Analysis of MinRoot

Gaëtan Leurent, Bart Mennink, Krzysztof Pietrzak, Vincent Rijmen, Alex Biryukov, Benedikt Bunz, Anne Canteaut, Itai Dinur, Yevgeniy Dodis, Orr Dunkelman, et al.

Analysis of MinRoot:
Public report
(requested by Ethereum Foundation)

Report written by Gaëtan Leurent[1], Bart Mennink[2], Krzysztof Pietrzak[3], Vincent Rijmen[4,17],

based on the collaboration with Alex Biryukov[5], Benedikt Bunz[6], Anne Canteaut[1], Itai Dinur[7], Yevgeniy Dodis[8], Orr Dunkelman[9], Ben Fisch[10], Ilan Komargodski[11], Nadia Heninger[12], Maria Naya Plasencia[1], Leo Perrin[1], Christian Rechberger[13], Gil Segev[10], Martijn Stam[14], Stefano Tessaro[15], Benjamin Wesolowski[16],

and contributions from Mike Schaffstein[18], and Ethereum Foundation members Dankrad Feist[19], Gottfried Herold[19], Antonio Sanso[19], Mark Simkin[19], and Dmitry Khovratovich[19]

[1]INRIA (France)
[2]Radboud University, Netherlands
[3]IST (Austria)
[4]KU Leuven (Belgium)
[5]University of Luxembourg, Luxembourg
[6]Espresso Systems, USA
[7]Ben Gurion University, Israel
[8]New York University, USA
[9]University of Haifa, Israel
[10]Yale, USA
[11]Hebrew University of Jerusalem, Israel
[12]UCSD, USA
[13]TU Graz, Austria
[14]Simula UiB, Norway
[15]University of Washington, USA
[16]ENS de Lyon, CNRS, UMPA, UMR 5669, Lyon, France
[17]University of Bergen, Norway
[18]Supranational, USA
[19]Ethereum Foundation

September 18, 2023

Contents

1 Preface **2**

2 Attack 1 Group **3**
2.1 Introduction . . . 3
2.2 Low-latency Evaluation of Low-degree Exponentiation . . . 5
2.3 Low-latency Evaluation of Homomorphism . . . 10
2.4 Optimizing the Smoothness Algorithm . . . 15
2.5 Application to VDF Constructions . . . 22
2.6 Practical Issues . . . 23
2.7 Possible Tweaks . . . 24
2.8 Other Remarks . . . 24

3 Attack 2 Group **26**
3.1 Introduction . . . 26
3.2 Masking-based attacks . . . 26
3.3 Costs . . . 28
3.4 Additional observations . . . 28

4 Theory 1 Group **30**
4.1 Introduction . . . 30
4.2 Sequentiality . . . 32

5 Theory 2 Group **37**
5.1 Introduction . . . 37
5.2 Concrete security definitions . . . 37

Chapter 1

Preface

Between April 28th and 30th, 2023, the Ethereum Foundation invited a group of researchers to conduct an initial analysis of the candidate sequential function MinRoot [KMT22]. The purpose of this gathering was to collectively delve into the intricacies of MinRoot and assess its potential implications for the Ethereum ecosystem. Preliminary analysis was conducted by Gaëtan Leurent, Maria Naya Plasencia, and Stefano Tessaro. At the event, the researchers were divided into three groups, each tasked with different aspects of MinRoot's evaluation. This report serves as a comprehensive joint summary, presenting the culmination of the researchers' intensive efforts during the event. Each chapter within this report has been thoughtfully composed by a researcher who is not affiliated with the Ethereum Foundation, thereby ensuring impartiality and transparency in the evaluation of MinRoot. The analysis presented in this report reveals valuable insights into the strengths, weaknesses, and areas of improvement for MinRoot. With this report, the Ethereum Foundation aims to foster greater understanding and discussion among stakeholders, inviting further exploration and scrutiny of MinRoot's capabilities.

Overview of report. This paper is organized as follows. Chapter 2 focuses on the security analysis conducted by Attack Group 1. Chapter 3 provides an overview of the security findings from the smaller Attack Group 2. Chapter 4 digs into the theoretical framework of MinRoot, offering a comprehensive analysis of its sequential function conducted by Theory Group. Finally, Chapter 5 suggests another way of looking into the concrete security of MinRoot-like VDFs.

Chapter 2

Attack 1 Group

Written by Gaëtan Leurent

2.1 Introduction

2.1.1 MinRoot

MinRoot is a proposal for a Verifiable Delay Function (VDF) designed by the Ethereum Foundation [KMT22]. It iterates a simple round function, using a root in a finite field, as shown in Algorithm 1. We denote the internal state of MinRoot as (u, v), and the round constant as i.

Parameters.

- p: prime number defining the field ($p = 2^{254} + 2^{32} \cdot \texttt{0x224698fc094cf91b992d30ed} + 1$)
- a: small exponent with $p \not\equiv 1 \bmod a$ ($a = 5$ with the given p)
- t: number of rounds (typically in the order of 2^{40})

Implementation. A standard implementation of MinRoot 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 $\sqrt[a]{x} = x^e$. Concretely, with $a = 3$ or $a = 5$, we have:

$$\sqrt[3]{x} = x^{\frac{2p-1}{3}} \quad (\text{since } p \equiv 2 \bmod 3) \qquad \sqrt[5]{x} = \begin{cases} x^{\frac{4p-3}{5}} & \text{if } p \equiv 2 \bmod 5 \\ x^{\frac{2p-1}{5}} & \text{if } p \equiv 3 \bmod 5 \\ x^{\frac{3p-2}{5}} & \text{if } p \equiv 4 \bmod 5 \end{cases}$$

Exponentiation to the power e with the square and multiply algorithm requires $\lg(e) \approx \lg(p)$ squarings. There are techniques to reduce the number of multiplications (such as the sliding-window method or

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

Algorithm 1: MinRoot VDF.

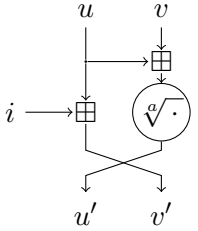

Figure 2.1: MinRoot round function.

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 $\lg(p)$ squarings (254 squarings with the proposed parameters), and the delay of MinRoot is essentially $t \lg(p)$ squarings (using two processors).

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

Security claim. The security claim is that MinRoot is a sequential function. Informally, this means that MinRoot cannot be computed faster by using parallelism. There should be a lower bound to the delay required to evaluate the function, and the standard implementation with a delay of $t \lg(p)$ squarings should be close to this lower bound (up to a small constant factor).

This can be understood as 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 t rounds faster than by iterating them).

Formally, the MinRoot designers claim "128 bits of multitarget VDF security". This means that a parallel attacker with up to 2^{128} processors should not able to evaluate the function with less than half of the latency of the standard implementation, even with a shared memory that can be accessed in constant time by all processors.

2.1.2 Notations and Assumptions

We focus on the latency of evaluating MinRoot 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 MinRoot naturally works with modular values in $\mathbb{F}_p$, in this report, we sometimes consider the values as integer instead; hopefully it 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.

We use $\lg(x)$ to denote the base-2 logarithm, and $\log(x)$ for the natural logarithm.

Latency Assumptions. We have the following results on low-latency arithmetic operation:

Integer addition: Addition of two n-bit integers has a latency of $\mathcal{O}(\lg(n))$ using a carry look-ahead adder (*e.g.* the Brent-Kung adder [BK82]).

Addition of k n-bit integers has a latency of $\mathcal{O}(\lg(k)+\lg(n))$, using a tree of carry-save adders [Ear65], followed by a carry look-ahead adder.

Integers multiplication: Multiplication of two n-bit integers has a latency of $\mathcal{O}(\lg(n))$ using a tree of carry-save adders (*e.g.* a Wallace tree [Wal64]) followed by a carry look-ahead adder.

Modular reduction: Modular reduction of a $2n$-bit value modulo an n-bit value has a latency of $\mathcal{O}(\lg(n))$. For instance, the Barett reduction [Bar86] computes the reduction using integer division by a constant, which is computed using an integer multiplication.

The results above implies that modular addition and modular multiplication in $\mathbb{F}_p$ have a latency of $\mathcal{O}(\lg(\lg(p)))$. For simplicity, we assume that those operation have the same latency, and consider it as one unit of time (for practical purpose this time unit is estimated to be 0.9 ns in an ASIC implementation).

Adding up to $\lg(p) \approx 256$ values (integers smaller than p or modular values in $\mathbb{F}_p$) also has latency $\mathcal{O}(\lg(\lg(p)))$; we assume this also corresponds to one unit.

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

Table lookup in a table with k entries has a latency $\mathcal{O}(\lg(k))$ when implemented as a combinatorial circuit. For small tables (up to $\lg(p) \approx 256$ entries), we consider this also has 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 $\lg(p)$ value and table lookup with with up to $\lg(p)$ values are assumed to require one processor.

2.1.3 Our results

We explore several approaches to compute MinRoot in parallel with lower latency than the standard implementation. The resulting algorithms achieve a modest modest gain in latency, but require a massive number of parallel processors; for instance one of our most interesting results (Section 2.4.5) achieves a speedup factor of 20 (ignoring communication latency) using 2^{29} processors and a memory of size 2^{40}. These results have very little interest outside of VDF analysis, and there is no much literature studying this problem. Moreover, our algorithms require communication between a large number of processors which is likely to have a significant latency in practice. Some of our results clearly break the security claims of MinRoot, but it is unclear whether they can be implemented in practice.

We start by exploring several ideas for low-degree functions in Section 2.2, and for homomorphisms in Section 2.3. Our best algorithm is a smoothness-based algorithm in Section 2.3.3, that uses the same basic ideas as a paper for Adleman and Kompella [AK88]. We propose various optimizations to this algorithm in Section 2.4. We show how this impacts several VDF constructions in Section 2.5. Finally, in Section 2.6 we briefly discuss practical issues to implement our algorithms and in Section 2.7 we discuss various modifications to MinRoot to improve its security.

