Cryptanalysis of Algebraic Verifiable Delay Functions

Alex Biryukov, Ben Fisch, Gottfried Herold, Dmitry Khovratovich, Gaëtan Leurent, Maria Naya Plasencia, Benjamin Wesolowski

Cryptanalysis of Algebraic Verifiable Delay Functions*

Alex Biryukov[1], Ben Fisch[2], Gottfried Herold[3], Dmitry Khovratovich[4], Gaëtan Leurent[5], María Naya-Plasencia[5], and Benjamin Wesolowski[6]

[1] University of Luxembourg, Esch-sur-Alzette, Luxembourg
[2] Yale University, New Haven, USA
[3] Ethereum Foundation, Bonn, Germany
[4] Ethereum Foundation, Luxembourg City, Luxembourg
[5] INRIA, Paris, France
[6] ENS de Lyon, CNRS, UMPA, UMR 5669, Lyon, France

Abstract. Verifiable Delay Functions (VDF) are a class of cryptographic primitives aiming to guarantee a minimum computation time, even for an adversary with massive parallel computational power. They are useful in blockchain protocols, and several practical candidates have been proposed based on exponentiation in a large finite field: Sloth++, Veedo, MinRoot. The underlying assumption of these constructions is that computing an exponentiation x^e requires at least $\log_2 e$ sequential multiplications.
In this work, we analyze the security of these algebraic VDF candidates. In particular, we show that the latency of exponentiation can be reduced using parallel computation, against the preliminary assumptions.

Keywords: Verifiable Delay Functions · MinRoot · Veedo · Sloth++ · cryptanalysis · smoothness

1 Introduction

A verifiable delay function (VDF) [11] is a function $f\colon X \to Y$ that on average cannot feasibly be evaluated much faster than some prescribed time T, even on a parallel machine, but every output $y = f(x)$ comes with a proof that allows anyone to quickly verify the output is correct. The first property makes it a delay function and the second makes it verifiable. In more detail, a VDF is a triple of algorithms:

- $\mathrm{Setup}(1^\lambda, T) \to pp$; returns public parameters pp given a security parameter λ and delay parameter T.
- $\mathrm{Eval}(pp, x) \to (y, \pi)$; returns $y \in Y$ given $x \in X$ and a proof π.
- $\mathrm{Verify}(pp, x, y, \pi) \to \{0, 1\}$; returns 1 if y is the correct output of input x.

The algorithms Setup and Verify are polynomial time and the algorithm Eval(pp, x), which is allowed to use up to poly(λ) parallel processors, runs in time at most $(1 + \varepsilon)T$ for some $0 < \varepsilon < 1$. Any parallel algorithm using at most poly(λ) processors and running in time less than T should not be able to compute $f(x) = y$ except with negl(λ) probability over random x. We refer to [11] for a more complete and formal definition.

VDFs have been most notably applied to removing bias in randomness beacons[7], which play an important role in consensus protocols, especially those used in modern *blockchains* [11,15,23,44]. The randomness may be used both to select block proposers unpredictably, or in the application layer, *e.g.* to simulate lotteries or more sophisticated protocols. Unbiasable randomness beacons can also be constructed from coin-tossing protocols [10], or from distributed key generation and verifiable random functions (VRFs) [30], but these constructions require a majority of parties participating in the protocol to be honest. In addition to requiring honest-majority assumptions, such protocols are communication-intensive and prone to delays in the case of a dynamic player set. In contrast, constructions of randomness beacons using VDFs are simple, communication efficient, and only require one participating party to be honest. VDFs have also been particularly important for constructing blockchains using proof-of-space as an alternative to proof-of-work[8], or more broadly achieving dynamic availability in consensus protocols without proof-of-work [20]. There are many other applications, including to proofs-of-replication [11,25], voting protocols [5], and preventing front-running of trades in decentralized exchanges [17]. See [37] for a survey of applications.

Any delay function can be made verifiable using generic SNARKs [11,36]. However, it is desirable to construct delay functions with specialized proof systems that result in a more efficient Eval algorithm than the generic construction (i.e., requiring fewer parallel processors and lower overall complexity).

There are two elegant and efficient constructions of VDFs based on repeated squaring in a group of unknown order [59,39], as well as constructions based on sequential compositions of isogenies on elliptic curves [19,46,4]. There are several candidates for groups of unknown order. One is the RSA group $\mathbb{Z}_N^*$ for $N = p \cdot q$ a product of two unknown safe primes, provided that the group is set up by a trusted party who discards p and q. The assumption that repeated squaring is sequential in this group was previously used in the construction of timelock puzzles [41], and over generic rings has been proven equivalent to the difficulty of factoring [42]. Multiparty computation can be used to generate a trusted RSA modulus [12,16,27], but this has been found to be challenging and error-prone to implement.[9] Two candidates that do not require a trusted setup are class groups of quadratic number fields and Jacobians of hyperelliptic curves [22]. Unfortunately, the security of VDFs in class groups or Jacobians is based on less

[7] https://a16zcrypto.com/posts/article/public-randomness-and-randomness-beacons/

[8] https://docs.chia.net/green-paper-abstract/

[9] https://medium.com/zengo/dogbyte-attack-playing-red-team-for-eth2-0-vdf-ea2b9b2152af

studied assumptions (*e.g.*, difficulty of finding low order elements in the class group of an imaginary quadratic number field). The security of VDFs based on isogenies is similarly based on relatively new assumptions.

In theory, a sequential composition of hash functions, when modeled as random oracles, is provably a delay function [35]. While this might suggest that the generic construction from SNARKs is the most robust direction for VDFs, in practice no concrete hash function behaves like a true random oracle. Moreover, the way this hash function is constructed can have a big impact on the efficiency of Eval, specifically the complexity of computing a SNARK for many interactions of the functions. The complexity of the SNARK is proportional to the complexity of verifying $f(x) = y$ directly, using an arithmetic circuit over a finite field. One direction that was suggested in [11] and explored further in [52,29] is to use an iterated permutation Π over $\mathbb{F}^n$ that has (a) low arithmetic complexity as a circuit over $\mathbb{F}$, and (b) Π^{-1} has even lower complexity than Π, which helps with the SNARK. One such candidate are permutations that use a round function $x^e \bmod p$ for $e \gg 2$ where the inverse operation is exponentiation by $a = e^{-1} \bmod p - 1$ for $a \ll e$. The assumption that $x^e \bmod p$ is sequential, *i.e.* that it cannot be evaluated substantially faster than repeated squaring on a reasonable number of parallel processors, originates in the Sloth system [33], a precursor to VDFs. This method is used in the VDF candidates Sloth++ [11], Veedo [52], and MinRoot [29].

In this work, we present parallel algorithms for evaluating $x^e \bmod p$ several factors faster than $\log_2 e$ steps, challenging this assumption that was widely used in constructions of VDFs, including one that was planned for use in Ethereum's consensus protocol and Filecoin, and for which an ASIC had already been developed [53,54]. As a response to our attacks, those plans have been put on hold [28]. We conclude by suggesting several alternative directions for VDFs that are not known to suffer from our attacks.

1.1 Algebraic VDF Proposals

We first describe concrete VDF proposals based on exponentiation in a finite group and their security claims.

Sloth++ [11]. The round function in the Sloth++ VDF candidate involves two interleaved permutations ρ and σ over $\mathbb{F}_{p^2}$ where $p \equiv 3 \bmod 4$. The function $\rho(\mathbf{x}) = \mathbf{x}^{(p^2+1)/4}$ returns a square root of $\mathbf{x} \in \mathbb{F}_{p^2}$ and has inverse $\rho^{-1}(\mathbf{y}) = \mathbf{y}^2$. The function $\sigma(\mathbf{x})$ interprets $\mathbf{x} = (x_1, x_2)$ as a vector over $\mathbb{F}_p^2$ (i.e., the coefficients of a degree two polynomial) and returns $\sigma(x_1, x_2) = (x_2 + c_1, x_1 + c_2)$ for some fixed constants $c_1, c_2 \in \mathbb{F}_p$.

Veedo [52]. The round function of Veedo operates on a pair of elements (u, v) in $\mathbb{F}_p$. It first applies a root operation to each element, then applies a linear layer defined by an MDS matrix.

Input: $x \in \mathbb{F}_{p^2}$
  for $0 \leq i < n_{\text{iterations}}$ **do**
    $x \leftarrow \sqrt{x}$
    $(x_1, x_2) \leftarrow x$ ▷ Map from $\mathbb{F}_{p^2}$ to $\mathbb{F}_p^2$
    $(x_1, x_2) \leftarrow (x_2 + c_1, x_1 + c_2)$
    $x \leftarrow (x_1, x_2)$ ▷ Map from $\mathbb{F}_p^2$ to $\mathbb{F}_{p^2}$
  return u, v

Algorithm 1. Sloth++ VDF.

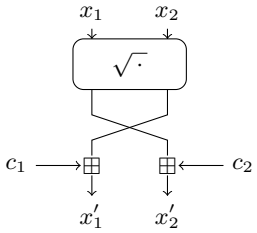

Fig. 1. Sloth++ round function.

Input: $u, v \in \mathbb{F}_p$
  for $0 \leq i < n_{\text{iterations}}$ **do**
    $(u, v) \leftarrow M \times (\sqrt[a]{u}, \sqrt[a]{u}) + (c_i, c'_i)$
  return u, v

Algorithm 2. Veedo VDF.

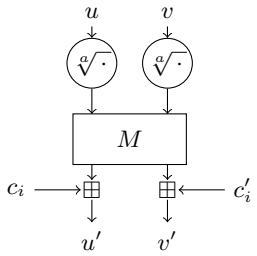

Fig. 2. Veedo round function.

MinRoot [29]. MinRoot is a VDF proposal that was designed to be used as a randomness beacon in the core layer of the Proof-of-Stake Ethereum protocol[10]. It iterates $n_{\text{iterations}}$ times a simple round function, which is concretely given as an a'th root for small a in a finite field defined by a prime number p such that $\gcd(a, p-1) = 1$, as shown in Algorithm 3. A proof of evaluation is supposed to be a SNARK proof, output by the Nova prover [31]. We denote the internal state of MinRoot as (u, v), and the round counter as i.

Input: $u, v \in \mathbb{F}_p$
  for $0 \leq i < n_{\text{iterations}}$ **do**
    $(u, v) \leftarrow (\sqrt[a]{u + v}, u + i)$
  return u, v

Algorithm 3. MinRoot VDF.

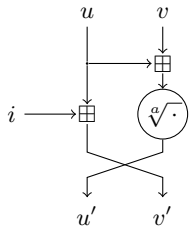

Fig. 3. MinRoot round function.

The concrete parameters proposed by the MinRoot designers are as follows:

- $p = 2^{254} + 2^{32} \cdot \texttt{0x224698fc094cf91b992d30ed} + 1$,
- $a = 5$,

[10] https://ethereum.github.io/consensus-specs/

– The number $n_{\text{iterations}}$ of iterations is typically in the order of 2^{40}.

MinRoot has concrete security claims and an ASIC implementation has been developed, therefore we explain in more detail its implementation properties, security claims, and functionality.

Currentlty, in the Proof-of-Stake Ethereum protocol every 32 blocks constitute an epoch, and the epoch block proposers contribute some entropy values $s_1, s_2, \ldots, s_{32}$ (omissions are possible). The values are passed to a classical hash function whose output is declared a *random beacon*, which determines proposers and signers for the next epochs. Clearly, the last of 32 proposers can influence the beacon value by trying different entropy values.

To mitigate this issue, VDF f would be applied to $(s_1, s_2, \ldots, s_{32})$. The calculation of f would run for the M next epochs, and the result would then determine the block proposers for the $M+1$-th epoch. Assuming that no parallel adversary could compute the VDF faster than by a factor of M and that each epoch has at least one honest proposer, there is no way for any malicious proposer, who learns the other contributions of a given epoch only during that epoch, to precompute f and find some contribution s_i of his own that would give a desired VDF output.

MinRoot Implementation. A standard implementation of MinRoot, such as the one by Supranational [53], computes the rounds iteratively. The root is the most expensive operation in the round function; in each round, it is computed using Fermat's little theorem (as $\sqrt[a]{x} = x^{1/a \bmod p-1}$), with a square and multiply algorithm. We define $e = 1/a \bmod p-1$ so that the round function is written as $\sqrt[a]{x} = x^e$. Recall that a is chosen such that $\gcd(a, p-1) = 1$.

Exponentiation to the power e with the square and multiply algorithm requires $\log_2(e) \approx \log_2(p)$ squarings. There are techniques to reduce the number of multiplications (such as the sliding-window method or addition chains), but in the context of a low latency implementation it is better to use a naive binary decomposition because the multiplications can be evaluated in parallel with the squarings. Therefore, the delay of one round is essentially $\log_2(p)$ squarings (254 squarings with the proposed parameters), and the delay of MinRoot is essentially $n_{\text{iterations}} \cdot \log_2(p)$ squarings (using two processors: one for the squarings and one for the multiplications).

In practice, Supranational built an optimized ASIC implementation of MinRoot, on a 12 nm node [53,55]. Their implementation requires 257 cycles per round, where each cycle essentially corresponds to a squaring and takes 0.9 ns (230 ns per round).

MinRoot security claims and their interpretation. The security claim of MinRoot is that it is a sequential function. Informally, this means that MinRoot cannot be computed faster by using some parallelism. There should be a lower bound to the delay required to evaluate the function, and the standard implementation with a delay of $n_{\text{iterations}} \log_2(p)$ squarings should be close to this lower bound (up to a small constant factor).

This implies at least two distinct assumptions:

1. The round function itself is a sequential function (*i.e.* the root cannot be computed faster using parallelism);
2. The iteration is sequential (*i.e.* there is no shortcut to evaluate $n_{\text{iterations}}$ rounds faster than by iterating them).

Formally, MinRoot is claimed to have 128 bits of VDF security, defined as follows: Given up to 2^{128} parallel processors, no adversary using these processors and spending up to 2^{128} MinRoot calls in precomputation, can compute MinRoot on any input from a challenge set faster than by a factor of 2.

