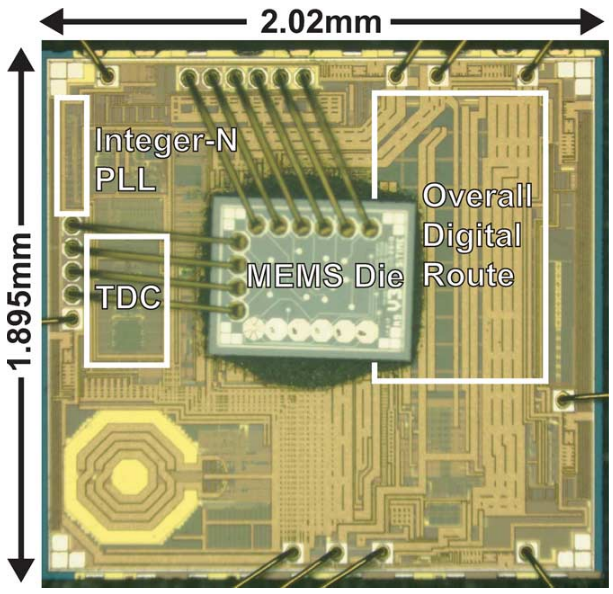


Figure 1: A MEMS-based resonator, showing the MEMS die itself with supporting circuitry (in 180nm CMOS).[PSL+12]

Early self-referenced CMOS oscillators worked by free-running an LC oscillator on the die (at over 1GHz) and then dividing the frequency down. With process trimming and inbuilt temperature compensation, this allows a stable 10-200MHz reference. A presentation from McCorquodale at the 2009 IEEE Radio Frequency IC symposium [KH09] describes the state of the art at that time.

More recent papers show that people are working on integrating the oscillators into digital processes. Unsurprisingly, there are no papers with actual implementations at the really recent process nodes. At larger sizes, there are papers like Huang and Wentzloff [HW14] (130 nm) who use an RC network and actually fabricated a test chip (see Figure 2 for a die photo). At more recent nodes, we could only find simulation results. For example, Lahiri and Tiwari describe a 30MHz ring based oscillator at 28nm [LT13], but only have SPICE simulations and no test chip. They claim 0.57% accuracy across the industrial temperature range: maybe something like this could work for Ethereum.

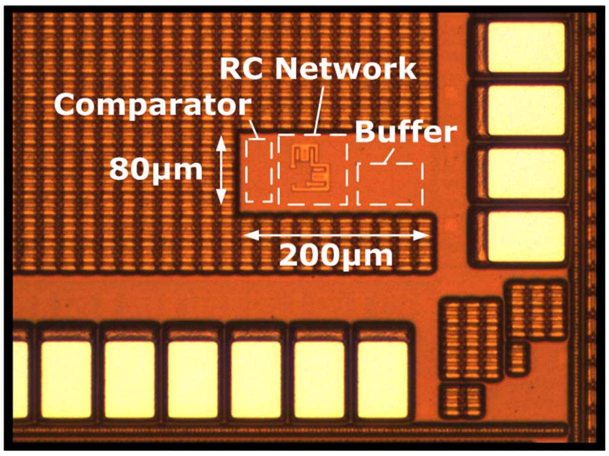


Figure 2: An RC-based resonator in 130nm CMOS.[HW14]

If Ethereum did intend to integrate something like this into their ASIC, the least risky approach would be to actually integrate several competing oscillator designs. The most stable one that worked could be enabled with OTP bits. Of course, this means that the ICs coming out of the fab wouldn't be hobbled and there would be a risk of unscrupulous parties getting hold of them before they were calibrated.

## 6 Conclusions and future work

This report has investigated the algorithmic complexity of several approaches to implementing a prover. There isn't a clear winner, because there are space/time trade-offs between them and the choice of the most appropriate algorithm depends on the size and latency of the multiplier used.

It seems like there's quite a bit more work to do in order to choose a multiplier. We believe that we could get some performance estimates from carefully studying existing papers, but accurate gate counts might even need a trial implementation. Maybe there would be mileage in reaching out to the papers' authors, asking for RTL.

Once we know more about the right multiplier, it should be easy to take the estimates in section 2 to work out the appropriate architecture. This ends up as an optimisation problem: Decide on the required throughput. For each algorithm, choose the tuning parameters (usually $k$ and $l$) that minimises the area for logic and storage combined. Choose the algorithm with the smallest value.

## References

[Ber02] Daniel J. Bernstein. Pippenger's exponentiation algorithm, 2002.

[BGMW92] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David B. Wilson. Fast exponentiation with precomputation. In *Advances in Cryptology – EUROCRYPT 92*, 1992.

[HW14] Kuo-Ken Huang and David D. Wentzloff. A 1.2-MHz 5.8-$\mu$W temperature-compensated relaxation oscillator in 130-nm CMOS. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(5):334–338, 2014.

[KH09] Waleed Khalil and Ahmed Helmy. Wsb: Current and future trends in frequency generation circuits. In *2009 IEEE Radio Frequency Integrated Circuits Symposium*, 6 2009.

[KWL16] Shiann-Rong Kuang, Kun-Yi Wu, and Ren-Yao Lu. Low-cost high-performance VLSI architecture for Montgomery modular multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):434–443, 2016.

[LL94] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology – CRYPTO 94*, 1994.

[LT13] Abhirup Lahiri and Anurag Tiwari. A 140$\mu$A 34ppm/$^{\circ}$c 30MHz clock oscillator in 28nm CMOS bulk process. In *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*, 1 2013.

[MMM04] Ciaran McIvor, Maire McLoone, and John V McCanny. Modified Montgomery modular multiplication and RSA exponentiation techniques. *IEE Proceedings-Computers and Digital Techniques*, 151(6):402–408, 2004.

[Pip80] Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, 1980.

[PSL$^+$12] Michael Perrott, Jim Salvia, Fred Lee, Aaron Partridge, Shouvik Mukherjee, Carl Arft, Jin-Tae Kim, Niveditha Arumugam, Pavan Gupta, Sassan Tabatabaei, Sudhakar Pamarti, Haechang Lee, and Fari Assaderaghi. A temperature-to-digital converter for a mems-based programmable oscillator with better than $\pm$0.5ppm frequency stability. In *2012 IEEE International Solid-State Circuits Conference*, 2 2012.

[Wes18] Benjamin Wesolowski. Efficient verifiable delay functions. Cryptology ePrint Archive, Report 2018/623, 2018. `https://eprint.iacr.org/2018/623`.