Even if the practicality of our algorithms is debatable, they clearly show that computing a root $\sqrt[a]{x}$ in $\mathbb{F}_p$ is not a sequential operation. Several previous work [LW15, BBBF18, Sta20, KMT22] assume that there is no parallel algorithm with lower latency than the square and multiply algorithm with latency $\lg(p)$, but our results show that this assumption is wrong.

2.2 Low-latency Evaluation of Low-degree Exponentiation

We start the analysis by looking at methods to compute x^d with a small d in parallel faster than with the standard square and multiply. This can be directly applied to MinRoot 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 actual MinRoot: an algorithm to compute x^d with latency smaller than $\lg(d)$ squarings can speed up the square and multiply algorithm computing x^e with arbitrary e.

2.2.1 Algorithm Using the Chinese Remainder Theorem

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 [BS07] for the concrete value of $d = 32 = 2^5$. We note that our algorithm is slightly different from that of [BS07]. For example, unlike [BS07], 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 $q_1 \cdot q_2 \ldots \cdot q_d \geq 2^{8192}$, 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

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

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 mod 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 takes. 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 $0 \le M_i Q_i < Q$, we have

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

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

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 MinRoot, where (some) computations can be efficiently checked, it is easy to deal with the rare erroneous computations (see Section 2.6.1).

More specifically, for each $i \le d$, 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 $20 + \lg(d) = 30$ 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 time unit 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}.

Example 1: 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).

2.2.2 Algorithm Using Shares and LUTs

Let us consider that we split each field element into s shares, 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 MinRoot-like 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 space that is much smaller than $\mathbb{F}_p$). Assuming 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. Overall, $2d$ processors are needed for this to work (to access $2d$ LUTs in parallel).

If $d \leq 4$ such a technique is not 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

- $s\binom{d+s-1}{d} = s\binom{d+s-1}{s-1}$ parallel processors,
- tables of size roughly $p^{1/s}$,
- a latency of 1 lookup, $\lg(s)$ 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 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 2.1.2, 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.

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

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

2.2.3 Extension to Multiple Rounds of low-degree MinRoot

We now 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. In such case given input (x, y) the output 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, 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 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 of shares s and the number of rounds r, since the size of tables is exponential in $\frac{|p|}{s}$ 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

It seems that doing things recursively (ex. 3 stages, bottom stages splitting into 3 shares each time) 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 2.1. Some additional observations:

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

Metric	Compressed	Recursive l	Standard VDF
Tables (in bits)	$e^r 2^{\frac{\lvert p\rvert}{s}} \lvert p\rvert$	$e^r 2^{\frac{\lvert p\rvert}{s^l}} \lvert p\rvert$	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
Latency (in $\lvert p\rvert$ bit MUL)	$2+\lceil\lg(s)\rceil$	$2+l\lceil\lg(s)\rceil$	$\lceil\lg(e)\rceil\cdot r$

Table 2.1: Comparison of highly parallel low latency computation with standard VDF (only counting the multiplications in the latency).

2.3 Low-latency Evaluation of Homomorphism

We now explore algorithms using the homomorphism property of root functions. The algorithms in this section can be applied to any function f over $\mathbb{F}_p$ with the following properties:

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

In particular it can be applied to any monomial function. In the following, we consider roots $f(x) = \sqrt[a]{x}$ in $\mathbb{F}_p$ with $p \approx 2^{256}$ as a concrete example, following the MinRoot parameters.

The root is the most expensive operation in MinRoot, and has a latency of 256 squaring in the standard implementation. Our goal is to evaluate it with smaller latency.

2.3.1 Algorithm using Precomputation

We first propose a simple algorithm using precomputation. It is similar to the precomputation attack discussed in the MinRoot proposal [KMT22], but we target the root operation instead of targeting the iterated function MinRoot, and we use a self-randomization property to make the attack successful for any given input instead of having a multi-target attack.

Precomputation

- Compute $\sqrt[a]{i}$ for $i \leq 2^{128}$
- Compute r^a, r^{-1} for a set of 2^{128} random values r

Online phase Given a challenge x, 2^{128} processor will do the following in parallel:

1. Pick one of the randomly generated r
2. Compute $y = x \cdot r^a \bmod p$
3. If $y \leq 2^{128}$, 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, it requires $\sqrt{p} = 2^{128}$ precomputations, and has a latency of two multiplications and one table lookup (8 time units using the assumptions of Section 2.1.2), using $\sqrt{p} = 2^{128}$ parallel processors. Since MinRoot claims 128 bits of security, this algorithm does not violate the claim, but shows that it is at best tight.

In practice, each of the 2^{128} processors will use a different r than can be hard-coded in the processor, together with r^a and r^{-1}. The algorithm requires a memory of size $\sqrt{p}$, but only a single processor (the one that succeeds) will access the memory for each root computation.

Example 3: Using precomputation to evaluate $\sqrt[a]{x}$

$T = 8$ $\quad\#CPU = 2^{128}$ $\quad M = 2^{128}$ $\quad$ speedup : 32

2.3.2 Algorithm using Subgroups of $\mathbb{F}_p^*$

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

$$F_{[p-1,a]}(x) = F_{[\overline{d},ad]}(x^d) \times F_{[d,a]}\left(\frac{x}{\left(F_{[\overline{d},ad]}(x^d)\right)^a}\right) . \tag{2.1}$$

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 (2.1).

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 additionnal 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 additionnal cost of around $\lg(d)$ in the latency. A second possible direction would be to investigate whether computing $F_{[\overline{d},ad]}$ could be significantly easier that the original problem.

Time-Memeory 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 $M \lg(ad) + d$ 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 additionnal table lookup on a single processor. This corresponds to a lantecy of $\lg(d)$ 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 $\lg(d)$ 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.

Example 4: Using precomputation and subgroups to evaluate $\sqrt[a]{x}$

$T = 48$	$\#CPU = 2^{96}$	$M = 2^{128}$	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 probablity 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.

2.3.3 Algorithm using Smooth Numbers

We now describe an algorithm based on smoothness, which is similar to a previous work published by Adleman and Kompella in STOC '88 [AK88], using smoothness to compute some arithmetic functions with logarithmic depth and a large number of processors. Starting with the randomization step of Section 2.3.1, 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 smallers than B, for some bound B.

Notations. We use $\pi(x)$ to denote the prime counting function ($\pi(x) \approx x/\log(x)$), and $\rho(x)$ for the Dickmann function. In particular, the probablity that an n-bit number is B-smooth is approximately $\rho(n/\lg(B))$.

The algorithm works as follow:

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 goups of processors (each group of size $\pi(B)$) will do the following in parallel (steps 4 and 5 use $\pi(B)$ processor, 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. Do 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 probablity of y being B-smooth and square-free can be approximated as $\rho(256/\lg(B)) \times 6/\pi^2$; therefore we choose $R \gg \pi^2/6\rho(256/\lg(B))$, and we obtain a high probability of success, thanks to the randomizing at step 1.

This algorithm is similar to index calculus to compute discrete logarithms. Its complexity is sub-exponential. We now discuss some optimization to make the algorithm more 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. In order to avoid the latency of computing z^a at the end, we can precompute an approximation of $\lg(q_i)$ and check that $\sum \lg(q_i) \approx \lg(y) \approx \lg(p)$ to detect when y is B-smooth.

Finally, we obtain the Algorithm of Figure 2.2, where black boxes represent processors and green boxes represent groups of processors. For simplicity, we assume that a single group of processor 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 factor and the complexity depend 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 $z_i = \sqrt[e]{q_i}$, 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

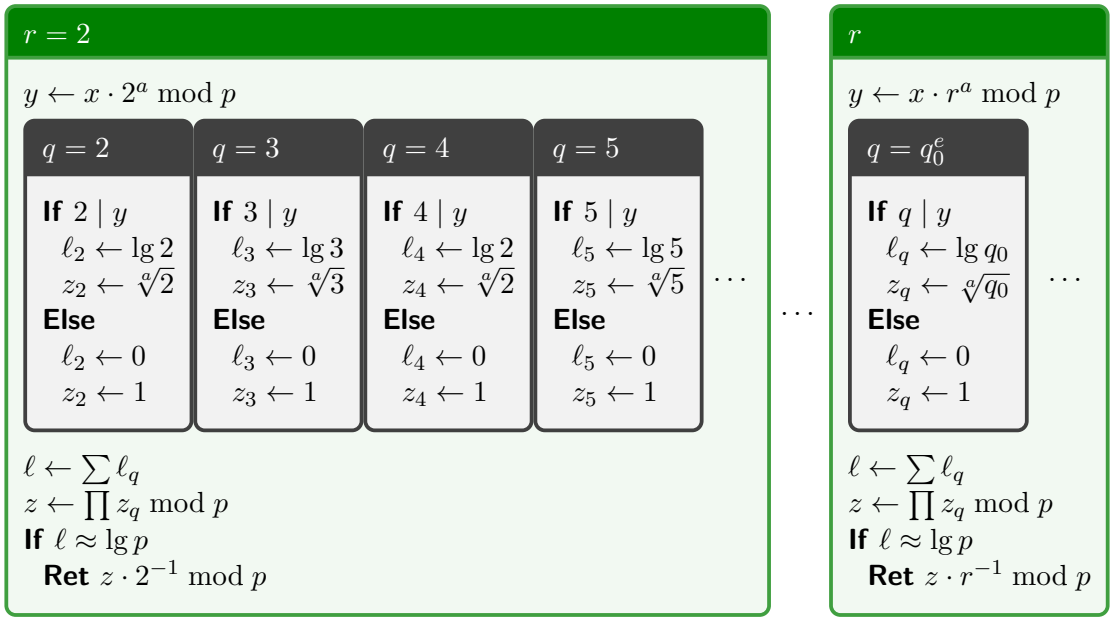

Figure 2.2: Algorithm using B-smooth numbers

$z = \prod z_i \bmod p$ by a sum $\nu = \sum \nu_i \bmod p-1$ and an exponentiation $z = g^\nu$. Using the assumptions of Section 2.1.2, the sum of $\pi(B)$ terms has latency only $\lg(\pi(B))/8$. 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. 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 succeed 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 [Bac88]. 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 ($2^{30.5} + 2^{14.1}$).