One may notice that the security claim of MinRoot is somewhat vague. Concretely, it misses the following details, which became apparent at the time of third party analysis:

- What the reference implementation of MinRoot is and what timing it claims.
- What the computation model is, which might be needed to determine whether the adversary is faster than the legitimate computation.

For the first issue, we refer to the third-party implementation of MinRoot in hardware by Supranational [53]. They report latency benchmarks and chip sizes [55]. For the second issue we had to choose between existing models in the parallel computation analysis (cf. a survey in [43]) and some ad-hoc model. We resorted to the second option for two reasons:

- Our model matches nicely with the existing hardware implementation by Supranational.
- We assign realworld-inspired numbers to memory access and CPU access timings, which scale reasonably with the network size.

1.2 Notations and Assumptions

We focus on the latency of evaluating VDFs in parallel from an algorithmic point of view. In particular we evaluate the latency of the computation and neglect the latency of communication between the processors, and implementation issues such as large fan-in or large fan-out.

While the VDF we study naturally works with modular values in $\mathbb{F}_p$, in this paper, we sometimes consider the values as integers instead; we believe this should be clear depending on the context. The notation $x \bmod q$ refers to the integer in $\{0, 1, \ldots, q-1\}$ that is congruent to x. When taking logarithms, we default to base-2 (but usually write it out unless it does not matter, such as in $\mathcal{O}$-expressions).

Latency Assumptions. We consider the following results on low-latency arithmetic operations:

Integer addition: Addition of two n-bit integers has a latency of $\mathcal{O}(\log(n))$ using a carry look-ahead adder (*e.g.* the Brent-Kung adder [13]).
Addition of k n-bit integers has a latency of $\mathcal{O}(\log(k) + \log(n))$, using a tree of carry-save adders [24], followed by a carry look-ahead adder.

Integer multiplication: Multiplication of two n-bit integers has a latency of $\mathcal{O}(\log(n))$ using a tree of carry-save adders (*e.g.* a Wallace tree [57]) followed by a carry look-ahead adder.

Trial division: Trial division of an up to n-bit integer x by a hardwired constant c of at most $m < n$ bits can *e.g.* be performed by checking whether $x \cdot c^{-1} \bmod 2^{n+m}$ has at most n bits if c is odd. Since division by powers of two is essentially free and $c^{-1} \bmod 2^{n+m}$ can be precomputed, trial divisions by such constants have latency $\mathcal{O}(\log n)$.

Modular reduction: Modular reduction of a $2n$-bit value modulo an n-bit value has a latency of $\mathcal{O}(\log(n))$. For instance, the Barrett reduction [8] computes the reduction using integer division by a constant, which is computed using an integer multiplication.

The results above imply that modular addition Add, modular multiplication Mul, and modular reduction (this includes trial division) Mod in $\mathbb{F}_p$ all have a latency of $\mathcal{O}(\log(\log(p)))$. For simplicity, we assume that those operations have the same latency, and consider it as one unit of time (for practical purposes, this time unit is estimated to be 0.9 ns in an ASIC implementation, following the Supranational implementation of $\mathsf{MinRoot}$).

Adding up to $\log_2(p) \approx 256$ values (integers smaller than p or modular values in $\mathbb{F}_p$) also has latency $\mathcal{O}(\log(\log(p)))$; we assume this also corresponds to one unit. Trial division of 256-bit integers by hardwired constants cost essentially the same and we assume unit latency for this.

We also assume that a multiply-and-add operation ($x \cdot y + z$) has unit latency, because the addition term can be included in the tree of carry-save adders.

A table lookup in a table with k entries, denoted by $\mathsf{Table}(k)$, has a latency $\mathcal{O}(\log(k))$ when implemented as a combinatorial circuit. For small tables (up to $\log_2(p) \approx 256$ entries), we treat this as having unit latency. Larger tables must be stored in RAM, and have a larger latency. Assuming a DDR RAM with a latency of 5 ns, we consider that a memory access has a cost of 6 units of time.

When evaluating the number of processors required to implement an algorithm, we consider that one processor corresponds to a circuit of roughly the size of a modular multiplier. Adding up to $\log_2(p)$ values and table lookups with up to $\log_2(p)$ values are each assumed to require one processor.

These latency estimations are strongly dependent on the architecture used to implement a circuit, and are therefore somewhat arbitrary.[11] Nonetheless, they are useful to compare algorithms in a concrete setting, when an asymptotic complexity estimate is hard to derive. In particular, we evaluate all our algorithms in the context of a 256-bit prime, to match the parameters of $\mathsf{MinRoot}$, since we have ASIC implementation results available. The implementation from [53] uses algorithms for addition, multiplication, and modular reduction that roughly match these latency assumptions.

[11] The Supranational team mentioned in private communication that it should be possible to reduce the latency by a factor of up to 3 at the cost of longer and more expensive circuit scrutiny (*e.g.* better layout), using smaller technical process, larger lookup tables etc.

1.3 Our results

We study how to compute VDF functions in parallel in order to achieve a lower latency than the described standard implementations. Instead of looking for shortcuts when iterating a large number of rounds, we focus on the round function itself, and the most expensive operation, the root computation. We are able to provide some algorithms that achieve latency gains that clearly break the original security claims, considering our (reasonable) model. Despite breaking the security claims, it is still unclear whether our algorithms can be implemented in practice, as they require a large number of parallel processors. Even if the practicality of our algorithms might be debatable, they clearly show that computing a root $\sqrt[e]{x}$ in $\mathbb{F}_p$ via square and multipy is not necessarily sequential. Several previous works [32,11,52,29] assume that there is no parallel algorithm with lower latency than the square and multiply algorithm with latency $\log_2(p)$, but our results show that this assumption is wrong.

We explore several ideas to evaluate monomial functions in parallel in Section 2. Our best algorithm is a smoothness-based algorithm in Section 2.2 that uses the same basic ideas as a paper from Adleman and Kompella [2]. We propose various optimizations of this algorithm in Section 3, and prove a close connection to the discrete logarithm problem in Section 4. We demonstrate the impact of our methods on several VDF constructions in Section 5. In Section 6, we consider the related problem of evaluating low-degree functions in parallel. In Section 7 we briefly discuss practical issues to implement our algorithms. Finally, in conclusion we discuss possible tweaks of algebraic VDF functions to improve their security.

Table 1 shows a summary of our results, with concrete number corresponding to the MinRoot setting with a 256-bit prime.

Table 1. Summary of our attacks to compute $\sqrt[e]{x}$, with concrete speedups for a 256-bit prime (as in MinRoot).

Algo	T	$\#CPU$	M	speedup	Techniques
naive	256	1	0	1	Fast exponentiation
1	8	2^{128}	2^{128}	32	Baby-step, giant-step
2	6	$2^{54.5}$	0	42	Smoothness
3	13	2^{48}	$2^{59.5}$	20	Smoothness with medium-size factor
4	14	2^{40}	$2^{59.5}$	18	Smoothness with medium-size factor and pre-filter
5	21	2^{36}	2^{64}	12	Smoothness with special shape of p
6	54	2^{34}	2^{40}	4.7	Smoothness with rational reconstruction
7	13	2^{29}	2^{40}	20	Smoothness with parallel smoothness test
8	68	2^{25}	2^{40}	3.7	Smoothness with parallel rational reconstruction

2 Low-latency Evaluation of Power Functions

We explore in this section how to reduce the latency of root functions using their homomorphism property. We propose algorithms that make use of the fact that monomial functions f over $\mathbb{F}_p$ have the following properties:

- f is easy to precompute on any single input.
- $f(x \cdot y) = f(x) \cdot f(y)$, $\forall x, y \in \mathbb{F}_p$.

In the following, we consider roots $f(x) = \sqrt[a]{x} = x^{1/e \bmod p-1}$ in $\mathbb{F}_p$, where $\gcd(a, p-1) = 1$. We take $p \approx 2^{256}$ for concrete examples to match the MinRoot parameters.

The root is the most expensive operation in VDF constructions, and has a latency of $\log_2(p)$ squaring in the standard implementation. We show in this section two basic algorithms for evaluating it with smaller latency. These algorithms will be refined and improved in the next section.

In an actual VDF such as MinRoot, the round function is applied a large number ($n_{\text{iterations}} \approx 2^{40}$) of times. Due to that, we really care about *expected* (over a random choice of input) time/latency. For the probabilistic algorithms in this and the next sections, we therefore aim for a good, but not necessarily overwhelming success probability. See 7.1 for some further discussion.

2.1 A Baby-step Giant-step Approach

We first propose a simple algorithm using precomputation. The MinRoot submission document [29] also described a precomputation attack, but in our case we target the root operation instead of targeting the iterated function MinRoot. Additionally, we use a self-randomization property to make the attack successful for any given input instead of having a multi-target attack. Let $\alpha, \beta > 0$ be two parameters.

Precomputation. We precompute the following values:
- Compute $\sqrt[a]{i}$ for $i \leq \alpha\sqrt{p}$.
- Compute r^a, r^{-1} for a set of $\beta\lfloor\sqrt{p}\rfloor$ random values r.

Online phase. Given a challenge x, do the following in parallel with $\beta\lfloor\sqrt{p}\rfloor$ processors:
1. Pick one of the randomly generated r.
2. Compute $y = x \cdot r^a \bmod p$.
3. If $y \leq \alpha\sqrt{p}$, then return $\sqrt[a]{y} \cdot r^{-1}$.

This algorithm is similar to the baby step-giant step algorithm to compute discrete logarithms. Due to the birthday paradox, its probability of success is approximately $1 - \exp(-\alpha\beta)$, so one can fix reasonably small values $\alpha, \beta \in \mathcal{O}(1)$. The method has a latency of two multiplications and one table lookup. This implies $T = 2+6 = 8$ time units using the assumptions of Section 1.2, using $\mathcal{O}(\sqrt{p})$ precomputations, and $\beta\sqrt{p} = \mathcal{O}(\sqrt{p})$ parallel processors, versus the $T = 256$ of the sequential computation.

For MinRoot, with $p \approx 2^{256}$, this requires approximately 2^{128} precomputations and parallel processors. Since MinRoot claims 128 bits of security, this algorithm has a very close complexity to the claim, and shows that it is at best tight.

In practice, each of the 2^{128} processors will use a different r that can be hard-coded in the processor, together with r^a and r^{-1}. The algorithm requires a memory of size $\sqrt{p}$. However, note that for uniformly random input, only $\alpha \cdot \beta$ processors are expected to succeed, and only a single one of those needs to access the memory for each root computation.

Algorithm 1: Using precomputation to evaluate $\sqrt[a]{x}$

$T_{\text{general}} = 2\mathsf{Mul} + \mathsf{Table}(M)$

$T_{\mathsf{MinRoot}} = 8 \qquad \#CPU = 2^{128} \qquad M = 2^{128} \qquad \text{speedup} : 32$

2.2 An Approach Using Smoothness

We now describe an algorithm based on smoothness, which is similar to a previous work published by Adleman and Kompella in STOC '88 [2]. This algorithm uses smoothness to compute some arithmetic functions with logarithmic depth and a large number of processors. Starting with the randomization step of Section 2.1, the main idea is to assume that the value $y = x \cdot r^a \bmod p$ is B-smooth when lifted to the integers, *i.e.* it only has prime factors smaller than B, for some bound B.

Preliminaries. We use $\pi(x)$ to denote the prime counting function $\pi(x) = \#\{p \leq x \mid p \text{ is prime}\}$. According to the prime number theorem [38, Theorem 6.9], we have $\pi(x) \approx x/\ln(x)$.

We use the following result by Dickman to estimate the probability of a random n-bit number to be B-smooth:

Theorem 1 ([38, Theorem 7.2], [21]). *Let $\psi(x, y)$ be the number of positive integers not exceeding x composed entirely of prime numbers not exceeding y, and let $\rho(u)$ be [the Dickman function]. Then, for any $U \geq 0$ we have*

$$\psi(x, x^{1/u}) = \rho(u)x + \mathcal{O}\left(\frac{x}{\log(x)}\right)$$

uniformly for $0 \leq u \leq U$ and all $x \geq 2$.

In particular, the probability that an n-bit number is B-smooth is approximately $\rho\left(n/\log_2(B)\right)$.

Finally we use the following result, with $Q(x)$ the number of square-free integers not exceeding x:

Theorem 2 ([38, Theorem 2.2]). *For all* $x \geq 1$

$$Q(x) = \frac{6}{\pi^2}x + \mathcal{O}(x^{1/2})$$

The algorithm works as follows:

Precomputation.

- Compute $\sqrt[e]{q}$ for all small primes $q \leq B$
- Compute r^a, r^{-1} for a set of R random values r

Online phase. Given a challenge x, R groups of processors (each group of size $\pi(B)$) will do the following in parallel (steps 4 and 5 use $\pi(B)$ processors, and the other steps use a single processor):

1. Pick one of the randomly generated r (one value per group)
2. Compute $y = x \cdot r^a \bmod p$
3. Lift y to the integers
4. Perform trial division of y by all primes $q \leq B$, in parallel. Denote $\{q_i\}$ the set of primes that divide y
5. Compute $z = \prod \sqrt[e]{q}_i \cdot r^{-1} \bmod p$ (all terms are precomputed)
6. If $z^a = x$ return z

The algorithm succeeds if the value y at step 2 is B-smooth and square free. In this case $y = \prod q_i$, hence $\sqrt[e]{x} = \sqrt[e]{y} \cdot r^{-1} = \prod \sqrt[e]{q_i} \cdot r^{-1}$. The probability of y being B-smooth and square-free can be approximated as $\rho(256/\log_2(B)) \times 6/\pi^2$ by heuristically assuming that B-smoothness and being square-free are independent; therefore we choose $R \gg \pi^2/6\rho(256/\log_2(B))$, and we obtain a high probability of success, thanks to the randomization done in step 1. Note that the required heuristic can be greatly weakened by the optimization of allowing square factors discussed below and we successfully tested the heuristic for the parameters of interest (cf. the discussion on page 12).