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.

Example 5: Using smoothness to evaluate $\sqrt[a]{x}$

$T = 6$ $\qquad$ $\#CPU = 2^{54.5}$ $\qquad$ speedup : 42

Alternatively, parameters can be chosen to minimize the latency:

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

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.

2.4 Optimizing the Smoothness Algorithm

We now discuss several ideas to optimize the smoothness algorithms. When combined together the number of processors required is reduced below 2^{30} (with a 256-bit prime), and we can run experiments on the full algorithm.

2.4.1 Using Almost-smooth Numbers

First, we relax the constraint of y being B-smooth to allow a single medium-size factor, in addition to an arbitrary number of small factors. 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.

After doing trial division, we detect that y is almost-smooth by checking the magnitude of the rough part after removing all factors small 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 2.3.

There is no simple analytic way to compute the probability of almost-smoothness, but we can estimate it from experiments.

Concrete Parameters. 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. 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 one processor (the process that gets an almost-smooth y) makes a memory access.

Example 6: Using smoothness to evaluate $\sqrt[a]{x}$, with a medium-size factor

$T = 13$ $M = 2^{59.5}$ $\#CPU = 2^{48}$ speedup : 20

2.4.2 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

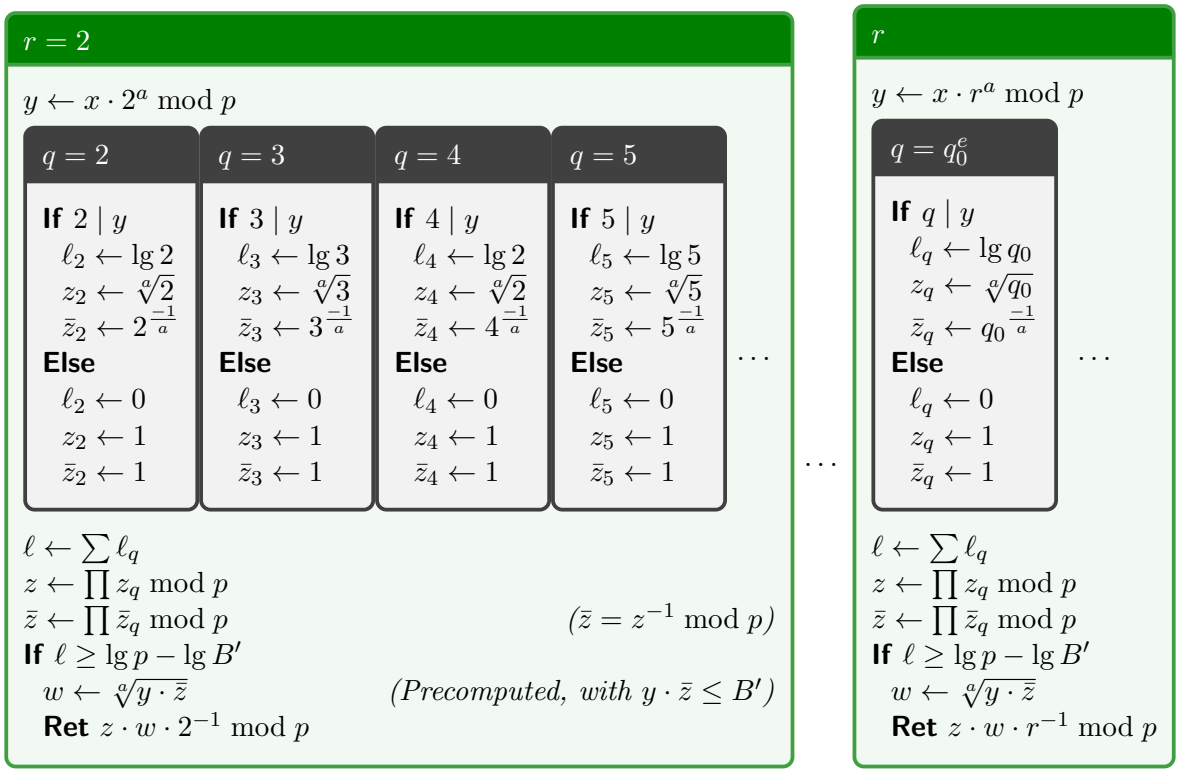

Figure 2.3: Improved algorithm using (B, B')-almost-smooth numbers

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 2.4 shows this pre-filtering algorithm.

Concrete Parameters. 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 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 $2^{-9.5}$.

Example 7: Using smoothness to evaluate $\sqrt[a]{x}$, with medium-size factor and pre-filter

$T = 14$ $\qquad$ $M = 2^{59.5}$ $\qquad$ $\#CPU = 2^{40}$ $\qquad$ speedup : 18

2.4.3 Using 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

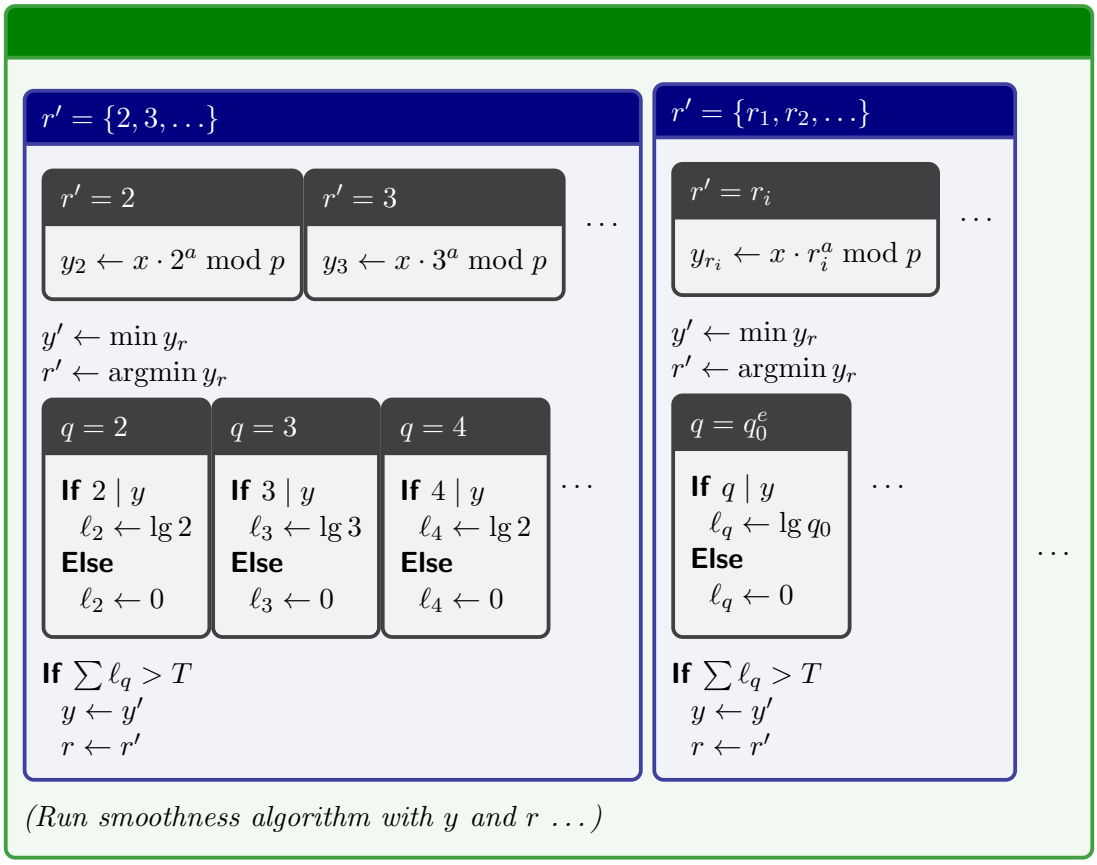

Figure 2.4: Pre-filtering step (only one CPU group shown)

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.

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 $x = x_{\text{low}} + 2^{224} x_{\text{high}}$
- For all $y < 2^{32}$, if $\text{LSB}_{32}\big(x \cdot y \bmod p\big) = 0$, set $\mathcal{T}[x_{\text{low}}, x_{\text{high}}] \leftarrow \sqrt[a]{y}$

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 $r_1 = \mathcal{T}[\text{LSB}_{32}(y_0), \text{MSB}_{32}(y_0)]$
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. 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-bits 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.

Example 8: Using smoothness to evaluate $\sqrt[a]{x}$, with medium-size factor, pre-filter, and the special shape of p

$T = 21$	$M = 2^{64}$	$\#CPU = 2^{36}$	speedup : 12

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 the 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.

2.4.4 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 [Wan81]. Indeed, running the extended Euclidean algorithm on x and p generate a series of relations $u_i \times x + v_i \times p = w_i$. Each relation define 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 the plus-minus algorithm of Brent and Kung [BR85]) 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 [SHT22] 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 89ns 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 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 [Sor94], with only $\mathcal{O}(\lg(n)/k)$ 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. 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)).

Example 9: Using smoothness to evaluate $\sqrt[a]{x}$, with medium-size factor, pre-filter, and rational reconstruction

$T = 54$ $\qquad$ $M = 2^{40}$ $\qquad$ $\#CPU = 2^{34}$ $\qquad$ speedup : 4.7

2.4.5 Parallel Smoothness Test

Another trick can be used to reduce the latency, at the expense of more communication. Instead of randomizing with $y \leftarrow x \cdot r^3 \bmod p$ 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 appoach 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 2.5 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 $\lg(n)$ other processors), n processors can route n messages in probabilistic time $\mathcal{O}(\lg(n))$ [Val82].

Concrete Parameters. 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:

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

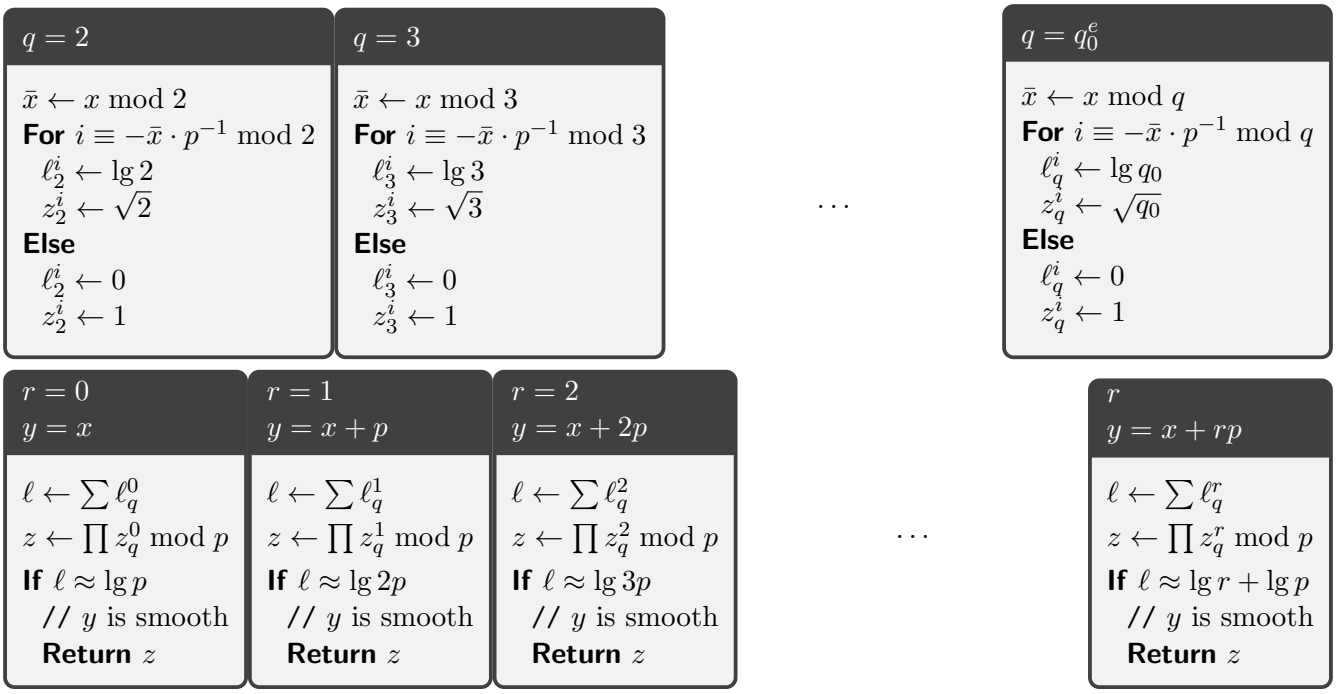

Figure 2.5: Parallel smoothness test

We 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 numer. The probability that they are $(2^{32}, 2^{45})$-almost-smooth is about 2^{-24}, so that the algorithm succeeds with high probability.

Example 10: Using smoothness to evaluate $\sqrt[a]{x}$, with medium-size factor, and parallel smoothness test

$T = 13$ $M = 2^{40}$ $\#CPU = 2^{29}$ speedup : 20

2.4.6 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, and we keep 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 slighlt 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 $\sqrt[a]{x}$ 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 2.4.5 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 2.1.2, the fastest approach is a precomputed table, with latency 6 units, but it requires a large amount of memory.

Concrete Parameters. 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 numer. 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 requires a memory of size $2^{48.8}$). Using an extended GCD algorithm or Euler's theorem, the number of iteration is $\mathcal{O}(\lg(q_i))$. Moreover, the extended GCD algorithm implemented in [SHT22] has latency 89ns, less than half the latency of the square and multiply algorithm implemented by Supranational for MinRoot. Therefore, we assume that computing inverses in $\mathbb{Z}_{q_i}$ has latency $\lg(q_i)/2 \approx 14$ units.

Example 11: Using smoothness to evaluate $\sqrt[a]{x}$, with medium-size factor, rational reconstruction, and parallel smoothness test

$T = 68$	$M = 2^{40}$	$\#CPU = 2^{25}$	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).

2.4.7 Relation with Discrete Logarithm

We observe that the algorithms above are very close to classical algorithms for the discrete logarithm problem: Section 2.3.1 is similar to the baby step-giant step algorithm [Sha71], Section 2.3.2 is similar to the Pohlig-Hellman algorithm [PH78], and Section 2.3.3 is similar to index calculus [Adl79].

Therefore, we considered whether more advanced discrete logarithm algorithms could be adapted to the context of low-latency computation of roots.

Using ECM. The elliptic curve method [Len87] (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 discrete log is in the class $L[1/2]$. There are more efficient algorithms known, whose complexity in the class $L[1/3]$, such as the Number Field Sieve [Gor93] and the Function Field Sieve [AH99]. variants.

Further work is needed to evaluate the potential of these ideas, but our impression is that these algorithms have too many sequential steps to be useful in the context of low-latency algorithms.

2.5 Application to VDF Constructions

The previous algorithms break the sequentiality of several VDF constructions, such as MinRoot [KMT22], VeeDo [Sta20] and Sloth++ [BBBF18]. 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 2.4.5, an attacker with 2^{29} processors can compute the round function about 20 times faster than a legitimate user.

2.5.1 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 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 this 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 latency of a MinRoot round is just the latency of the a-th root.

2.5.2 Application to Sloth++

Sloth++ uses square roots in $\mathbb{F}_{p^2}$. The precomputation attack from Section 2.3.1 can applied directly, but attacks from Section 2.3.3 and 2.4 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 a of $\mathbb{F}_{p^2}$ is a polynomial $a_0 + a_1 X$. The square root z of a satisfies:

$$\begin{aligned}z^2 &= a \\ (z_0 + z_1 X)^2 &= a_0 + a_1 X \\ z_0^2 - \alpha z_1^2 + 2 z_0 z_1 X &= a_0 + a_1 X \\ &\begin{cases} 2 z_0 z_1 & = a_1 \\ z_0^2 - \alpha z_1^2 & = a_0 \end{cases} \\ &\begin{cases} z_0 & = a_1 / 2 z_1 \quad \text{(assuming } z_1 \neq 0\text{)} \\ \frac{a_1^2}{4 z_1^2} - \alpha z_1^2 & = a_0 \end{cases}\end{aligned}$$

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

$$\frac{a_1^2}{4u} - \alpha u = a_0 \qquad\qquad \frac{a_1^2}{4} - \alpha u^2 = a_0 \cdot u$$

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 same low latency algorithm as the square root).

2.6 Practical Issues

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

2.6.1 Dealing with Errors

Most of the algorithms given above are probabilistic. Since we consider only the round function $\sqrt[e]{x}$, and MinRoot has in the order of 2^{40} iterations we need a very high success rate in order to successfully compute the full MinRoot.

However, it is easy to deal with rare erroneous computations, because the computations can be efficiently checked (if $y = \sqrt[e]{x}$ then $y^a = x$). One option is to repeat the algorithm when it fails; this increase the latency but if the failure rate is small, the average latency stays small. Another option is to run the standard implementation in parallel with 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.

2.6.2 Communication Cost

The analysis above essentially neglects communication cost. In practice, this is likely to be an important bottleneck, because communication between millions of CPU takes times, and requires a large communication network; this is likely to dominate the cost of the machine [Wie04]. 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 between each round, to collect the result from the core that succeeded and broadcast it to other cores (some algorithm also require even more communication). In this section, we briefly discuss how to implement those communication, 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 probability 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 output from a handful of lucky processors. There is ongoing research on integrating optical elements into existing chip design [AMP+18], and the hypothetical Twinkle factoring device by Shamir also used in optics for finding B-smooth numbers [Sha99].

2.6.3 Physical Constraints

Speed of light. We want the attack to be faster than the standard implementation. Taking the Supranational implementation 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 processors 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} multiplication 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.

2.7 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 this 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 Section 2.3) compute $\sqrt[q]{x}$ with low latency. This breaks the VDF property because the standard implementation requires about 256 squarings to compute the root. An option could be to use x^e with small e in the round function instead of $\sqrt[q]{x}$: this reduces the latency of the standard implementation and the algorithms of Section 2.3 are no longer competitive.

However, the algorithms of Section 2.2 show that a low-degree round function can also be speed 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 obtain confidence that there are no better attack, because the field has barely been 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 left multiplication by a constant.

2.8 Other Remarks

We observed that when computing the inverse function, two root computations can be parallelized. This is a well known property of MISTY networks as opposed to Feistel networks, but it doesn't seem to have an impact in the VDF setting.

If the round constant is a fixed value c instead being the round counter i, then there are fixed points in the round function that can be found by solving a simple algebraic expression: the fixed points have the form $z, z + c$ with $z^a = 2z + c$.

2.8.1 On Algebraic Attack

Algebraic attack are briefly mentioned in the MinRoot proposal. We tried to evaluate the complexity of a simple algebraic attack to speed up the computation of several MinRoot rounds.

We consider k rounds of MinRoot, and we write the input as a pair of polynomial of the output (x, y). Using the low-degree of the inverse round function, we obtain polynomials $P(x, y), Q(x, y)$, with degree at most a^k. Actually, due to the MISTY structure of the round function, the degree is only $a^{k/2}$. Then, evaluating k rounds on input (u, v) is equivalent to solving a polynomial system $P(x, y) = u, Q(x, y) = v$.