This algorithm is similar to the index calculus one to compute discrete logarithms. Its complexity is sub-exponential. We now discuss some optimizations to make the algorithm closer to usable in practice.

Allowing Square Factors in y. For a prime q with $q^e \leq B < q^{e+1}$, we do trial division with q, q^2, $\ldots q^e$, and each trial adds a copy of q to the set $\{q_i\}$ if it is successful. With this tweak, the algorithm succeeds as long as y is B-powersmooth, *i.e.* all prime powers q^ν dividing y satisfy $q^\nu < B$. There is no simple formula to evaluate the complexity of this variant, but with the parameters we use, the probability of being B-powersmooth is essentially the same as being B-smooth; therefore we increase the success rate by a factor $\pi^2/6 \approx 1.64$ (the inverse probability of being square-free) with a very small increase in the number of processors.

Avoiding the Final Test. The purpose of the final test, computing z^a, is to verify if y is B-smooth. In order to avoid the latency of computing this, we can check if the q_i factors are enough and we are missing no bigger ones by precomputing an approximation of $\log(q_i)$ and check if $\sum \log(q_i) \approx \log(y) \approx \log(p)$ to detect when y is B-smooth. Note that the approximation does not need to be very good: on failure, z^a differs from y by at least some prime factor that was not found, which is at least 2 (and for square-free y, at least B). We can improve this and rule out missing very small prime factors q by testing divisibility by q^ν for larger ν up to $q^\nu < p$ rather than $q^\nu < B$ for very small q at a very small increase in the number of processors.

Finally, we obtain the Algorithm of Figure 4, where black boxes represent processors and green boxes represent groups of processors. For simplicity, we assume that a single group of processors will succeed and return a value. The latency is one multiplication, one trial division, and the final multiplication of several terms. We assume that the multiplication circuit can efficiently deal with empty factors and the complexity depends on the number of non-unit terms in the product.

Using precomputed Discrete Logarithms. In case it is not practical to build a multiplication circuit dealing only with the non-empty terms, we propose an alternative approach. Instead of having each processor returning $z_i = \sqrt[a]{q_i}$, we precompute the discrete logarithm of z_i in $\mathbb{F}_p$ using a generator g. If we denote the logarithm as ν_i with $z_i = g^{\nu_i}$ (and $\nu_i = 0$ for empty factors) we can replace the product $z = \prod z_i \bmod p$ by a sum $\nu = \sum \nu_i \bmod p-1$ and an exponentiation $z = g^\nu$. Using the assumptions of Section 1.2, the sum of $\pi(B)$ terms has latency only $\log_2(\pi(B))/8$. To compute the exponentiation with low latency, we split ν into 32 bytes, and use table lookups for each byte followed by a multiplication tree with latency 5.

Concrete Parameters for $log_2(p) \approx 256$. Parameters that minimize the total complexity can be chosen as:

$$B = 2^{35} \qquad\qquad R = 2^{24}$$

With those parameters, there are 2^{24} groups of processors, and each group has roughly $\pi(2^{35}) \approx 2^{30.5}$ processors to do trial division in parallel (a total of $2^{54.5}$ processors). This succeeds with high probability because the probability that a 256-bit number is 2^{35}-smooth can be approximated as $\rho(256/35) \approx 2^{-21.6} \gg 1/R$, and we assume than the probability of being 2^{35}-powersmooth is the same.

To verify this estimate, we performed experiments with Bach's algorithm to generate factored random numbers [7]. Out of 2^{30} random 256-bit numbers, we observed 367 2^{35}-smooth numbers (a fraction of $2^{-21.5}$); all of them are also 2^{35}-powersmooth, and 169 of them are square-free (a fraction of $2^{-22.6}$). This closely matches the theoretical estimation. In order to deal with factors $q^\nu < B$, we actually need slightly more than $\pi(B)$ processors but the increase is negligible $(2^{30.5} + 2^{14.1})$.

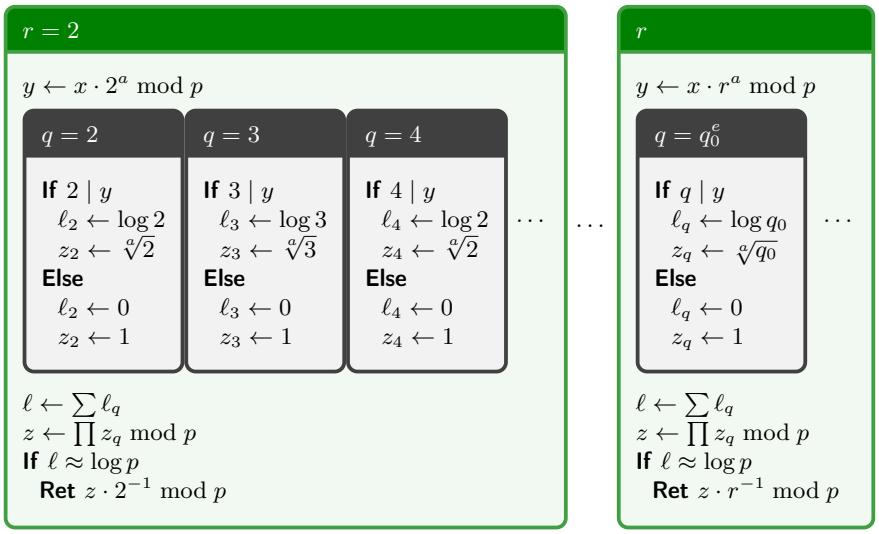

Fig. 4. Algorithm using B-smooth numbers

Assuming that we can neglect the communication time, the latency of this algorithm is one multiplication, one trial division, and about 4 multiplications at the end (assuming there are at most 15 primes in the decomposition of y), for a total latency of 6 units. This algorithm does not use any table lookup, each processor only uses a single hard-coded value for $\sqrt[a]{q_0}$, r^a, and r^{-1}.

Algorithm 2: Using smoothness to evaluate $\sqrt[a]{x}$

$T_{\text{general}} = 6\mathsf{Mul}$

$T_{\mathsf{MinRoot}} = 6 \qquad \#CPU = 2^{54.5} \qquad M = 0^a \qquad \text{speedup} : 42$

[a] We write $M = 0$ because there are no table lookups in this algorithm, but each processor uses a constant amount of memory, resulting in $2^{54.5}$ memory cells in total.

Alternatively, parameters can be chosen to minimize the latency:

$$B = 2^{64} \qquad R = 2^{10}$$

With those parameters the total number of processors is somewhat higher at $2^{68.5}$ but we expect fewer factors q_i so that the final multiplication only has latency 3 (assuming at most 7 primes), and the total latency is 5 units.

3 Optimizing the Smoothness Algorithm

In this section we present several ideas that allow us to reduce the number of CPUs needed in the smoothness algorithm. Our aim is to make our algorithm as close to practical as possible, and for doing this we will reduce the speedup provided by the algorithm in the previous section and provide different trade-offs. When combined, we are able to reduce the number of processors required below 2^{30} (with a 256-bit prime). This has allowed us to run experiments on the full algorithm (the code is available as supplementary material[12]).

3.1 Using Almost-Smooth Numbers

Our first improvement relaxes the constraint of y being B-smooth to allow a single medium-size factor, in addition to an arbitrary number of small ones. Formally, we say that an integer is (B, B')-almost-smooth (with $B' > B$) if all its prime factors are smaller than B', and at most one factor is larger than B. It will require a large precomputed table to deal with the bigger factor, but this will allow to increase the probability of y verifying the condition for similar values of B, reducing R and the overall number of processors needed.

After doing trial division, we detect that y is almost-smooth by checking the magnitude of the rough part after removing all factors smaller than B. Since we are mostly interested in cases with $B' < B^2$, if the rough part is smaller than B' then there is a single prime factor between B and B' (or none if y is B-smooth). We precompute the roots of all prime numbers between B and B' and the processor that finds an almost-smooth y will make a table lookup to recover it.

In practice, we modify the algorithm to compute z^{-1} in parallel with the computation of z, using precomputed tables of $q_i^{-1/a} = 1/\sqrt[a]{q_i}$. We evaluate z^{-1} as $\bar{z} \leftarrow \prod q_i^{-1/a} \bmod p$, and we compute the rough part as $y \cdot \bar{z}$. The resulting algorithm is shown as Figure 5.

We know no simple analytic way to compute the probability of almost-smoothness, but we could estimate it from experiments.

Concrete Parameters for $log_2(p) \approx 256$. Good parameters can be chosen as:

$$B = 2^{32} \qquad B' = 2^{65} \qquad R = 2^{20}$$

Experimentally, the probability of a 256-bit number to be $(2^{32}, 2^{65})$-almost-smooth is about 2^{-18}; this is a significant increase compared to the probability of begin 2^{32}-smooth ($\rho(256/32) \approx 2^{-24.9}$). Therefore with $R = 2^{20}$ there is a high probability of success.

The latency increases because of the table-lookup. Using our assumptions, this has a latency of 6; we obtain a latency of one multiplication, one trial division, 4 multiplications to compute $y \cdot \prod \bar{z}_q$, one table lookup, and one final multiplication.

[12] https://github.com/Cryptosaurus/VDF_cryptanalysis_code

This algorithm requires a huge amount of memory, but we stress that the memory is not accessed simultaneously by all processors; for each computation of $\sqrt[a]{x}$ only a few processors succeed (those that get an almost-smooth y) and only one of them needs to make a memory access.

Algorithm 3: Using smoothness to evaluate $\sqrt[a]{x}$, with a medium-size factor

$T_{\text{general}} = 6\mathsf{Mul} + \mathsf{Mod} + \mathsf{Table}(M)$

$T_{\mathsf{MinRoot}} = 13 \qquad M = 2^{59.5} \qquad \#CPU = 2^{48} \qquad \text{speedup} : 20$

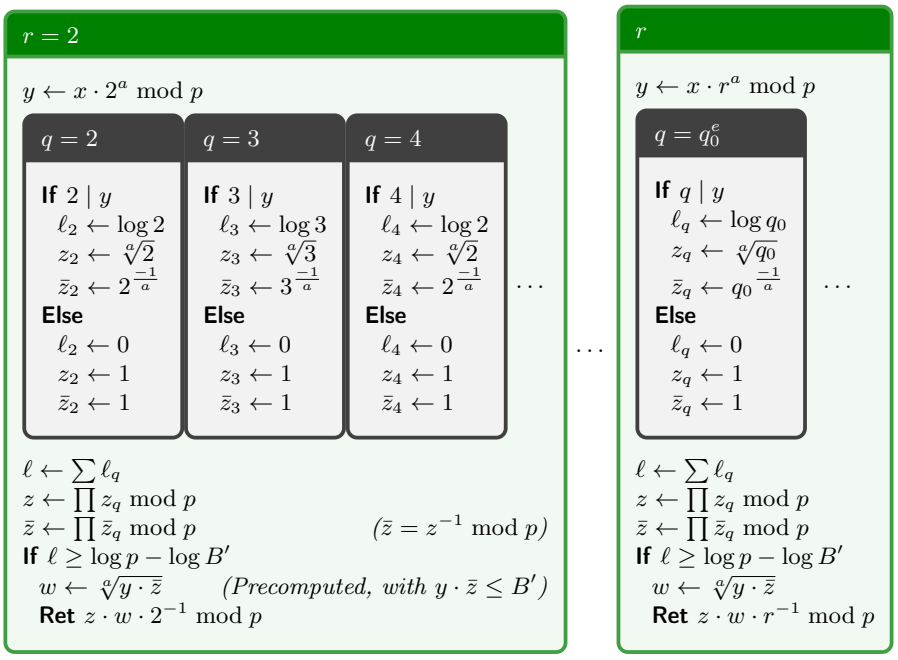

Fig. 5. Improved algorithm using (B, B')-almost-smooth numbers

3.2 Pre-filtering

We observe that the randomizing step ($y \leftarrow x \cdot r^a \bmod p$) uses only one processor per group, so that many processors sit idle during this step. To optimize the algorithm, we can try several values r in each group, and keep the value $y = x \cdot r^a \bmod p$ that seems more promising in each group. For instance, with a

smoothness bound $B = 2^{32}$ as above, each group has $\pi(B) \approx 2^{27.6}$ processors. We can pick $2^{27.6}$ random r's in each group and keep the smallest value $y = x \cdot r^a \bmod p$; we expect y to be of size $256 - 27.6 \approx 228$ bits which increases the probability of smoothness.

For a more advanced filtering, we do a smoothness test with a smaller bound $B_0 < B$, and we keep the candidate y with the largest B_0-smooth part (or, we keep candidates with a B_0-smooth part larger than some threshold t). This filtering requires $\pi(B_0)$ processors, so we can chain it with a first step that keep the smallest y out of $\pi(B_0)$ candidates. Figure 6 shows this pre-filtering algorithm.

Concrete Parameters for $log_2(p) \approx 256$. Parameters can be chosen as:

$$B = 2^{32} \qquad B' = 2^{65} \qquad B_0 = 2^{20} \qquad t = 2^{76} \qquad R = 2^{12}$$

Experimentally, the probability of having a 2^{20}-smooth part larger than $t = 2^{76}$ is about $2^{-9.3}$ for a 256-bit value. With the parameters above, we consider $\pi(B)/\pi(B_0) \approx 2^{11.3}$ candidates y in each group, therefore with high probability one of them will pass the filter. After filtering those candidates, the probability that they are $(2^{32}, 2^{65})$-almost-smooth is about 2^{-11}.

Using the initial filter the size of y is reduced by 10 bits; this increases the probability to roughly $2^{-9.5}$.

Algorithm 4: Using smoothness to evaluate $\sqrt[e]{x}$, with medium-size factor and pre-filter

$$T_{\text{general}} = 7\mathsf{Mul} + \mathsf{Mod} + \mathsf{Table}(M)$$
$$T_{\mathsf{MinRoot}} = 14 \qquad M = 2^{59.5} \qquad \#CPU = 2^{40} \qquad \text{speedup} : 18$$

3.3 Exploiting the Shape of p

The prime used in MinRoot has a special shape: $p = 2^{32}q + 1$. Therefore, the low 32-bits of a product $x \cdot y \bmod p$ with $y < 2^{32}$ only depend on the lowest and highest 32 bits of x (and y). We use this property to improve our algorithm, by precomputing values r with $r^a < 2^{32}$ that generate a product $x \cdot r^a$ with zeros in the least significant bits, given the low and high bits of x. This will allow to reduce R for similar parameters.