In order to solve such a system, the most efficient way is to compute the resultant of $P(x, y) - u$ and $Q(x, y) - v$. We obtain a univariate polynomial $R(x)$ of degree a^k, and we can find the root of this polynomial with sequential complexity quasi-linear in the degree a^k by computing the polynomial GCD between R and $x^p - x \bmod R$. For a low-latency variant, we can precompute the resultant R as a polynomial whose coefficients are polynomials in u and v (and potentially the round constant). Then evaluating k rounds of MinRoot boils down to:

- Substitute parameters u and v in the coefficients of R
- Compute $x^p - x \bmod R$ (using square and multiply for $x^p \bmod R$)
- Compute the polynomial GCD between R and $x^p - x \bmod R$

The first step is done in parallel and has low latency. The second step requires further investigation, but it is typically similar to the last step in complexity.

For the last step, there exist an algorithm to compute the GCD of two univariate polynomials of degree d in parallel time $\mathcal{O}(\lg^3(d))$ using $d^2/\lg^2(d)$ processors [Pan96]. This implies that the latency of the shortcut attack to evaluate r rounds is in $\mathcal{O}(k)$. However, as mentioned in MinRoot proposal, it is unlikely that the GCD can be computed fast enough to obtain a algorithm faster than the standard implementation: for instance with $k = 20$ and $a = 3$ this would require to compute the GCD of 2 degree-2^{32} polynomials with latency smaller than 5120 field operations.

Chapter 3

Attack 2 Group

Written by Vincent Rijmen

3.1 Introduction

This report summarises the findings of cryptanalysis group 2. It is based on the notes sent to me by Christian Rechberger, my own notes and the scribblings on the flip-over.

The main result of our brainstorm was an attack based on masking, as used in side-channel attack countermeasures. It is explained in Section 3.2. Section 3.3 contains our notes on hardware costs related to this attack. They are based on discussion with the representative of Supranational. Section 3.4 contains some other observations.

Before we start, it is worth while to explain our (informal) security model as explained by Dmitry. We consider a break if an adversary exists that can compute the MinRoot function significantly faster than an ordinary user can. The adversary is allowed to perform 2^{128} precomputations, to use many parallel processors and to use large tables.

An important observation is that an efficient algorithm to compute the MinRoot function without using precomputations, parallelism or tables is not a break, because then the ordinary user can also use this algorithm.

3.2 Masking-based attacks

3.2.1 Generalities on masking

By *masking* we mean the method where the computation of a function $f(x) : \mathbb{F} \to \mathbb{F}$, is implemented using t functions $f_j : \mathbb{F}^s \to \mathbb{F}^s, j = 1, \ldots t$, with as property that

$$\forall (x_1, x_2, \ldots, x_s) \in \mathbb{F}^s : f\left(\sum_{i=1}^{s} x_i\right) = \sum_{j=1}^{t} f_j(x_1, x_2, \ldots, x_s) \tag{3.1}$$

Furthermore, (for resistance against side-channel attacks) we require that for each j there is at least one i such that f_j does not depend on x_i. The variables x_i are called the *shares*. Masking has been used as a countermeasure against side-channel attacks but here we exploit that if f_j does not depend on x_i, then under certain conditions (discussed below) we can implement f_j using a table that is smaller than a table needed to implement f directly. Since each f_j can be computed in parallel, the time for the t lookups is not larger than the time for one lookup. In order to obtain the total execution time of (3.1) we need to add the time to execute the sum, which equals $\lceil \log_2 t \rceil$ additions.

3.2.2 Masking for faster computation

We now consider a small modification to the masking method, in order to make it suitable for accelerated computation of f. In particular, we restrict the shares x_i to sets $S_i \subset \mathbb{F}$, where each S_i is significantly smaller than $\mathbb{F}$. The sets S_i need to satisfy the following requirement:

$$\forall x \in \mathbb{F}, \exists x_1 \in S_1, \ldots x_s \in S_s : x = x_1 + \cdots + x_s$$

Furthermore, we need an efficient projection function

$$p : \mathbb{F} \to S_1 \times \cdots S_s : x \mapsto p(x) = (x_1, \ldots, x_s)$$

where $x_1 + \cdots + x_s = x$.

The adversary will precompute tables T_j defined by

$$\forall x \in \mathbb{F} \text{ with } p(x)_j = 0 : T_j[x] = f_j(p(x)).$$

It follows that the size of the largest table T_j is determined by the cardinality of the smallest set S_j.

We propose to define the sets S_j as follows. Denote $n = \lceil \log_2 |\mathbb{F}| \rceil$. Let $m = \lceil n/s \rceil$ and take

$$S_j = \{x \in \mathbb{F} | (j-1)2^m \leq x < j2^m\}.$$

This definition allows for an efficient projection function p, that simply maps m bits of x to each coordinate x_i (after scaling).

Finally, in order to compute $f(x)$, the adversary first computes $(x_1, \ldots, x_s) = p(x)$, then uses the tables T_j to look up the values $f_j(p(x)), j = 1, \ldots, t$ and sums the t values.

3.2.3 Application

Let d denote the algebraic degree of the function f. In [PAB+22] a general construction method is given to determine the functions $f_1, \ldots f_t$ where $t = d + 2$. The method can be applied recursively on each of the f_j in order to obtain arbitrarily small tables. The depth of the summation will of course increase accordingly.

For functions with a low algebraic degree, the number of x_i that f_j do not depend on can be larger than one. This observation can be used to further reduce the size of the tables. Let f be the monomial

$$f(x) = x^d = (x_1 + x_2 + \cdots + x_s)^d$$

Working out this exponentiation results in $\binom{s}{d}$ monomials each containing d variables. Then we can take each monomial to be a function f_j. We obtain $t = \binom{s}{d}$ tables (functions). The size of (the number of entries in) each table is $2^{dn/s}$. A very similar approach can be applied if f consists of several monomials. The monomial of the largest degree determines t and the size of the tables. Table 3.1 gives some numerical values.

3.2.4 Extensions

So far we have been using arithmetic masking. We can also use Boolean masking: $x^d \to (x_1 \oplus x_2 \oplus \cdots \oplus x_s)^d$ or multiplicative masking $x^d \to (x_1 \times x_2 \times \cdots \times x_s)^d$. This replaces the summations in (3.1) by XORs or multiplications.

The multiplication case comes close to what has been done in Attack group 1, but it is not clear whether it is 100% the same. The unsolved issue is how to define the sets S_j for this case. If $p-1$ factors into small primes then we could use an algorithm based on Pohlig-Hellman to compute tables of roots modulo factors of $p-1$. Both approaches require further research to reach a conclusion.

degree of f d	no. of shares s	no. of tables $t = \binom{s}{d}$	no. of entries in one table $2^{256d/s}$
2	16	120	2^{32}
2	32	496	2^{16}
4	64	$\approx 2^{20}$	2^{16}
8	16	$\approx 2^{14}$	2^{128}
8	32	$\approx 2^{23}$	2^{64}
8	64	$\approx 2^{32}$	2^{32}

Table 3.1: Some numerical values for table size and number of tables.

3.3 Costs

Large tables would have to be stored as DRAM on one or more dedicated chips.

Using 5nm technology we are looking at a latency of 5ns for a single lookup. This seems to be slow, but the fact that we require really random access implies that burst mode cannot be used to drive down the average delay. In this technology we can expect a cost of 75M€ per lithography-on-wafer "run." One chip could store one table of 2^{25} 256-bit entries. Using the example $d = 2$ and $s = 16$ from Table 3.1, we see that we would need about 100 chips to hold one table; we need 120 of these tables. In summary, our total cost would be almost 10^{12}€. The additional latency for the computation of the sum of the 120 entries was estimated at 1ns.

An alternative approach would be to use 12nm technology. This technology would be available for 2M€ per run, The latency would be twice as high.

Finally, one could also consider the use of much smaller tables such that they fit on the same die.

As conclusion we can state that for hardware implementations it is difficult to predict which approach will give the best result, because there are many factors in play and it is not easy to develop an intuition for their relations.

3.4 Additional observations

We use the following notation for the MinRoot round function:

$$F_i(x, y) = (\sqrt[5]{x + y}, x + i) \pmod{p} \tag{3.2}$$

The following property of $F_i(x, y)$ can easily be verified:

$$F_i(x, y) = F_{i+t}(x - t, x + t)$$

We see no way to extend this property over multiple rounds.

We denote by $G(x, y)$ a simplified version of MinRoot: without addition of the counter. Hence we get:

$$G(x, y) = (\sqrt[5]{x + y}, x) \pmod{p} \tag{3.3}$$

Then, for any s satisfying $s^4 = 1 \pmod{p}$ (i.e. s is a fixpoint of the root, for example $s = p - 1$), we have

$$G(sx, sy) = (\sqrt[5]{s}\sqrt[5]{x + y}, sx) = sG(x, y) \pmod{p}$$

We denote by $H_i(x, y)$ the modified MinRoot function defined as follows

$$H_i(x, y, z) = (\sqrt[5]{x + y}, x/z + i) \pmod{p}$$

This modified function has the following property:

$$H_{i\,\sqrt[5]{r}}(rx, ry, r^{4/5}) = \sqrt[5]{r} H_i(x, y, 1)$$

3.4.1 Differentials

We consider two-round iterative differentials $(\alpha, \beta) \to (\alpha, \beta)$ with a difference γ in the round counter. The right pairs for such a differential satisfy the following equation:

$$F_{i+1+\gamma}\left(F_{i+\gamma}(x+\alpha, y+\beta)\right) = F_{i+1}\left(F_i(x,y)\right) + (\alpha, \beta)$$

This requires:

$$\sqrt[5]{z+\alpha+\beta} = \sqrt[5]{z} + \beta - \gamma$$
$$\sqrt[5]{u+\beta} = \sqrt[5]{u} + \alpha$$

(Here $z = x + y$ and $u = x + i + \sqrt[5]{z}$.) In order to be useful in differential cryptanalysis, we would need values for α, β, γ for which these equations have many solutions. It seems unlikely that such values exist.

3.4.2 Nonlinear invariants

We also looked for nonlinear invariants of G. A nonlinear invariant would be a function f such that if an input (x, y) satisfies $y = f(x)$ then the corresponding output $(X, Y) = G(x, y)$ would satisfy $Y = f(X)$, possibly only with some probability. Using (3.3) we get the following equation in f:

$$x = f\left(\sqrt[5]{x + f(x)}\right)$$

which is equivalent to

$$f^{-1}(x) = \sqrt[5]{x + f(x)}$$

We did not find a solution for f. One idea that could be explored further is to impose a certain structure on f, e.g. $f(x) = ax^b$ and try solving for a and b.

Chapter 4

Theory 1 Group

Written by Krzysztof Pietrzak

4.1 Introduction

This document summarises the initial results and ideas discussed by the "theory group" at the Ethereum VDF event in Lyon April 28-30, 2023.

Contributors. The people present during the discussions were
S. Tessaro University of Washington, USA
Y. Dodis New York University, USA
G. Segev Hebrew University of Jerusalem, Israel
I. Komargodski Hebrew University of Jerusalem, Israel
K. Pietrzak ISTA, Austria
B Bunz Espresso Systems, USA
B Fisch Yale, USA
M. Simkin Ethereum Foundation, Denmark

4.1.1 VDF security

A VDF is a function $(y, \sigma) \leftarrow \mathsf{VDF.Eval}(pp, x, t)$ which can be evaluated (with little parallelism) on any input x and public parameters pp in t sequential steps, and the proof σ (certifying that y is the correct output) can be efficiently verified. A VDF must satisfy two security properties, *sequentiality* and *soundness*. Informally, sequentially means that computing the output y on a random (or at least unpredictable) input x and some time parameter t takes t sequential steps (and thus linear in t time)

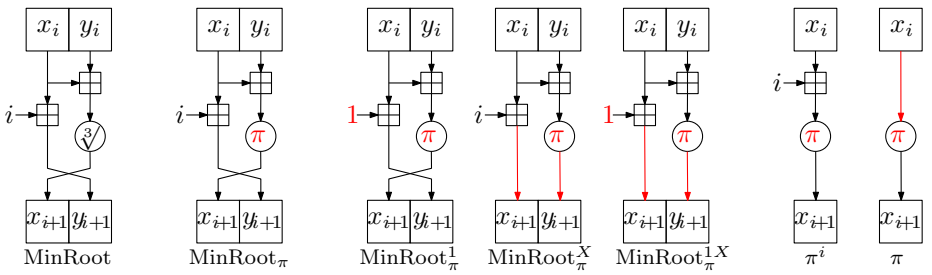

Figure 4.1: A single round of the MinRoot VDF and $\mathsf{MinRoot}_\pi$, where the cube root is replaced with a random permutation π. The remaining figures show simplified versions whose analysis can give indications on the role and efficacy of the design choices made for MinRoot.

even with massive parallelism, while the soundness requirement states that it should be computationally infeasible to come with a proof for a wrong statement.

MinRoot is a verifiable delay function (VDF) proposed for the Ethereum ecosystem in [KMT22], during the meeting we only discussed the sequentiality property of the MinRoot VDF, and for the rest of the document we'll ignore the verifiability part.[1]

4.1.2 (hashed) MinRoot

The MinRoot VDF evaluation function

$$(x_t, x'_t) \leftarrow \mathsf{MinRoot.Eval}((x_0, x'_0), t)$$

computes the output by iterating a round function $\mathsf{minroot} : \mathbb{F} \times \mathbb{F} \to \mathbb{F} \times \mathbb{F}$

$$\mathsf{minroot}(x_i, x'_i) \to (\ (x_i + x'_i)^{\frac{2p-1}{3}}\ ,\ x_i + 1\ ) \ = \ (x_{i+1}, x'_{i+1})$$

shown in Figure 4.1. Here $\mathbb{F}$ is a field of size $|\mathbb{F}| = p$ where the cubic root $\sqrt[3]{x} = x^{\frac{2p-1}{3}}$ is defined. The authors of [KMT22] claim 128 bit security for sequentiality when $\mathbb{F}$ is a $\log(p) \approx 256$ bit field. The most demanding computation in the round function is taking the cubic root, the rationale for choosing this particular function is the fact that inverting this function (i.e., $x \to x^3$) is much faster than going in forward direction. This is useful when computing the proof σ.

Apart from the basic $\mathsf{MinRoot}{:}\mathbb{F}\times\mathbb{F} \to \mathbb{F}\times\mathbb{F}$ evaluation we will also consider $\mathsf{hMinRoot}{:}\{0,1\}^* \to \mathbb{F}\times\mathbb{F}$

$$\mathsf{hMinRoot}(m, t) = \mathsf{MinRoot}(\mathcal{H}(m), t)$$

which simply first hashes the input using a hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{F} \times \mathbb{F}$. This not only allows for an arbitrary input domain, but will also allow us to argue security for a new and stronger "knowledge" type security notion.

4.1.3 Idealized Models

Proving sequentiality of MinRoot or any other construction unconditionally is too ambitious as it would require breakthroughs in computational complexity. Instead, we can try to prove the sequentiality in idealized models. There are at least two natural idealizations for MinRoot.

generic field: Instead of considering a concrete field $\mathbb{F}$, analyse the security in some idealized algebraic model capturing only generic algorithms over fields.

random permutation: Instead of considering the round function $\sqrt[3]{\ }$, model the round function as a random permutation π over $\mathbb{F}$. To model the fact that $\sqrt[3]{\ }$ can be inverted faster than computed in forward direction, we will assume that computing π takes 1 unit of time, while inverting it takes some $\gamma < 1$ time (for MinRoot this γ is around $1/128$ as taking cube roots).

random oracle: For hMinRoot we can model the hash function $\mathcal{H}$ as a random oracle.

A security proof in a generic group model would imply that any attacker breaking sequentiality (beyond the proven bound) must exploit the underlying field in a non-generic way. Such a proof would thus be very meaningful, but proving security in such a model seems very ambitious (see the discussion in [RSS20]). In this document we will consider the random permutation model, and from now on MinRoot denotes the construction in the random permutation model, for hMinRoot we additionally model $\mathcal{H}$ as a random oracle.

[1] For the proof, MinRoot will adapt an approach originally proposed in [BBBF18] and use incrementally verifiable computation to compute such a proof. Here the computation of the proof requires significantly more effort than computing the output y, but the computation can be parallelized, so the in practice one can create the proof shortly after y has been computed.

4.1.4 A Knowledge Type Definition for Sequentiality

The original definition of sequentiality [BBBF18] requires that given some random challenge x and some time parameter t one almost certainly needs almost t sequential steps to compute the output $y = \mathsf{VDF.Eval}(x, t)$.

One could also consider a knowledge type assumption where we assume that whenever the adversary outputs (x, t, y) where $y = \mathsf{VDF.Eval}(x, t)$ at some timepoint, it must have "known" x some time t ago (so it could just have run the honest evaluation $y \leftarrow \mathsf{VDF.Eval}(x, t)$).

It's easy to see that MinRoot is *not* secure in sense as inverting the round function can be done in $\gamma < 1$ time. Concretely, one can just pick any output (x_t, x'_t) and then compute the corresponding input $(x_0, x'_0) \leftarrow \mathsf{MinRoot}^{-1}(x_t, x'_t)$ in time $\gamma \cdot t < t$. Note that this is not possible for hMinRoot as to find an input one would have to additionally invert the random oracle, i.e., find some m s.t. $\mathcal{H}(m) = (x_0, x'_0)$, and we will prove that hMinRoot is indeed secure in this stronger sense.

To capture what "knowing" a preimage in this context means we'll define an extractor, which as inputs gets the oracle queries made by an adversary so far, and outputs a (ideally small) list $\mathbb{F}$ of inputs, such that for all inputs *outside* of this list it's extremely unlikely that the adversary would succeed in computing the output much faster than by using the regular evaluation algorithm.

It's not clear if or where this notion could be useful. It's stronger than the classical notion, for example it immediately implies variants of the standard notion where the challenge is not necessarily random, but just needs to have some type of entropy (i.e., we just need the probability of the challenge falling into the list $\mathbb{F}$ to be very small). One can also think of settings where this stronger notion is necessary, but they are rather artificial and we currently don't have a natural application where the stronger notion would be really necessary. Finally, such a knowledge type notion for sequentiality might be useful when arguing about composition (like in UC).

4.1.5 Stripped Down MinRoot

The main computation in a MinRoot round as illustrated in Figure 4.1 is computing a cube root (or random permutation π in the idealized setting). Just iteratively taking cube roots would not be a sequential function (as the group order is known, iterating this any t times just boils down to taking a single exponentiation), so this operation is embedded in a mode of operation, which is basically the MISTY mode of operation, with the extension of adding the round number in every round. To understand which parts of the construction contribute to the security it could be instructive to not just determine the exact security of (an idealized version of) MinRoot but also "stripped down" versions which are derived by, e.g. by adding a constant instead of the round, not adding anything at all, omitting the swap of the right and left half, etc., as illustrated in Figure 4.1.