To take advantage of this, we modify the online algorithm to perform two steps of randomization: first with an arbitrary r_0, then with the precomputed value r_1:

Precomputation. Initialize table $\mathcal{T}$:

- For all $x_{\text{low}} < 2^{32}$, $x_{\text{high}} < 2^{32}$, consider $x = x_{\text{low}} + 2^{224}x_{\text{high}}$

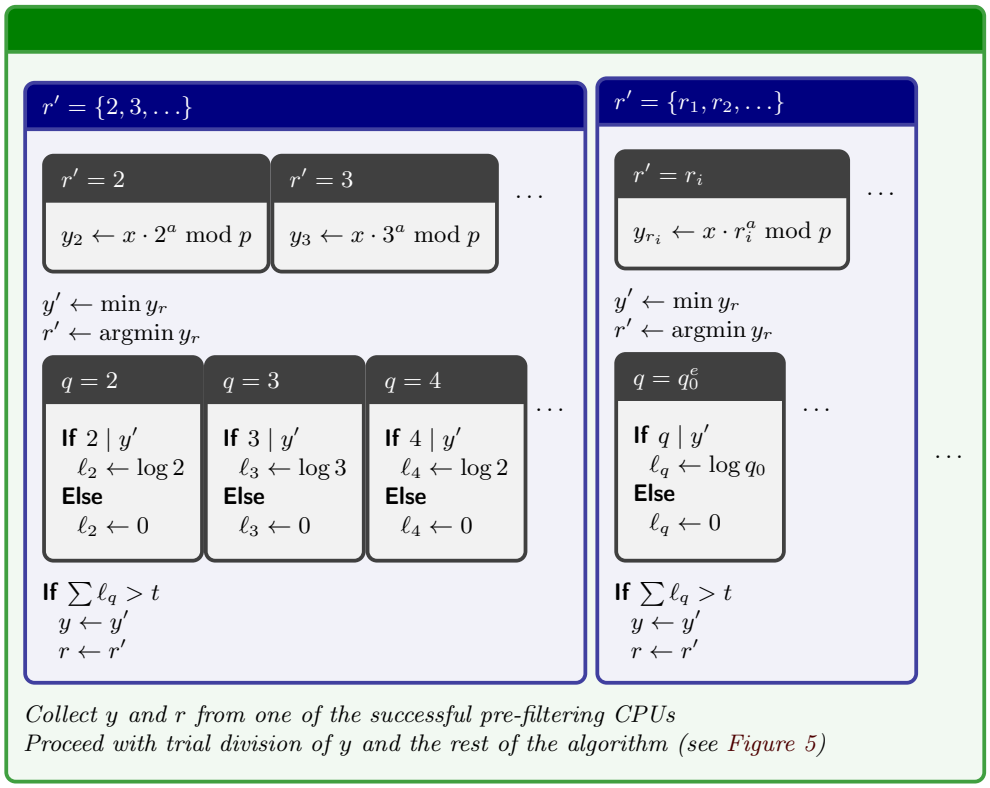

Fig. 6. Pre-filtering step (only one CPU group shown)

– For all $y < 2^{32}$, if $\mathrm{LSB}_{32}\big(x \cdot y \bmod p\big) = 0$, set $\mathcal{T}[x_{\text{low}}, x_{\text{high}}] \leftarrow \sqrt[e]{y}$

Online phase. Given a challenge x:

1. Pick one of the randomly generated r_0
2. Compute $y_0 = x \cdot r_0^a \bmod p$
3. Recover the precomputed value $r_1 = \mathcal{T}[\mathrm{LSB}_{32}(y_0), \mathrm{MSB}_{32}(y_0)]$
4. Compute $y_1 = y_0 \cdot r_1^a \bmod p$

This produces a value y_1 that is a multiple of 2^{32}, therefore the effective length for the smoothness test is reduced by 32 bits.

Concrete Parameters for MinRoot. We keep the same smoothness parameters as above, but we reduce the number of groups needed:

$$B = 2^{32} \qquad B' = 2^{65} \qquad B_0 = 2^{20} \qquad t = 2^{76} \qquad R = 2^8$$

By reducing the size of y by an extra 32 bits, we obtain 214-bit values; after filtering candidates with a 2^{20}-smooth part larger than $t = 2^{76}$, the probability that they are $(2^{32}, 2^{65})$-almost-smooth is about 2^{-6}, corresponding to a gain of 3.5 bits.

Algorithm 5: Using smoothness to evaluate $\sqrt[a]{x}$, with medium-size factor, pre-filter, and the special shape of p

$T_{\text{general}} = 8\mathsf{Mul} + \mathsf{Mod} + 2\mathsf{Table}(M)$[a]

$T_{\mathsf{MinRoot}} = 21 \qquad M = 2^{64} \qquad \#CPU = 2^{36} \qquad \text{speedup} : 12$

[a] This algorithm also requires a precomputation of 2^{96} roots

However this variant requires a memory access to the table $\mathcal{T}$ from each processor doing the initial randomization, therefore this is unlikely to be an improvement in practice with a huge table. Trade-offs with a smaller table (controlling fewer bits of y) might be more practical; for instance, we can use tables of size 2^{32} to control 16 bits of y (each processor could hold a copy of table). We obtain 230-bit values, and the probability that they are $(2^{32}, 2^{65})$-almost-smooth after filtering is about $2^{-7.6}$.

In the end, this technique produces small improvements, and cannot be used together with the techniques proposed in the following sections.

3.4 Rational Reconstruction

Rational reconstruction takes an element x in $\mathbb{F}_p$ and writes it as a fraction: $x = \alpha/\beta$ with $\alpha, \beta \in \mathbb{F}_p$. Wang proposed an algorithm that finds a solution with $\alpha \approx \beta \approx \sqrt{p}$, based on the extended Euclidean algorithm [58]. Indeed, running the extended Euclidean algorithm on x and p generates a series of relations $u_i \times x + v_i \times p = w_i$. Each relation defines a rational reconstruction: $x \equiv w_i/u_i \bmod p$. During the extended Euclidean algorithm, the magnitude of (u_i) and (v_i) are increasing while (w_i) is decreasing; if we stop after half the number of iterations we obtain $u_i \approx w_i \approx \sqrt{p}$.

The online algorithm can be improved as follows using rational reconstruction:

1. Pick one of the randomly generated r
2. Compute $y = x \cdot r^a \bmod p$
3. Reconstruct a fraction $y = \alpha/\beta$
4. Lift α and β to the integers
5. Do trial division of α and β

If α and β are both smooth, we easily deduce $\sqrt[a]{x}$ from precomputed tables:

$$\sqrt[a]{x} = \prod \sqrt[a]{q'_j} \cdot \prod q_i^{\frac{-1}{a}} \cdot r^{-1} \quad \text{if } \beta = \prod q_i \text{ and } \alpha = \prod q'_j$$

Since α and β are of the order of $\sqrt{p}$, they are significantly more likely to be smooth and we can use a smaller bound B. The smoothness test for α and β is done in parallel, using $2 \times \pi(B)$ processors.

Latency of Rational Reconstruction. In practice, variants of the binary GCD algorithm (such as the plus-minus algorithm of Brent and Kung [14]) should have a lower latency than the Euclidean algorithm because they only use addition and subtractions. Extended versions of the binary GCD algorithm produce relations of the form $u_i \times x + v_i \times p = w_i \times 2^{z_i}$, that can also be used in our case.

A recent work [51] studies low-latency implementation of extended GCD algorithms. They describe an ASIC built on 16nm that performs extended GCD of 256-bit integers with latency 89 ns in constant time. For rational reconstruction, only half the rounds are necessary. Moreover, the constant-time circuit uses more rounds than required on average to maintain a constant time, and the circuit uses 16 nm technology while the Supranational implementation of MinRoot uses 12 nm technology. Therefore, we estimate that the regular extended GCD algorithm for n-bit numbers has latency $\mathsf{EGCD}(n) = \frac{5}{16}n$, and thus rational reconstruction would have a latency of roughly 40 time units.

To reduce the latency further, some GCD algorithms use precomputed tables to reduce the number of iterations, such as the right-shift k-ary algorithm of Sorenson [50], with only $\mathcal{O}(\log(n)/k)$ iterations using tables of size k. Further work would be needed to evaluate the latency of such algorithms when a massive number of processors and precomputation is available.

Concrete Parameters for $log_2(p) \approx 256$. We combine rational reconstruction with the pre-filtering step: we consider many fractions α/β, and we only keep fractions where α and β both have a large B_0-smooth factor. Parameters can be chosen as:

$$B = 2^{24} \qquad B' = 2^{45} \qquad B_0 = 2^{16} \qquad t = 2^{28} \qquad R = 2^{13}$$

Experimentally, the probability of having a 2^{16}-smooth part larger than $t = 2^{28}$ is about $2^{-2.9}$ ($2^{-5.8}$ for a pair (α, β)). With the parameters above, we consider $\pi(B)/\pi(B_0) \approx 2^{7.4}$ candidates $y = \alpha/\beta$ in each group, therefore with high probability one pair (α, β) will pass the filter. After filtering those candidates, the probability that they are $(2^{24}, 2^{45})$-almost-smooth is about $2^{-5.5}$ (2^{-11} for a pair (α, β)).

Algorithm 6: Using smoothness to evaluate $\sqrt[e]{x}$, with medium-size factor, pre-filter, and rational reconstruction

$$T_{\text{general}} = 0.5\mathsf{EGCD}(p) + \mathsf{Table}(M) + 7\mathsf{Mul} + \mathsf{Mod}$$
$$T_{\mathsf{MinRoot}} = 54 \qquad M = 2^{40} \qquad \#CPU = 2^{34} \qquad \text{speedup} : 4.7$$

3.5 Parallel Smoothness Test

We propose here another idea that can be used to reduce the latency, at the expense of more communication. Instead of randomizing with $y \leftarrow x \cdot r^a \bmod p$

for random r until the value y is smooth when lifted to the integers, we can compute $y \leftarrow x + r \cdot p$ over the integers, for small $r \in \{0, 1, \dots R\}$. We obtain slightly larger integers, but again if one of them is smooth we can compute $\sqrt[a]{y} \bmod p$ and deduce $\sqrt[a]{x} \bmod p = \sqrt[a]{y} \bmod p$.

The advantage of this approach is that we can test all candidates y for smoothness simultaneously. Indeed, we don't have to do trial division for all $x + r \cdot p$ and all small prime powers q_i. We just have to compute $x \bmod q_i$, and we directly know the values of r for which $x + r \cdot p$ is divisible by q_i: those with $r \equiv -x \cdot p^{-1} \bmod q_i$. Since p is prime, it is always invertible mod q_i and $p^{-1} \bmod q_i$ can be precomputed. Figure 7 shows this algorithm.

In a model with free communication (*e.g.* with a parallel RAM that can be accessed simultaneously by each processor), this should be quite efficient: each processor doing trial division just has to send a list of candidates r such that $x + r \cdot p$ is divisible by q_i, and one processor per candidate r will merge the data (with a parallel RAM: each processor writes the factor found in a region dedicated to a given candidate r).

The main factor for the complexity of this algorithm is the number of messages to send; on average it is equal to $\sum_{q^\nu < B} \frac{R}{q^\nu}$. In order to minimize latency, we use multiple processors for small factors q_i, so that each processor has a single message to send; therefore, the number of processor is $\sum_{q^\nu < B} \lceil \frac{R}{q^\nu} \rceil$.

When taking communication into account, there will be some cost to pay to route the messages. Using a hypercube topology (each processor is connected to $\log(n)$ other processors), n processors can route n messages in probabilistic time $\mathcal{O}(\log(n))$ [56].

Concrete Parameters for $log_2(p) \approx 256$. This idea can be combined with the use of almost-smoothness, but not with pre-filtering because we consider many values of y simultaneously. We consider the following parameters:

$$B = 2^{32} \qquad B' = 2^{45} \qquad R = 2^{26}$$

With those parameters, the number of processors for trial division is $\sum_{q^\nu < B} \lceil \frac{R}{q^\nu} \rceil = 2^{28.8}$, and the average number of messages to route is $\sum_{q^\nu < B} \frac{R}{q^\nu} = 2^{28}$ (slightly higher than $\pi(B) \approx 2^{27.6}$). Each candidate y is a $256 + 26 = 282$-bit number. The probability that they are $(2^{32}, 2^{45})$-almost-smooth is about 2^{-24}, so that the algorithm succeeds with high probability.

Algorithm 7: Using smoothness to evaluate $\sqrt[a]{x}$, with medium-size factor, and parallel smoothness test

$$T_{\text{general}} = \mathsf{Table}(M) + 6\mathsf{Mul} + \mathsf{Mod}$$
$$T_{\mathsf{MinRoot}} = 13 \qquad M = 2^{40} \qquad \#CPU = 2^{29} \qquad \text{speedup} : 20$$

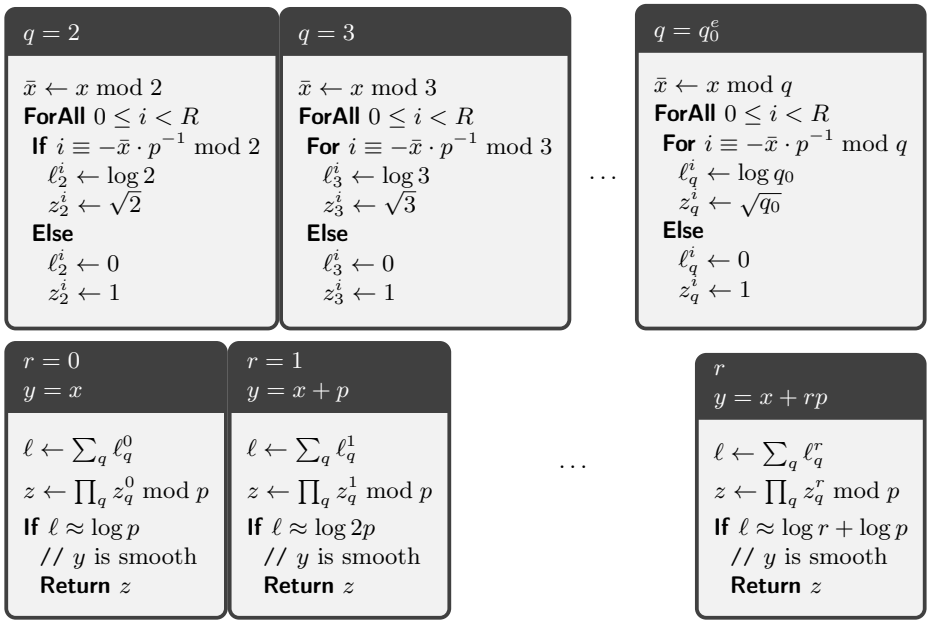

Fig. 7. Parallel smoothness test. We use a for "**ForAll** i" loop in the algorithmic description, but a real implementation will directly iterate over values of i such that $i \equiv -\bar{x} \cdot p^{-1} \bmod q$.