4.2 Sequentiality

We'll discuss a few ways in which sequentiality can be defined. The notions we consider will model the adversary as a tuple $\mathcal{A}_0, \mathcal{A}_1$ where $\mathcal{A}_0$ models a potential precomputation and $\mathcal{A}_1$ the actual attack. More precisely, $\mathcal{A}_0$ gets as input the public parameters pp and outputs some state st, we then sample some challenge input(s) which together with st are given to $\mathcal{A}_1$, wo then tries to compute the evaluation on the challenge input faster than the honest evaluation algorithm. As the hardness will come from the ideal permutation π (and for hMinRoot $\mathcal{H}$), we can assume $\mathcal{A}_0$ and $\mathcal{A}_1$ are computationally unbounded, and just bound the way they can access their oracles $\pi, \pi^{-1}, \mathcal{H}$. A (potentially parallel) query to $\pi, \pi^{-1}, \mathcal{H}$ will take time $1, \gamma$ and 0, respectively.

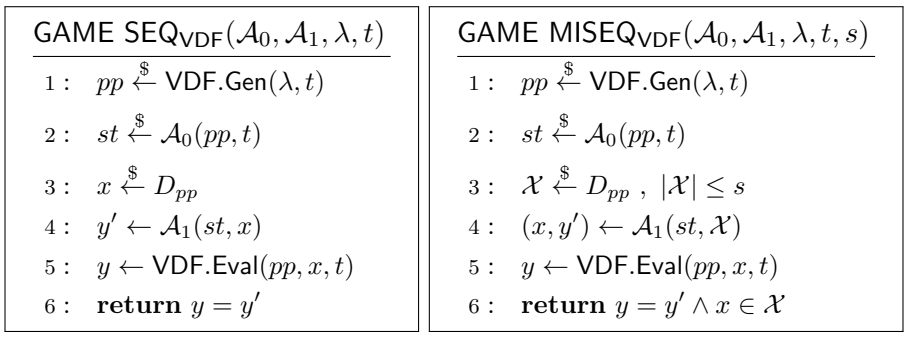

Figure 4.2: The sequentiality and the multiple-inputs sequentiality game

4.2.1 bounded query vs. space and why it probably doesn't matter

We need to bound $\mathcal{A}_0$ so it can't simply output the entire function table of π as the state st to be passed to $\mathcal{A}_1$. Two natural ways to bound $\mathcal{A}_0$ are to (1) bound the number of oracle queries $\mathcal{A}_0$ can make to some value q and the state st simply contains all queries $\mathcal{A}_0$ learned or (2) bound the size of the state st that can be passed to the 2nd stage, here $\mathcal{A}_0$ can learn the entire function table of $\pi, \mathcal{H}$, but may only pass a compressed version of it to $\mathcal{A}_1$.

Generally, for a fixed size of the state st, passing all queries as in (1) results in a much weaker adversarial model than allowing arbitrary precomputation as in (2). For example inverting a random permutation over a domain of size p on a random value can be done with $\approx p/|st|\log(p)$ queries in (2) but requires $\approx p - |st|\log(p)$ in (1).

For the sequentiality of MinRoot in the random permutation model this distinction doesn't seem to matter: the time-memory trade-off for inverting a random permutation mentioned above can't be used to break sequentiality as for breaking sequentiality an inversion must happen in parallel (i.e., non-adaptively), and here no speedups are known [CHM20].

Ideally we would like to have a lower bound in the bounded space setting (2), and a matching upper bound in the bounded query setting of (1).

In this note we give an upper bound (Theorem 2) in the bounded query setting which we believe is close to the real security. We also give a lower bound in (Theorem 1), but this bound is clearly far from the real one and also in the bounded query model. We leave proving a tight lower bound in the bounded space setting as an open challenge.

4.2.2 challenge space

The simplest way to define the challenge for $\mathcal{A}_1$, i.e., the (x_0, y_0) on which it must output $\mathsf{MinRoot.Eval}((x_0, y_0, t))$ in $< t$ time, is to sample it uniformly at random, this is captured in the $\mathsf{SEQ}_{\mathsf{VDF}}(\mathcal{A}_0, \mathcal{A}_1, \lambda, t)$ security game. We can also define a multi-challenge game $\mathsf{MISEQ}_{\mathsf{VDF}}(\mathcal{A}_0, \mathcal{A}_1, \lambda, t, m)$ where $\mathcal{A}_1$ can pick one out of m possible challenges to attack. While security in the single challenge game implies security in the multi challenge game, this reduction loses a factor m in the advantage, while a direct analysis of the multi challenge game might give better bounds. In this note we prove (in Thm. 1)) a bound in the single challenge game and then state the bound this implies in the multi-challenge game via the trivial reduction.

4.2.3 A simple birthday type lower bound

In this section we will give a bound on the advantage of any adversary $(\mathcal{A}_0, \mathcal{A}_1)$ winning the sequentiality game $\mathsf{SEQ}_{\mathsf{VDF}}(\mathcal{A}_0, \mathcal{A}_1, \lambda, t)$ game from Figure 4.2 (winning means the output of the game is 1) if the VDF is MinRoot instantiated with a random permutation π.

We define a (T_0, T_1, t') adversary as follows. $\mathcal{A}_0$ and $\mathcal{A}_1$ get oracle access to π and π^{-1}. They have a computation "budget" of T_0 and T_1 respectively, where each π query cost 1 and each inverse query to π costs $\gamma < 1$. As discussed before, we'll just consider the setting where the state st passed from $\mathcal{A}_0$ to $\mathcal{A}_1$ contains all the queries made by $\mathcal{A}_0$. For $\mathcal{A}_1$ not just the computation, but also its sequential time matters, this is captured by the t' parameter, which for our theorem can be any value less than t, say $t' = t - 0.0001$. The clock starts when $\mathcal{A}_1$ is invoked on the challenge input (x_0, y_0) in the game, and whenever $\mathcal{A}_1$ makes a query x to π at time τ, it only receives the output $\pi(x)$ at time $\tau + 1$, similarly $\mathcal{A}_1$ gets the output $\pi^{-1}(y)$ to on an inverse query y made at time τ at time time $\tau + \gamma$.

The bound in the theorem is a simple "birthday bound" type result: the proof exploits the fact that assuming the adversary never makes a query that collides with any of the t invocations to π required to evaluate MinRoot before that query is actually known, they will not be able to learn the output.

Theorem 1 (Birthday bound for sequentiality)**.** *Let $q = (T_0 + T_1)/\gamma$, then the probability that any $(T_0, T_1, t' < t)$ adversary $(\mathcal{A}_0, \mathcal{A}_1)$ (as defined above) wins the $\mathsf{SEQ}_{\mathsf{VDF}}(\mathcal{A}_0, \mathcal{A}_1, \lambda, t)$ game if the VDF is instantiated with MinRoot using a random permutation π over some field $\mathbb{F}$ is at most*

$$\frac{q \cdot t + 1}{|\mathbb{F}| - q} \approx \frac{t \cdot (T_0 + T_1)}{\gamma \cdot |\mathbb{F}|}$$

This implies a $s \cdot \frac{q \cdot t + 1}{|\mathbb{F}| - q}$ bound for the multi-challenge game $\mathsf{MISEQ}_{\mathsf{VDF}}(\mathcal{A}_0, \mathcal{A}_1, \lambda, t, s)$.

Proof. Let $(x_0, y_0) \leftarrow D_{pp}$ be the challenge and let $(x_0, y_0), \ldots, (x_t, y_t)$ denote the states during an evaluation of $\mathsf{MinRoot}((x_0, y_0), t)$. Let $i_0 \ldots i_t$ with $i_j = x_j + y_j$ denote the inputs to π during this evaluation.

$i_0 = x_0 + y_0$ is uniformly random, and also i_j is (very close to) uniformly random unless $\pi(i_{j-1})$ has already been evaluated. The only way $\mathcal{A}_1$ can learn the correct output (x_t, y_t) is by learning $\pi(i_j)$ for all $j = 0, \ldots, t-1$, but there's not enough time to make those queries sequentially. So if $\mathcal{A}_1$ does learn those queries, some tuple $(i_j, \pi(i_j)), 0 \le j < t$ was already known before the query $\pi(i_{j-1})$ was made (by either $\mathcal{A}_0$ or $\mathcal{A}_1$).

As at any timepoint in the game π is known on at most $q = \frac{T_0 + T_1}{\gamma}$ points, the output of π on a (yet not known) input i_j will be uniform over a set of size at least $|\mathbb{F}| - q$, and thus hit any of the (at most q) already known values with probability at most $q/(|\mathbb{F}| - q)$. Taking the union bound over all $i_j, 0 \le j < t$ we get

$$\Pr[\mathcal{A}_1 \text{ learns all } \pi(i_j), 0 \le j \le t \text{ in time } t' < t] \le \frac{q \cdot t}{|\mathbb{F}| - q}$$

If $\mathcal{A}_1$ did not learn all $\pi(i_j), 0 \le j \le t$, the best it can do is guess the output (x_t, y_t), which again will succeed with probability at most $1/(|\mathbb{F}| - q)$. The bound in the theorem is the sum of the two probabilities above. □ □

4.2.4 An simple upper bound

It's not clear how tight the bound in Theorem 1 is, but if we not just ask for $t' < t$ but make the gap more substantial, say $t' = t - 10$, the bound is obviously far from tight (in practice t is in the ballpark of 2^{30}, so a gap of 10 between t and t' doesn't matter).

The reason (why the bound is not tight) is that even if the adversary is lucky and causes a collision between some already computed π value and the values on which π is queried during the evaluation on the challenge input, this just means they can speed up the evaluation of MinRoot from t to $t - 1$ (or to $t - k$ if they find k collisions).

To break sequentiality by a significant amount it seems necessary to not just guess some inputs to π that pop up during evaluation of the challenge, but to the entire MinRoot round function (this is equivalent to guessing the input to π for two *consecutive* rounds). As the MinRoot round function is over a much larger domain than π ($|\mathbb{F}|^2$ vs. $|\mathbb{F}|$), the upper bound we get is much worse than the lower bound from Theorem 1.