3.6 Parallel Smoothness Test and Rational Reconstruction

Finally, we combine the ideas of rational reconstruction and the parallel smoothness test. First, we use rational reconstruction on x, to obtain two different fractions $x = \alpha/\beta \bmod p = \gamma/\delta \bmod p$. Using intermediate values from the extended Euclidean algorithm, we just keep two consecutive steps, and we expect α, β, γ, and δ to be slightly larger than $\sqrt{p}$. We observe that for any r we have (assuming $\beta + \delta \cdot r \not\equiv 0 \bmod p$):

$$\frac{\alpha + \gamma \cdot r}{\beta + \delta \cdot r} \equiv \frac{\beta \cdot x + \delta \cdot x \cdot r}{\beta + \delta \cdot r} \bmod p \equiv x \bmod p$$

Therefore, we consider a series of fractions $\frac{\alpha+\gamma \cdot r}{\beta+\delta \cdot r}$ for small $r \in \{0, 1, \ldots R\}$ and deduce $\sqrt[e]{x}$ when $\alpha + \gamma \cdot r$ and $\beta + \delta \cdot r$ (integers of magnitude roughly $R \cdot \sqrt{p}$) are simultaneously smooth. As in the previous section, we obtain the divisibility information on all candidates ($\alpha+\gamma \cdot r$ or $\beta+\delta \cdot r$) with a single modular reduction.

Concretely, when doing trial division of $\alpha + \gamma \cdot r$ by q_i, we have $\alpha + \gamma \cdot r \equiv 0 \bmod q_i \iff r \equiv -\alpha \cdot \gamma^{-1} \bmod q_i$. Therefore each processor must compute $\alpha \bmod q_i$ and $\gamma^{-1} \bmod q_i$; this differs from Section 3.5 where $p^{-1} \bmod q_i$ was precomputed. We note that γ is not necessarily invertible in $\mathbb{Z}_{q_i}$, but having a non-invertible value is relatively rare and we neglect it to simplify the analysis (this only introduces some false negatives).

There are several possibilities to compute $\gamma^{-1} \bmod q_i$: we can precompute a table of inverses in $\mathbb{Z}_{q_i}$, or can compute it on the fly using either the extended Euclidean algorithm or as $\gamma^{\varphi(q_i)-1} \bmod q_i$ using Euler theorem. Using the assumptions of Section 1.2, the fastest approach is a precomputed table, with latency 6 units, but it requires a large amount of memory.

Concrete Parameters for $log_2(p) \approx 256$. We consider the following parameters:

$$B = 2^{27} \qquad B' = 2^{45} \qquad R = 2^{21}$$

With those parameters, the number of processors for trial division is $\sum_{q^\nu < B} \lceil \frac{R}{q^\nu} \rceil = 2^{23.9}$ ($2^{24.9}$ to do trial division of $\alpha + \gamma \cdot r$ and $\beta + \delta \cdot r$ in parallel), and the average number of messages to route is $\sum_{q^\nu < B} \frac{R}{q^\nu} = 2^{23}$ (2^{24} when considering both the numerator and denominator). Each candidate y is a $128 + 21 = 149$-bit number. The probability that they are $(2^{27}, 2^{45})$-almost-smooth is about $2^{-9.2}$, so that the algorithm succeeds with high probability after 2^{21} attempts.

We assume that $\gamma^{-1} \bmod q_i$ is computed on the fly (using precomputed tables would require a memory of size $2^{48.8}$). Following the latency estimate for the extended GCD algorithm in Section 3.4, we assume that computing inverses in $\mathbb{Z}_{q_i}$ has latency $\frac{5}{16} \log_2(q_i)$, which we round up to $\log_2(q_i)/2 \approx 14$ units.

Algorithm 8: Using smoothness to evaluate $\sqrt[e]{x}$, with medium-size factor, rational reconstruction, and parallel smoothness test

$$T_{\text{general}} = 0.5\mathsf{EGCD}(p) + \mathsf{EGCD}(B) + \mathsf{Table}(M) + 7\mathsf{Mul} + \mathsf{Mod}$$
$$T_{\mathsf{MinRoot}} = 68 \qquad M = 2^{40} \qquad \#CPU = 2^{25} \qquad \text{speedup} : 3.7$$

We have implemented (a serialized version of) this algorithm in practice with those parameters, and it succeeds with probability more than 99% (2 failures out of 1000 trials). When working with a 128-bit prime p (as in Veedo), the attack requires only 2^{13} processors and 2^{40} memory (with $B = 2^{14}$, $B' = 2^{45}$, $R = 2^9$), which might be implementable in practice (as a reference, the largest GPUs today have more than 2^{14} cores and some motherboards support 12TB of memory). The code is available as supplementary material[12].

4 Relation with Discrete Logarithm

We observe that the algorithms in Sections 2 and 3 are very close to classical algorithms for the discrete logarithm problem: Section 2.1 is similar to the baby-step giant-step algorithm [47], and Section 2.2 is similar to index calculus [1]. In Appendix A.1 we describe another algorithm similar to the Pohlig-Hellman algorithm [40].

4.1 Advanced logarithm methods

It is natural to consider more advanced discrete logarithm algorithms in the context of low-latency computation of roots.

Using ECM. The elliptic curve method [34] (ECM) is a factorization algorithm that is particularly efficient to find small factors. It could be used instead of trial division for smoothness tests in the index calculus type attacks. The idea would be to precompute some curves so that the value you want has smooth order. But it is unclear how to make this work, or whether the amount of computation for doing ECM would end up being within the desired constraints.

Using NFS-type algorithms. The complexity of index calculus for the discrete logarithm problem is in the class $L[1/2]$. There are more efficient algorithms known, with complexity in the class $L[1/3]$, such as the Number Field Sieve [26]. Unfortunately, these algorithms seem to have an intrinsically sequential structure. Further work is needed to evaluate the potential of these ideas, but the apparent sequentiality is a serious obstacle in the context of low-latency algorithms.

4.2 Reduction to parallel discrete logarithm

On another hand, we show that a large class of attacks on algebraic VDFs, which includes all our attacks, is reducible to the parallel discrete logarithm computation. Informally, an attack that computes roots by precomputation and low-latency multiplications may be used to compute the discrete log.

Definition 1. *We call an algorithm $\mathcal{A}$ algebraic w.r.t. precomputed memory $\mathsf{C} = [c_1, c_2, \ldots, c_k]$ if on input x it outputs a result in the form*

$$\mathcal{A}(x; \mathsf{C}) \to (y, d_1, d_2, \ldots, d_k, d') \tag{1}$$

such that

$$y = c_1^{d_1} c_2^{d_2} \cdots c_k^{d_k} x^{d'} \tag{2}$$

It is easy to see that all algorithms from Section 3 are algebraic: the result is always a product of some powers of stored values, which in our case are numbers with known roots.

Theorem 3. *Let p be a prime, and let DL be an algorithm that computes the discrete logarithm in $\mathbb{F}_p$ with computational cost $T_{DL}(p)$. Let A be a parallel algebraic algorithm that computes the power function*

$$f(x) = x^d \bmod p$$

using D processors and memory in online time (latency) L and precomputation time T and produces a result in the form (2), where $d - d'$ is coprime with $p - 1$. Then there exists a parallel algorithm B that computes the discrete logarithm in $\mathbb{F}_p$ (using base g) in time $L' = L + \log_2 k$ using D processors and memory and precomputation time $T + D \cdot T_{DL}(p)$.

Proof. We construct such an algorithm as follows. For each precomputed memory element we precompute its discrete logarithm in time $T_{DL}(p)$ spending the total of at most $D \cdot T_{DL}(p)$ precomputation and store it together with the value c_i.

In the online phase we run $\mathcal{A}$ as a subroutine for x and obtain

$$x^d = c_1^{d_1} c_2^{d_2} \cdots c_k^{d_k} x^{d'}$$

Given $\{c_i, d_i\}, d'$ we retrieve the log of each c_i from the memory and compute

$$\log_g x = \frac{\sum d_i \log_g c_i}{d - d'}$$

in time $\log_2 k$ using k processors. This ends the proof.

Therefore, any algebraic attack implies a parallel discrete logarithm computation with the same online latency and the same online computational complexity. Given that the latter problem has received some public scrutiny [49,18,48] (and impacts the real-world security of Diffie-Hellman [3]), we may guess that a further progress in low-latency attacks on algebraic VDFs, in particular the reduction in the CPU number D, would require a non-algebraic approach.

5 Application to VDF Constructions

The previous algorithms break the sequentiality of several VDF constructions, such as MinRoot [29], VeeDo [52] and Sloth++ [11]. An attacker with a large number of processors can compute the round function several times faster than a legitimate user, if we neglect communication and memory cost.

In particular, it contradicts the security claims of MinRoot: for instance, using the algorithm of Section 3.5, an attacker with 2^{29} processors can compute the round function about 20 times faster than a legitimate user.

5.1 Optimization for Iterated MinRoot

We can save the latency of the initial multiplication used by r'^a in the next round if the processor that succeeds broadcasts the factors to be multiplied ($\left\{ \sqrt[a]{q_i} \right\}$ and r^{-1}) rather than the final reduced result. Then each processor would compute the randomized input for the next round as:

$$\begin{aligned}(u' + v') \cdot r'^a &= (\sqrt[a]{u+v} + (u+i)) \cdot r'^a \\ &= \prod \sqrt[a]{q_i} \cdot r^{-1} \cdot r'^a + (u+i) \cdot r'^a\end{aligned}$$

This results in a product with one more term, but it does not affect the latency if the number of terms was not a power of two. The term $(u+i) \cdot r'^a$ is added using a multiply-and-add operation at the end, so that the latency of a MinRoot round is still just the latency of the a-th root.

5.2 Application to Sloth++

Sloth++ uses square roots in $\mathbb{F}_{p^2}$. The precomputation attack from Section 2.1 can be applied directly, but attacks from Section 2.2 and 3 rely on smoothness and there is no direct way to apply it in $\mathbb{F}_{p^2}$. Instead, we show how to reduce the computation of square roots in $\mathbb{F}_{p^2}$ to square roots in $\mathbb{F}_p$.

We assume that $\mathbb{F}_{p^2}$ is constructed as $\mathbb{F}_p[X]/\left(X^2+\alpha\right)$. An element b of $\mathbb{F}_{p^2}$ is a polynomial $b_0 + b_1 X$. The square root $z = z_0 + z_1 X$ of $b = b_0 + b_1 X$ satisfies:

$$\begin{aligned}
z^2 &= b \\
(z_0 + z_1 X)^2 &= b_0 + b_1 X \\
z_0^2 - \alpha z_1^2 + 2 z_0 z_1 X &= b_0 + b_1 X \\
&\begin{cases} 2 z_0 z_1 & = b_1 \\ z_0^2 - \alpha z_1^2 & = b_0 \end{cases} \\
&\begin{cases} z_0 & = b_1 / 2 z_1 \quad \text{(assuming } z_1 \neq 0\text{)} \\ \frac{a_1^2}{4 z_1^2} - \alpha z_1^2 & = b_0 \end{cases}
\end{aligned}$$

We denote $u = z_1^2$ and we obtain a quadratic equation in u:

$$\frac{b_1^2}{4u} - \alpha u = b_0 \qquad\qquad \frac{b_1^2}{4} - \alpha u^2 = b_0 \cdot u$$

We solve this equation by computing a square root in $\mathbb{F}_p$, and deduce z_0 and z_1 using another square root operation in $\mathbb{F}_p$, an inverse and a few multiplications. The inverse in $\mathbb{F}_p$ can be computed with the low latency algorithms of Sections 2 and 3.

5.3 Bigger moduli

It is natural to ask how our attacks scale with modulus size. Given the relation with discrete logarithm computation (Section 4), we expect that the number of CPUs needed for the attack becomes less feasible with moduli of 1024 bits and higher, as it happens with the RSA/DSA security. As different applications may be okay with the number of CPUs above a certain threshold, we provide attack complexity estimates for moduli sizes up to 2048 bits in Table 2.

Due to the space limit, we do not list the latency numbers and speedups in the table. However we note that the speedup in our model actually grows as the modulus increases. This is due to the fact that the plain root computation grows linearly with the modulus size, whereas, for example, the memory access cost is sublinear in it as long as the memory itself grows slower than prime p.

6 Low-latency Evaluation of Low-degree Exponentiation

We now consider a different problem: how to compute x^d with a *small* d in parallel faster than with the standard square and multiply. This can be directly

Table 2. Scalability of smoothness-based attacks. The numbers are given in $\log_2$ and assume an attack with success rate $1 - \exp(-1)$ (lower than the rates in Section 3). The code to compute these numbers is available at https://github.com/Cryptosaurus/VDF/tree/master/code/alt_code.

	Algorithm						
	Sec. 2.2	Sec. 3.1		Sec. 3.5		Sec. 3.6	
Modulus (bits)	#CPU	M	#CPU	M	#CPU	M	#CPU
96	27	34	22	29	13	27	11
128	33	36	28	30	17	29	14
192	43	41	38	40	22	34	19
256	51	51	47	41	27	40	23
384	67	57	62	49	34	44	30
512	80	63	75	59	41	53	36
768	102	78	98	65	53	62	47
1024	121	92	117	74	62	69	57
1536	154	108	150	91	79	87	72
2048	184	105	180	108	94	102	85

applied if the round function uses x^e with small e rather than $\sqrt[a]{x}$ with small a. Moreover, those techniques can be used to reduce the latency of the root computation in a VDF: an algorithm to compute x^d with latency smaller than $\log_2(d)$ squarings can speed up computing x^e with arbitrary e by replacing the square and multiply algorithm by a d-th power and multiply algorithm; if done naively, each step now requires up to $d-1$ multiplications, but those can be parallelized and do not affect the latency much.

6.1 Algorithm Using Shares and LUTs

Let us consider that we split each field element into a sum of s shares from a smaller domain, say, $s = 2$. We then have that $x \in \mathbb{F}_p$ is equal to $x = \ell + h$, where for instance ℓ corresponds to the lowest bits and h to the highest bits of x. The overall idea consists in precomputing some monomials in order to speed up the computation of the VDF primitive we consider.

In general, we have that (with operations in $\mathbb{F}_p$)

$$(\ell + h)^d = \sum_{i=0}^{d} \binom{d}{i} \ell^i h^{d-i}$$

Suppose that an adversary has precomputed $\binom{d}{i}\ell^i$ and h^i for all values of i, and for all possible ℓ and h (recall that ℓ and h live in spaces that are much smaller than $\mathbb{F}_p$). Assume also that they have access to several parallel processors, then it is possible to compute $(\ell + h)^d$ with a latency of 1 lookup, 1 multiplication, and whatever is needed for additions and reduction mod p. After optimizing away 2 trivial tables for $i = 0$ and $i = d$, overall, $2d$ processors are needed for this to work (to access $2d$ LUTs in parallel).

If $d \leq 4$ such a technique turns out not to be too interesting. However, as the degree d increases, this technique becomes more interesting since the latency does not change when d increases, only the number of processors needed (and the number of tables). Overall, in order to evaluate $(h + \ell)^d$, we need:

- $2d$ parallel processors,
- $2d$ tables of size roughly $p^{1/2}$,
- a latency of 1 lookup, 1 multiplication, and many additions.

This can be generalized to a higher number s of shares, in which case the numbers above become (without optimizing away some trivial tables)

- $s\binom{d+s-1}{d} = s\binom{d+s-1}{s-1}$ parallel processors,
- tables of size roughly $p^{1/s}$,
- a latency of 1 lookup, $\log_2(s)$ multiplications (since we don't have to do them sequentially), and many additions.

Again, while the overall complexity (and in particular the latency) increases with s, the latency does not depend on d. Thus, this technique can be become interesting when the degree is higher, in particular if d is much bigger than s.

Concrete Parameters for $log_2(p) \approx 256$. For instance, with $d = 2^{16}$ and $s = 4$, we obtain a circuit to evaluate $x^{2^{16}}$ using $4 \times 2^{46} = 2^{48}$ parallel processors, each using a table of size 2^{64}, with a latency of 1 lookup, 2 multiplications and a modular sum of 2^{46} terms. Using the assumptions in Section 1.2, the tables would be stored in RAM with an access latency of 6 units, and the sum of 2^{46} terms has a latency of 6 (using a tree with 6 levels where each level adds 256 terms). This corresponds to a latency of $6 + 2 + 6 = 14$, which is smaller than a direct evaluation with latency 16.

However the memory requirements of this algorithm are prohibitive, with $sd = 2^{18}$ tables of size 2^{64}, and a total of 2^{48} accesses to the tables.

Algorithm 9: Using shares and LUTs to evaluate $x^{2^{16}}$

$T = 14$ $\quad \#CPU = 2^{48}$ $\quad M = 2^{82}$ $\quad$ speedup : 16/14

6.2 Extensions

The above method can be applied to directly compute r consecutive applications of a VDF round function. Furthermore, we observe the bottleneck is the large tables of size $\sqrt[s]{p}$ for maps of the form $x \mapsto C \cdot x^i$ each. However, the functions computed by those tables are amenable to the very same technique and we can use the technique recursively with l layers by splitting the shares into sub-shares. We refer to Appendix A.3 for more details. The results are given in Table 3.

Table 3. Comparison of highly parallel low latency computation with standard VDF (only counting the multiplications in the latency).

Metric	Compressed	Recursive with l layers	Standard VDF
Tables (in bits)	$e^r 2^{\frac{\log_2 p}{s}} \log_2 p$	$e^r 2^{\frac{\log_2 p}{s^l}} \log_2 p$	0
Processors	$\binom{e^r+s-1}{s-1}\frac{e^{2r}}{2}$	$\binom{e^r+s-1}{s-1}^l \frac{e^{2r}}{2}$	1 or 2
Latency (in $\log_2 p$ bit MUL)	$2+\lceil \log_2(s) \rceil$	$2+l\lceil \log_2(s) \rceil$	$\lceil \log_2(e) \rceil \cdot r$

7 Practical Issues

The previous sections mostly consider ideal implementations of VDF from an algorithmic point of view. In this section we briefly discuss some practical issues, such as the communication cost.

7.1 Dealing with Errors

The algorithms given in this paper are probabilistic. Since we consider only the round function $\sqrt[a]{x}$, and VDFs use a large number of iterations (typically 2^{40} for MinRoot), we need a very high success rate in order to successfully compute the full VDF function.

However, it is easy to deal with rare erroneous computations, because the computations can be efficiently checked (if $y = \sqrt[a]{x}$ then $y^a = x$). One option is to repeat the algorithm when it fails; this increases the latency but if the failure rate is small, the average latency stays small. Another option is to run the standard implementation in parallel to the attack. If the attack succeeds we have the result with low latency, and if it fails we wait until the standard algorithm succeeds. If the failure rate is small, the average latency is still small.

7.2 Communication Cost

The analysis above essentially neglects communication costs. In practice, this is likely to be an important bottleneck, because communication between millions of CPUs takes time, and requires a large communication network; this is likely to dominate the cost of the machine [60]. The setting is quite different from usual cryptanalytic attacks, often embarrassingly parallel and not requiring communication between the processors (*e.g.* brute-force key search). Since our attacks target the round function, all cores must synchronize at least after each round, to collect the result from the core that succeeded and broadcast it to other cores (some algorithms might require even more communication). In this section, we briefly discuss how to implement those communications, and how practical this might be.

Massive Communication Network. In many of the discussed settings with high available parallelism it is necessary to broadcast a short input (*e.g.* 256 bits) with

very low latency to a large set of processors and later to collect short outputs from a small random subset of "successful" processors.

With more than 2^{20} processors, this would probably require too long wires for such broadcast and retrieval. Alternatives could be broadcasting wirelessly or even over the optical domain. Indeed the speed of optics might make it possible to flash with low latency the common input to the field of processors and later, with a few receiving detectors to retrieve outputs from a handful of lucky processors. There is ongoing research on integrating optical elements into existing chip design [6], and the hypothetical Twinkle factoring device by Shamir also used in optics for finding B-smooth numbers [45].

7.3 Physical Constraints

Speed of light. We want the attack to be faster than the standard implementation. Taking the Supranational implementation of MinRoot as a benchmark, the attack must have a latency of at most 230 ns; during this time light can only travel 70 meters; if we aim for an attack twice as fast as the standard implementation, light can only travel 35 meters. This limits the physical size of the machine that runs the attacks: it should be within a sphere of diameter 35 meters. Assuming that each processor has a volume of $0.025\,\mathrm{mm}^2$, at most $\frac{\pi}{6}35\,\mathrm{m}^3/0.025\,\mathrm{mm}^2 \approx 2^{50}$ processors can communicate within one round.

Cooling limit. Assuming each core consumes 1 W of power, the limit above would result in a power density of $50\,\frac{\mathrm{GW}}{\mathrm{m}^3}$, far exceeding the power density of a nuclear reactor.

The largest nuclear reactor in the world is the Taishan EPR, rated at 1.66 GW, and 4.59 GW of thermal capacity. This is a major constraint on building nuclear power plants leading us to claim that it is near impossible to build a system dissipating more than 100 GW of thermal power in one location on land. Adding an "engineering safety factor of 10" gives a limit of 1 TW, leading to a limit of about 2^{40} cores in a single machine.

Practical engineering constraints. Practical engineering constraints are very likely to lead to much lower limits than any of the above. It completely ignores power supply, cooling, and space for interconnect for communications, which will far exceed the size of the cores. However, the point of designing for "128 bit security" is to account for future improvements by adding safety margins, and so it is unclear how much practical engineering problems should influence this if they can't be translated into clear physical limits.

8 Conclusion

In this paper we propose several algorithms to compute roots in a finite field $\mathbb{F}_p$ with low latency. Even though it is not clear how efficient those algorithms would be if implemented in practice, they clearly show that computing roots is not a

sequential operation, breaking an assumption used in several VDF constructions. In particular, MinRoot is a VDF candidate that was proposed as a randomness beacon in the core layer of the Proof-of-Stake Ethereum protocol, but this project has been put on hold following our results [28].

Possible Tweaks. In order to limit the impact of these results, we considered some options to construct a VDF that would plausibly not be affected by the attacks. We have not looked into these alternatives in detail, and we do not claim that they are secure, but they could offer ideas for further analysis.

Using a low degree round function. Our strongest attacks (in Sections 2 and 3) compute $\sqrt[e]{x}$ with low latency. This breaks the VDF property because the standard implementation requires about $\log_2(p)$ squarings to compute the root. An option could be to use x^e with small e in the round function instead of $\sqrt[e]{x}$: this reduces the latency of the standard implementation and the algorithms of Section 2 are no longer competitive.
However, the algorithms of Section 6 show that a low-degree round function can also be sped up to a smaller extend using parallel computation.

Using a larger prime p. All the algorithms that we proposed have a complexity (number of processors) that is at best sub-exponential in p: the number of processors required to obtain a given advantage increases with p. If p is chosen large enough, it might be possible to achieve a sufficient security level. However, further work is required to gain confidence on the non-existence of better attacks, because the field has been barely explored.

Using extension fields (as in Sloth++). While taking square roots over $\mathbb{F}_{p^2}$ reduces to solving a quadratic equation over $\mathbb{F}_p$, which is solved by taking square roots over $\mathbb{F}_p$ (quadratic formula), this doesn't appear to be the case for cube roots or higher. The cubic formula may be useful, but for roots larger than 3 it isn't clear how to leverage the attack over $\mathbb{F}_p$. On the other hand, the attack may generalize more directly to extension fields, using a suitable smoothness basis for the extension field (similar to index calculus being extended to NFS).

Using elliptic curve groups. Another approach is to use elliptic curve groups for the round permutation because there is no straightforward notion of smoothness as there is in finite fields. Index calculus is less effective for solving DL on elliptic curve groups too. For instance, the round function could use $r \times x$, with x a curve point and r such that $3r = 1 \bmod q$ (assuming the EC group over $\mathbb{F}_p$ has order q). However, between rounds we would need to interleave this with some permutation on the curve group that is simple/algebraic over $\mathbb{F}_p$ and not a scalar multiplication.

Acknowledgements

This work was started during a gathering organized by Ethereum Foundation, where experts were invited to analyze the MinRoot VDF. A technical report of

the gathering is available at [28]. We would like to thank Ethereum Foundation for organizing the gathering, and all the participants for fruitful discussions. In particular, Dankrad Feist contributed to the discussion of physical constraints, and Anne Canteaut and Itai Dinur contributed appendices A.1 and A.2.

Gaëtan Leurent is supported by project Cryptanalyse from PEPR Cybersécurité (22-PECY-0010). Alex Biryukov was funded in part by the Luxembourg National Research Fund (FNR), project CryptoFin C22/IS/17415825.

References

1. Adleman, L.M.: A subexponential algorithm for the discrete logarithm problem with applications to cryptography (abstract). In: 20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979. pp. 55–60. IEEE Computer Society (1979). https://doi.org/10.1109/SFCS.1979.2, https://doi.org/10.1109/SFCS.1979.2
2. Adleman, L.M., Kompella, K.: Using smoothness to achieve parallelism (abstract). In: 20th ACM STOC. pp. 528–538. ACM Press (May 1988). https://doi.org/10.1145/62212.62264
3. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect forward secrecy: How Diffie-Hellman fails in practice. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 5–17. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813707
4. Ahrens, K., Zumbrägel, J.: DEFEND: towards verifiable delay functions from endomorphism rings. IACR Cryptol. ePrint Arch. p. 1537 (2023), https://eprint.iacr.org/2023/1537
5. Arun, A., Bonneau, J., Clark, J.: Short-lived zero-knowledge proofs and signatures. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part III. LNCS, vol. 13793, pp. 487–516. Springer, Heidelberg (Dec 2022). https://doi.org/10.1007/978-3-031-22969-5_17
6. Atabaki, A.H., Moazeni, S., Pavanello, F., Gevorgyan, H., Notaros, J., Alloatti, L., Wade, M.T., Sun, C., Kruger, S.A., Meng, H., Al Qubaisi, K., Wang, I., Zhang, B., Khilo, A., Baiocco, C.V., Popović, M.A., Stojanović, V.M., Ram, R.J.: Integrating photonics with silicon nanoelectronics for the next generation of systems on a chip. Nature **556**(7701), 349–354 (Apr 2018). https://doi.org/10.1038/s41586-018-0028-z, https://doi.org/10.1038/s41586-018-0028-z
7. Bach, E.: How to generate factored random numbers. SIAM J. Comput. **17**(2), 179–193 (1988). https://doi.org/10.1137/0217012, https://doi.org/10.1137/0217012
8. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) CRYPTO'86. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (Aug 1987). https://doi.org/10.1007/3-540-47721-7_24
9. Bernstein, D.J., Sorenson, J.P.: Modular exponentiation via the explicit Chinese remainder theorem. Math. Comput. **76**(257), 443–454 (2007). https://doi.org/10.1090/S0025-5718-06-01849-7, https://doi.org/10.1090/S0025-5718-06-01849-7
10. Blum, M.: Coin flipping by telephone. In: Proc. IEEE Spring COMPCOM. pp. 133–137 (1982)

11. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 757–788. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_25
12. Boneh, D., Franklin, M.K.: Efficient generation of shared RSA keys (extended abstract). In: Kaliski Jr., B.S. (ed.) CRYPTO'97. LNCS, vol. 1294, pp. 425–439. Springer, Heidelberg (Aug 1997). https://doi.org/10.1007/BFb0052253
13. Brent, R.P., Kung, H.T.: A regular layout for parallel adders. IEEE Trans. Computers **31**(3), 260–264 (1982). https://doi.org/10.1109/TC.1982.1675982, https://doi.org/10.1109/TC.1982.1675982
14. Brent, R.P., Rung, H.: A systolic algorithm for integer gcd computation. In: 1985 IEEE 7th Symposium on Computer Arithmetic (ARITH). pp. 118–125. IEEE (1985)
15. Buterin, V.: Randao++. https://redd.it/4mdkku (2017)
16. Chen, M., Cohen, R., Doerner, J., Kondi, Y., Lee, E., Rosefield, S., shelat, a.: Multiparty generation of an RSA modulus. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part III. LNCS, vol. 12172, pp. 64–93. Springer, Heidelberg (Aug 2020). https://doi.org/10.1007/978-3-030-56877-1_3
17. Cline, D., Dryja, T., Narula, N., CommitO: Clockwork: An exchange protocol for proofs of non front-running (2020)
18. Coppersmith, D., Shparlinski, I.: On polynomial approximation of the discrete logarithm and the diffie—hellman mapping. Journal of Cryptology **13**, 339–360 (2000)
19. De Feo, L., Masson, S., Petit, C., Sanso, A.: Verifiable delay functions from supersingular isogenies and pairings. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019, Part I. LNCS, vol. 11921, pp. 248–277. Springer, Heidelberg (Dec 2019). https://doi.org/10.1007/978-3-030-34578-5_10
20. Deb, S., Kannan, S., Tse, D.: PoSAT: Proof-of-work availability and unpredictability, without the work. In: Borisov, N., Díaz, C. (eds.) FC 2021, Part II. LNCS, vol. 12675, pp. 104–128. Springer, Heidelberg (Mar 2021). https://doi.org/10.1007/978-3-662-64331-0_6
21. Dickman, K.: On the frequency of numbers containing prime factors of a certain relative magnitude. Arkiv for matematik, astronomi och fysik **22**(10), A–10 (1930)
22. Dobson, S., Galbraith, S.D., Smith, B.A.: Trustless unknown-order groups. ArXiv **abs/2211.16128** (2022), https://api.semanticscholar.org/CorpusID:236932351
23. Drake, J.: Minimal vdf randomness beacon. https://ethresear.ch/t/minimal-vdf-randomness-beacon/3566 (2018)
24. Earle, J.: Latched carry-save adder. IBM Technical Disclosure Bulletin **7**(10), 909–910 (1965)
25. Fisch, B.: Tight proofs of space and replication. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 324–348. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17656-3_12
26. Gordon, D.M.: Discrete logarithms in *GF(P)* using the number field sieve. SIAM J. Discret. Math. **6**(1), 124–138 (1993). https://doi.org/10.1137/0406010, https://doi.org/10.1137/0406010
27. Hazay, C., Mikkelsen, G.L., Rabin, T., Toft, T., Nicolosi, A.A.: Efficient RSA key generation and threshold paillier in the two-party setting. Journal of Cryptology **32**(2), 265–323 (Apr 2019). https://doi.org/10.1007/s00145-017-9275-7
28. Herold, G., Kadianakis, G., Khovratovich, D., Maller, M., Simkin, M., Sanso, A., Zapico, A., Zhang, Z.: Statement regarding the public report on the anal-

ysis of minroot. https://ethresear.ch/t/statement-regarding-the-public-report-on-the-analysis-of-minroot/16670 (Sep 2023)
29. Khovratovich, D., Maller, M., Tiwari, P.R.: MinRoot: Candidate sequential function for ethereum VDF. Cryptology ePrint Archive, Report 2022/1626 (2022), https://eprint.iacr.org/2022/1626
30. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 357–388. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63688-7_12
31. Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive zero-knowledge arguments from folding schemes. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part IV. LNCS, vol. 13510, pp. 359–388. Springer, Heidelberg (Aug 2022). https://doi.org/10.1007/978-3-031-15985-5_13
32. Lenstra, A.K., Wesolowski, B.: A random zoo: sloth, unicorn, and trx. Cryptology ePrint Archive, Report 2015/366 (2015), https://eprint.iacr.org/2015/366
33. Lenstra, A.K., Wesolowski, B.: Trustworthy public randomness with sloth, unicorn, and trx. Int. J. Appl. Cryptogr. **3**(4), 330–343 (2017). https://doi.org/10.1504/IJACT.2017.10010315, https://doi.org/10.1504/IJACT.2017.10010315
34. Lenstra, H.W.: Factoring integers with elliptic curves. Annals of Mathematics **126**(3), 649–673 (1987), http://www.jstor.org/stable/1971363
35. Mahmoody, M., Moran, T., Vadhan, S.P.: Publicly verifiable proofs of sequential work. In: Kleinberg, R.D. (ed.) ITCS 2013. pp. 373–388. ACM (Jan 2013). https://doi.org/10.1145/2422436.2422479
36. Mahmoody, M., Smith, C., Wu, D.J.: Can verifiable delay functions be based on random oracles? In: Czumaj, A., Dawar, A., Merelli, E. (eds.) ICALP 2020. LIPIcs, vol. 168, pp. 83:1–83:17. Schloss Dagstuhl (Jul 2020). https://doi.org/10.4230/LIPIcs.ICALP.2020.83
37. Medley, L., Loe, A.F., Quaglia, E.A.: Sok: Delay-based cryptography. In: 36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023. pp. 169–183. IEEE (2023). https://doi.org/10.1109/CSF57540.2023.00028, https://doi.org/10.1109/CSF57540.2023.00028
38. Montgomery, H.L., Vaughan, R.C.: Multiplicative number theory I: Classical theory. No. 97, Cambridge university press (2007)
39. Pietrzak, K.: Simple verifiable delay functions. In: Blum, A. (ed.) ITCS 2019. vol. 124, pp. 60:1–60:15. LIPIcs (Jan 2019). https://doi.org/10.4230/LIPIcs.ITCS.2019.60
40. Pohlig, S.C., Hellman, M.E.: An improved algorithm for computing logarithms over gf(p) and its cryptographic significance (corresp.). IEEE Trans. Inf. Theory **24**(1), 106–110 (1978). https://doi.org/10.1109/TIT.1978.1055817, https://doi.org/10.1109/TIT.1978.1055817
41. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Technical Report, Massachusetts Institute of Technology (1996)
42. Rotem, L., Segev, G.: Generically speeding-up repeated squaring is equivalent to factoring: Sharp thresholds for all generic-ring delay functions. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part III. LNCS, vol. 12172, pp. 481–509. Springer, Heidelberg (Aug 2020). https://doi.org/10.1007/978-3-030-56877-1_17
43. Savage, J.E.: Models of computation, vol. 136. Addison-Wesley Reading, MA (1998)
44. Schindler, P., Judmayer, A., Hittmeir, M., Stifter, N., Weippl, E.R.: RandRunner: Distributed randomness from trapdoor VDFs with strong uniqueness. In: NDSS 2021. The Internet Society (Feb 2021)

45. Shamir, A.: Factoring large numbers with the Twinkle device (extended abstract). In: Koç, Çetin Kaya., Paar, C. (eds.) CHES'99. LNCS, vol. 1717, pp. 2–12. Springer, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48059-5_2
46. Shani, B.: A note on isogeny-based hybrid verifiable delay functions. Cryptology ePrint Archive, Report 2019/205 (2019), https://eprint.iacr.org/2019/205
47. Shanks, D.: Class number, a theory of factorization, and genera. In: Proc. Symp. Math. Soc., 1971. vol. 20, pp. 415–440 (1971)
48. Shparlinski, I.: Number theoretic methods in cryptography: Complexity lower bounds, vol. 17. Birkhäuser (2012)
49. Sorenson, J.: Polylog depth circuits for integer factoring and discrete logarithms. Information and Computation **110**(1), 1–18 (1994)
50. Sorenson, J.: Two fast GCD algorithms. J. Algorithms **16**(1), 110–144 (1994). https://doi.org/10.1006/jagm.1994.1006, https://doi.org/10.1006/jagm.1994.1006
51. Sreedhar, K., Horowitz, M., Torng, C.: A fast large-integer extended GCD algorithm and hardware design for verifiable delay functions and modular inversion. IACR TCHES **2022**(4), 163–187 (2022). https://doi.org/10.46586/tches.v2022.i4.163-187
52. StarkWare: Presenting: VeeDo. https://medium.com/starkware/presenting-veedo-e4bbff77c7ae (june 2020)
53. Supranational LLC: MinRoot VDF Hardware Engine (2022), https://github.com/supranational/minroot_hardware
54. Supranational LLC: Minroot ASIC Driver (2023), https://github.com/supranational/minroot_driver
55. Supranational LLC: MinRoot VDF ASIC (apr 2023), private presentation
56. Valiant, L.G.: A scheme for fast parallel communication. SIAM J. Comput. **11**(2), 350–361 (1982). https://doi.org/10.1137/0211027, https://doi.org/10.1137/0211027
57. Wallace, C.S.: A suggestion for a fast multiplier. IEEE Trans. Electron. Comput. **13**(1), 14–17 (1964). https://doi.org/10.1109/PGEC.1964.263830, https://doi.org/10.1109/PGEC.1964.263830
58. Wang, P.S.: A p-adic algorithm for univariate partial fractions. In: Wang, P.S. (ed.) Proceedings of the Symposium on Symbolic and Algebraic Manipulation, SYMSAC 1981, Snowbird, Utah, USA, August 5-7, 1981. pp. 212–217. ACM (1981). https://doi.org/10.1145/800206.806398, https://doi.org/10.1145/800206.806398
59. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part III. LNCS, vol. 11478, pp. 379–407. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17659-4_13
60. Wiener, M.J.: The full cost of cryptanalytic attacks. Journal of Cryptology **17**(2), 105–124 (Mar 2004). https://doi.org/10.1007/s00145-003-0213-5

A Other Low-latency algorithms

A.1 Algorithm using Subgroups of $\mathbb{F}_p^*$ (by Anne Canteaut)

For any divisor d of $(p-1)$, we denote by G_d the multiplicative subgroup of $\mathbb{F}_p^*$ of order d, i.e.,

$$G_d := \{x \in \mathbb{F}_p^* : x^d = 1\} .$$

Let $F_{[d,a]}$ be the a-th root in G_d, i.e.

$$F_{[d,a]}(x) = y \text{ if and only if } y^a = x, \text{ for any } x \in G_d \ .$$

In other words,

$$F_{[d,a]}(x) = x^e \text{ with } ea \equiv 1 \bmod d \ .$$

Then, $F_{[p-1,a]}$ can be computed from $F_{[d,a]}$ and $F_{[\overline{d},ad]}$, when $\overline{d} = (p-1)/d$ is coprime with d. Namely, for any $x \in \mathbb{F}_p^*$, we have

$$F_{[p-1,a]}(x) = F_{[\overline{d},ad]}(x^d) \times F_{[d,a]}\left(\frac{x}{\left(F_{[\overline{d},ad]}(x^d)\right)^a}\right) \ . \tag{3}$$

Indeed, if $\gcd(d, \overline{d}) = 1$, $F_{[p-1,a]}(x)$ can be uniquely decomposed as the product of two elements, g_1 in G_d and g_2 in $G_{\overline{d}}$. Then, by definition

$$x = F_{[p-1,a]}(x)^a = g_1^a g_2^a \ ,$$

implying that

$$x^d = g_2^{ad} \ .$$

It follows that

$$g_2 = F_{[\overline{d},ad]}(x^d) \ .$$

Moreover

$$g_1^a = \frac{x}{g_2^a} = x \times \left(F_{[\overline{d},ad]}(x^d)\right)^{-a} \ ,$$

leading to (3).

It follows that, up to a few operations, computing $F_{[p-1,a]}(x)$ boils down to computing (possibly in parallel) both $F_{[\overline{d},ad]}$ and $F_{[d,a]}$, followed by an additional exponentiation by d. This observation can be used in two different manners. First, basic TMTO algorithms can be improved at the price of an exponentiation by d, i.e., an additional cost of around $\log_2(d)$ in the latency. A second possible direction would be to investigate whether computing $F_{[\overline{d},ad]}$ could be significantly easier than the original problem.

Time-Memory Trade-off An interesting point is that the previous observation enables to divide the computation into two parts: computing a root in G_d and computing a root in $G_{\overline{d}}$. We set $d < \overline{d}$.

In the following, we assume that that d is smaller than the available memory size so that computing $T_{[d,a]}$ can be done by precomputing a lookup table. Otherwise, the second step of the on-line phase requires another randomization and needs to be distributed among several processors.

Precomputation

- Build a table T_d with all pairs (z, z^a), $z \in G_d$, indexed by the values of z^a.

– Build a table $T_{\overline{d}}$ with triples $(z, \frac{1}{z^a}, z^{ad})$ for M values of $z \in G_{\overline{d}}$. This table is indexed by z^{ad}.

The on-line phase then consists of the following two steps.

Find the component in $G_{\overline{d}}$
- $u \leftarrow x^d$
- On each processor, do:
  - Pick $(r, \frac{1}{r^a}, r^{ad})$ in $T_{\overline{d}}$.
  - $y \leftarrow r^{ad} \times u$
  - If y is in $T_{\overline{d}}$ then return $(y_2, v, y) = T_{\overline{d}}[y]$

Find the component in G_d:
- $z \leftarrow x \times r^a \times v$
- $g_1 = T_d[z]$
- return $\frac{g_1 \times y_2}{r}$

This algorithm costs $M \log_2(ad) + d$ precomputation time, $(M + d)$ memory.

The number of processors is $\frac{p-1}{dM}$, and the number of operations to be performed is one multiplication and one table lookup on each processor, as well as one exponentiation by d, a few multiplications and one additional table lookup on a single processor. This corresponds to a latency of $\log_2(d)$ plus a small constant.

It is worth noting that for $d = 1$, this corresponds to the usual TMTO algorithm, where M values of $F_{[p-1,a]}$ are precomputed. Using subgroups allows to divide the number of processors (or the memory) by a factor d at a cost $\log_2(d)$ of latency.

In particular, if there is a small factor d of $p-1$, we obtain an algorithm with fewer than 2^{128} processors, breaking the claim of 128-bit security. Since p has a particular shape $p = 2^{32}q + 1$, we choose $d = 2^{32}$ and obtain an algorithm with latency slightly higher than 32, using 2^{96} processors.

Algorithm 10: Using precomputation and subgroups to evaluate $\sqrt[a]{x}$

$T = 48$ $\quad \#CPU = 2^{96}$ $\quad M = 2^{128}$ $\quad$ speedup : 5.3

Computing $F_{[\overline{d},ad]}$ In general, there is no particular algorithm to speed up the computation of $F_{[\overline{d},ad]}(y)$ in the previous algorithm. For instance, the probability that the value $y = r^{ad}x^d$ is smooth is not higher for a random element. It then seems difficult to combine the use of subgroups with some other algorithm exploiting the fact that computing a-th root is easier for inputs having a specific form (e.g. for smooth integers).

However, for a given value of p, it should be checked that there is no divisor d of $p-1$ such that computing $F_{[\overline{d},ad]}$ is much easier than expected. One condition is that the corresponding e such that $ead \equiv 1 \bmod \overline{d}$ has to be close to $\overline{d}$. For the MinRoot prime p, no d gives an anomalously small e.

A.2 Algorithm Using the Chinese Remainder Theorem (by Itai Dinur)

Given $x \in \mathbb{F}_p$ for $p < 2^{256}$, assume we want to compute $x^d \bmod p$ for $d \in \mathbb{Z}_{p-1}$ in minimal parallel time and limited number of processors. We demonstrate how to apply a variant of the CRT-based algorithm by Bernstein and Sorenson [9] for the concrete value of $d = 32 = 2^5$. We note that our algorithm is slightly different from that of [9]. For example, unlike [9], we do not use the explicit form of the Chinese remainder theorem, but this does not seem to have a significant impact in our computational model.

As the advantage of the algorithm over a standard square-and-multiply algorithm is small at best, it heavily depends on the computational model. Using the assumptions above, a standard square-and-multiply algorithm that computes x^{32} requires 5 time units (5 squarings).

Definitions Let $y = x^{32}$, over the integers. We have $0 \leq y \leq 2^{256 \cdot 32} = 2^{8192}$. Let d be the smallest value such that $q_1 \cdot q_2 \cdot \ldots \cdot q_d \geq 2^{8192}$, where $q_1 < q_2 < \ldots < q_d$ are the first d prime numbers. We have $d = 759 < 2^{10}$ and $q_d = 5783 < 2^{13}$. Denote

$$Q = q_1 \cdot q_2 \cdot \ldots \cdot q_d.$$

Our aim is to do computation in the CRT basis defined by Q. Therefore, we define $Q_i = Q/q_i$, $M_i = 1/Q_i \bmod q_i$ (modular inverse), and $y_i = y \bmod q_i$ for $1 \leq i \leq d$.

Since $y \leq Q$, we have by the Chinese remainder theorem:

$$y = \left(\sum_{i=1}^{d} y_i M_i Q_i \right) \bmod Q.$$

We define $z = \sum_{i=1}^{d} y_i M_i Q_i$, as a sum of integers.

Overview Our goal is to compute $y \bmod p = (z \bmod Q) \bmod p$. We define $k = \lfloor z/Q \rfloor$, so that $z = (z \bmod Q) + kQ$, with $z \bmod Q < Q$ and $k \in \mathbb{Z}$. Hence

$$y \bmod p = (z \bmod Q) \bmod p = (z \bmod p - kQ \bmod p) \bmod p$$

This is the main equation used by the algorithm. In the following, we describe how to compute $z \bmod p$ and $kQ \bmod p$, while the result is obtained by subtracting them and reducing modulo p.

Computing $z \bmod p$ Given x, our goal is to compute

$$z \bmod p = \sum_{i=1}^{d} y_i M_i Q_i \bmod p,$$

with $y_i = y \bmod q_i$. We define $x_i = x \bmod q_i$, and we observe that

$$y_i = y \bmod q_i = x^{32} \bmod q_i = x_i^{32} \bmod q_i.$$

We compute the term $y_i M_i Q_i \bmod p$ as follows. We begin by computing $x_i = x \bmod q_i$ by looking at the binary representation of x and making use of precomputed values of $2^j \bmod q_i$ for $j = 0, \ldots, 255$. Thus, computing x_i requires 256 additions $\bmod q_i$ (with word size about 12 bits), which can be done in in parallel-time 1 using our assumptions, with a circuit that is smaller than a multiplier. Alternatively, we can use larger precomputed tables. For example, if we use tables of size 2^8 (per 8 bits of x), we can compute x_i using 32 processors but this doesn't reduce the latency under our assumptions (we assume that a table lookup takes the same time as adding 256 values).

Once we have computed x_i, we can use precomputed tables that map x_i to $y_i M_i Q_i \bmod p$. Since $x_i < q_i$, each table has size of at most $q_d < 2^{13}$. We assume that the tables are small enough to be implemented with unit delay, and we consider that the circuit size for one table corresponds to one processor. Finally, $z \bmod p = \sum_{i=1}^{d} y_i M_i Q_i \bmod p$ can be computed in unit time.

Overall, using precomputed tables as above, computing $z \bmod p$ can be done using about $d \approx 2^{10}$ processors in 3 units of parallel time.

Computing $kQ \bmod p$ Recall that $z = \sum_{i=1}^{d} y_i M_i Q_i$. Since $0 \leq M_i Q_i < Q$, we have

$$0 \leq z < Q \cdot \sum_{i=1}^{d} y_i \leq d \cdot q_d \cdot Q \leq 2^{23} \cdot Q.$$

Since $k = \lfloor z/Q \rfloor$, we have $0 \leq k < 2^{23}$.

Moreover we can estimate z/Q to a precision of 20 bits (for example), and then round it down to the nearest integer. This may introduce errors, which are rare (assuming the input is uniform). However, in the setting of VDFs, where (some) computations can be efficiently checked, it is easy to deal with the rare erroneous computations (see Section 7.1).

More specifically, for each $i \leq d$, after computing $x_i = x \bmod q_i$ as above, we use a precomputed table that maps x_i to the value $(y_i M_i Q_i)/Q$, up to a precision of $20 + \log_2(d) = 30$ bits (overall, we use $13 + 30 = 43$-bit values). We then add these d values in parallel and round the result down to the nearest integer to estimate k. Finally, we use the precomputed value of $Q \bmod p$ and compute $kQ \bmod p$.

Note that the addition of 2^{10} values introduces an additional error term, but it is unlikely to propagate beyond 10 bits.

After the computations of $x_i = x \bmod q_i$, the computation of $kQ \bmod p$ can be performed in parallel to that of $z \bmod p$. Thus, the only overhead in parallel time is caused by the computation of $k \cdot Q \bmod q$, with unit latency. Since $Q \bmod q$ is a constant and k is small, the multiplication can be done with lookup tables, but this does not reduce the latency in our model.

Total Cost After computing $z \bmod p$ and $kQ \bmod p$, it takes unit time to compute the final result. However, we can avoid this additional latency by adding $z \bmod p$ to the result before the final reduction in the modular multiplication $k \cdot Q \bmod q$ (using a multiply-and-add operation).

Overall, the total time is estimated to be 4 units, which improves upon the standard square-and-multiple algorithm with latency 5 (with $d = 32$). The number of processors required is about 2^{11}.

Algorithm 11: Using CRT to evaluate x^{32}

$T = 4$	$\#CPU = 2^{11}$	speedup : 5/4

Variants

CRT coordinates. We may avoid the initial reductions $x_i = x \bmod q_i$ in consecutive computations of exponentiations by "remaining in CRT coordinates", i.e., performing all intermediate computations modulo q_j for each j. However, this only saves the initial modular reductions and requires about $d \approx 2^{10}$ times more processors.

Trade-Off by changing d. We can use the same approach to compute $x^d \bmod p$ for other values of d. In general, choosing a smaller value of d will result in a smaller gain compared to the standard square-and-multiple algorithm in our computational model. Yet, a smaller value of d requires fewer processors, smaller lookup tables and smaller fan-in/fan-out, and hence the cost model may be more realistic. Choosing a larger value of d has the opposite effect (in particular, we quickly obtain lookup tables that are too large for a combinatorial implementation and must be stored in RAM).

A.3 Extensions to Multiple Rounds of low-degree MinRoot

We now consider extensions to the attack from section 6. Let us first consider the evaluation of several rounds of a MinRoot variant, where the round function x^e uses a small e rather than $e = 1/a \bmod p - 1$ with small a.

Note that when used as a subroutine of a large-power exponentiation via e-th power and multiply, we need to beat a latency of (non-rounded) $\log_2 e$, whereas if the round function itself uses $x \mapsto x^e$ for small e directly, we need to beat the latency of the standard implementation of $x \mapsto x^e$. This may differ if e is not a power of 2.

For such a round function, given input (x, y), each of the two outputs after r rounds would be a degree e^r polynomial $P(x, y)$. This polynomial can be written in a general form:

$$P(x, y) = \sum_{k=0}^{e^r} \sum_{i=0}^{k} A_{ki} x^i y^{k-i}$$

In practice it seems most of the terms are present and there are a bit less than $e^r(e^r-1)/2$ terms. (Without the counters there would be only odd terms if e is odd.)

Caveat: the constants A_{ki} will depend on the round counter, so we need a clever way to precompute/store them. They would be themselves polynomials of degree at most e^{r-2} in the round counter (it does not participate in the non-linear part of the 1st round and is only added at the end of the last round). In the naive implementation they can be all stored in tables for all the 2^{40} steps of the VDF computation. There might be a smarter recurrent way to compute them on the fly.

Attack Complexity There is clearly a trade-off between the number of processors and the size of tables which is governed by the number s of shares and the number of rounds r, since the size of tables is exponential in $\frac{\log_2 p}{s}$ and r, and the number of processors is exponential in s and r (it can be very roughly approximated as e^{rs}).

To give a concrete preliminary example, we consider $e = 3$ (cubing function); the standard implementation has a latency of 3 units per cycle (2 multiplications and 1 addition).

With $s = 8$, tables would take $T = 3^r 2^{37}$ bytes. There are at most $e^{2r}/2$ terms in the polynomial; each term is the product of an x monomial and a y monomial and each monomial requires at most $\binom{3^r+7}{7}$ processors for a low-latency evaluation; therefore the number of processors would be $P = \binom{3^r+7}{7}\frac{3^{2r}}{2}$. So for $r = 8$ get: $T \approx 2^{50}$ bytes, $P = \binom{6568}{7} \cdot \frac{9^8}{2} \approx 2^{76+24} = 2^{100}$. The latency is 1 table lookup and 3 multiplications (to compute one share of one monomial), 10 units to sum 2^{76} shares, 1 multiplication (between the x monomial and the y monomial), and 3 units for the sum of 2^{24} terms. The total latency is estimated at 23 units, instead of 24 units for the standard implementation.

Recursive Approach Observe that the large memory cost comes from lookups to compute maps of the form $x \mapsto A \cdot x^i$, where x is from a set of the form $x \in \{0, B, 2B, \ldots\}$ of size $\sqrt[s]{p}$. The way we split elements into shares can be recursively applied to sets of this form to give shares from an even smaller domain.

By doing this (ex. 3 stages, bottom stages splitting into 3 shares each time), we can trade-off a bit of latency for large gains in T. For example with two layers of recursion $l = 2$ and $s = 3$ (optimal numbers of shares are $2^m - 1$ which allows to do one layer in m multiplicative steps) we will need $T = e^r 2^{29+5} = e^r 2^{34}$ bytes and $P = (\frac{e^{2r}}{2})^3 = \frac{e^{6r}}{2}$ processors. For $r = 12$, we have $T = 2^{48}$ bytes, $P = 2^{114}$, and a latency of $6 + 2 + 5 + 2 + 5 + 1 + 5 = 26$, instead of 36.

A rough comparison of the various techniques is shown in Table 4 (a copy of Table 3). Some additional observations:

– There are lots of redundant, overlapping calculations, so there should be possible savings in terms of tables and processors.

Table 4. Comparison of highly parallel low latency computation with standard VDF (only counting the multiplications in the latency).

Metric	Compressed	Recursive with l layers	Standard VDF
Tables (in bits)	$e^r 2^{\frac{\log_2 p}{s}} \log_2 p$	$e^r 2^{\frac{\log_2 p}{s^l}} \log_2 p$	0
Processors	$\binom{e^r+s-1}{s-1}\frac{e^{2r}}{2}$	$\binom{e^r+s-1}{s-1}^l \frac{e^{2r}}{2}$	1 or 2
Latency (in $\log_2 p$ bit MUL)	$2+\lceil\log_2(s)\rceil$	$2+l\lceil\log_2(s)\rceil$	$\lceil\log_2(e)\rceil \cdot r$

– Dependence of coefficients A_{ki} on the counters needs to be taken into account. This should be done on the fly by the processor responsible for the specific monomial.
– It seems there is no big difference between Feistel or MISTY-like round functions in terms of the number of monomials they can generate. But it does help to perform the first step $x_1 = x_0 + y_0$ before the rest of the computation, gaining one round in terms of monomials.