Theorem 2 (Simple upper bound)**.** *In the setting of Theorem 1, for any t, d, T_0 and $T_1 = t - d$, a $(T_0, T_1, t - d)$ adversary can win the $\mathsf{SEQ}_{\mathsf{VDF}}(\mathcal{A}_0, \mathcal{A}_1, \lambda, t)$ game with advantage*

$$\frac{T_0}{\gamma \cdot d \cdot |\mathbb{F}|^2} \tag{4.1}$$

Proof. We describe the adversary $(\mathcal{A}_0, \mathcal{A}_1)$ that achieves the advantage as stated. $\mathcal{A}_0$ simply computes $\mathsf{MinRoot}(z_i, d)$ for as many distinct values $z_1, \ldots, z_s$ as the T_0 budget allows. If the challenge to $\mathcal{A}_1$ falls into this set, $\mathcal{A}_1$ can compute the output and win the game computing just the $t - d$ remaining rounds (in time $t - d$ making $t - d$ sequential queries to π).

To sample an input/output tuple for $z, \mathsf{MinRoot}(z, d)$ requires d queries to π or to π^{-1} (by sampling the output and then inverting). The latter is cheaper and costs $d \cdot \gamma$, this allows $\mathcal{A}_0$ to compute a total of $T_0/d \cdot \gamma$ such tuples. The probability that the challenge input will hit this set is $\frac{T_0}{\gamma \cdot d \cdot |\mathbb{F}|^2}$. □ □

It seems reasonable to conjecture that (for practically relevant parameters, in particular $T_0 \ll |\mathbb{F}|$ and d a sufficiently large constant) the simple upper bound from Thm. 2 is close to the real security of $\mathsf{MinRoot}$ in the random permutation model. Let us mention a few observations on the bound in eq.(4.1).

The denominator in eq.(4.1) contains d, this d would be lost if instead of adding the round i in every $\mathsf{MinRoot}$ round one would add some fixed value (as illustrated in $\mathsf{MinRoot}^1_\pi$ in Figure 4.1), so (assuming the bound is indeed tight) this justifies the addition of the round function (or more generally, a value that is different for every round).

The denominator also contains $|\mathbb{F}|^2$, which is thanks to the fact that the domain of the $\mathsf{MinRoot}$ round function is $|\mathbb{F}| \times |\mathbb{F}|$ not just $\mathbb{F}$. The upper bound in eq.(4.1) can only be close to the real security while $T_0 \ll |\mathbb{F}|$,[2] which holds for the suggested 256 bit field. But this also means we cannot simply choose a much smaller field and compensate for that by making the state of the round larger (say a 64 bit field with a state of 8 field elements which would give the same 512 bit state as in $\mathsf{MinRoot}$).

As a last observation let us note that the online complexity $T_1 = t - d$ of the attack is very small and we don't see how to make the attack better if T_1 was much larger. This indicates that, unlike e.g. for the time-memory trade-offs for inverting permutations we discussed before, the adversarial power during the online phase is basically irrelevant. Let us stress that we only claim this for the case where the round function is a random permutation, the situation might be very different if the operation is taking a cube root in some (idealized) field.

4.2.5 A loose knowledge bound

In this section we will informally prove a simple "knowledge" type lower bound on sequentiality where an adversary who outputs a valid tuple (x, y, t) with $y = \mathsf{hMinRoot}(m, t)$ at time τ must "know" m at time $\tau - t$. As mentioned in §4.1.4, we consider $\mathsf{hMinRoot}$ as $\mathsf{MinRoot}$ in trivially not secure in this sense.

Consider the setting of Theorem 1, but against $\mathsf{hMinRoot}$, and where we consider a $(T_0, H_0, T_1, H_1, \gamma, t')$ adversary, where the additional H_0, H_1 parameters denote the number of oracle queries we allow $\mathcal{A}_0$ and $\mathcal{A}_1$ to the $\mathcal{H}$ oracle. We assume that a query to $\mathcal{H}$ takes no time at all. As in Theorem 1, $t' < t$ can be arbitrary close to t.

We consider a slightly different security game, where an extractor $\mathsf{Extract}$ gets as input all the oracle queries made by $\mathcal{A}_0$, and then outputs a list $\mathbb{F}$ of "forbidden" inputs. In the second phase $\mathcal{A}_1$ only gets $\mathbb{F}$ (and t), and wins the game if it outputs a tuple

$$(m, y = \mathsf{hMinRoot}(m, t)) \text{ for } \textit{any } m \notin \mathbb{F}$$

The extractor we'll use is particularly simple, it just outputs a list $\mathbb{F}$ that contains all queries made by $\mathcal{A}_0$ to $\mathcal{H}$, so $|\mathbb{F}| \leq H_0$.

[2] In particular, once $T_0 = |\mathbb{F}|$ we can just learn the entire function table of π and (in the game where we put no bound on the state st) sequentiality is completely broken.

Theorem 3 (Knowledge type security of hMinRoot)**.** *For any T_0, H_0, T_1, H_1 and t, any $(T_0, H_0, T_1, H_1, \gamma, t' < t)$ adversary (as defined above) can win the knowledge game (as outlined above) with probability at most*

$$H_1 \cdot \frac{q \cdot t + 1}{|\mathbb{F}| - q} \qquad \textit{where} \quad q = (T_0 + T_1)/\gamma$$

(note that this is H_0 times the bound from Theorem 1).

Proof. The above bound is, and directly follows from, the multi-challenge security bound as stated in Theorem 1 when $s = H_1$. To see this note that if $\mathcal{A}_1$ outputs $(m, y), m \notin \mathbb{F}$ where it did not make the query $\mathcal{H}(m)$, the output y is correct with probability only $1/|\mathbb{F}|^2$ as there's exactly one x s.t. y is correct if and only if $\mathcal{H}(m) = x$ ($\mathcal{A}_0$ did not query $\mathcal{H}(m)$ as otherwise m would be in the forbidden list). So we assume $\mathcal{A}_1$ did make the query $\mathcal{H}(m)$, but in this case we're basically at the multi-challenge game where $\mathcal{A}_1$ gets a set of $s = H_1$ uniformly random challenges, namely the outputs of $\mathcal{H}$ on at most H_1 queries which it did not already make (as those are in the forbidden set $\mathbb{F}$). □ □

Chapter 5

Theory 2 Group

Written by Bart Mennink

5.1 Introduction

This part attempts to develop a concrete, i.e., non-asymptotic, version of the definition of Boneh et al. [BBBF18].

5.2 Concrete security definitions

Definition 1. *Let $f : X \mapsto Y$ be a function. The function is $(t, p, T, A, B, \varepsilon)$-sequential if the following conditions hold:*

1. *There exists an algorithm that can evaluate $f(x)$ for any x in parallel time t on p processors.*
2. *For any algorithm $\mathcal{A}$ operating in T preprocessing time, t/A parallel time, and on pB processors, the probability that it can compute $f(x)$ for uniform random $x \xleftarrow{\$} X$ is at most ε. Stated differently:*

$$\mathbf{Pr}_{x \xleftarrow{\$} X}\left(\mathcal{A}(x) \mapsto y \ : \ f(x) = y\right) \leq \varepsilon\,.$$

Here, we throughout assume $T \geq 0$, $A, B \geq 1$.

The definition differs from [BBBF18, Definition 6] in that it now explicitly includes preprocessing and that the definition is parameterized with concrete values (t, p, T, A, B) instead of using asymptotics. This explicit parameterization is convenient to make claims about schemes that are unprovable. The preprocessing can be made explicit using a two-phase adversary (akin to [KMT22, Definition 4]) but it does not seem to add much.

Reductions. The definition is not necessarily made to prove results with. However, some type of reductions is possible. Suppose we take the MinRoot function [KMT22] with a random permutation. Basically, a legitimate evaluation can evaluate the random permutation in t time (whatever t is) using 1 processor. However, even if you have an arbitrary amount of $B \geq 1$ processors, you cannot compute the permutation in $< t$ time. The success probability of guessing y is $1/|prime|$. The precomputation does not matter in this setting. Thus, in the random permutation model, the construction would be $(t, 1, T, A, B, 1/|prime|)$-sequential for any T, A, B.

In general, if a function f is $(t, p, T, A, B, \varepsilon)$-sequential, then it is also $(t, p, T', A', B', \varepsilon)$-sequential for $T' \leq T$ (as this means less precomputation), $A' \geq A$ (as this means less time) and $B' \leq B$ (as this means fewer processors). We can also make an observation about composition. Suppose that

f is $(t, p, T, A, B, \varepsilon)$-sequential and g is $(t', p', T', A', B', \varepsilon')$-sequential *with* $p = p'$, then $g \circ f$ is $(t + t', p, \min\{T, T'\}, \max\{A, A'\}, \min\{B, B'\}, \varepsilon \cdot \varepsilon')$-sequential. The reduction becomes a bit clumsier if $p \neq p'$ as it may influence the time $t + t'$.

The cascade of an identical function f may be hard to investigate, the main reason being that weaknesses may span multiple rounds.

Comparison with attacks. Typically, we take t and p the time and processors for a legitimate system to run f, i.e., f is ran in 1 unit of time and 1 processor. So the first condition is satisfied. Looking at the attacks of the cryptanalysis team, they run with $(A, B) \approx (30, 2^{40})$ or $(A, B) \approx (50, 2^{50})$. It is yet unclear how to bound the precomputation. Apart from this, this suggests that the MinRoot round function may for example be $(t, p, T, 35, 2^{35}, \varepsilon)$-sequential or $(t, p, T, 55, 2^{45}, \varepsilon)$-sequential for some small ε and for some appropriate T. This is still worse than what is claimed in the MinRoot specification, but at least the definition allows to be explicit on what the number of allowed processors and the role of A (called A_{max} in the presentations) is. These claims are mere suggestions and may not be accurate anymore in light of newer cryptanalysis results.

Comparison with "Concrete security" of MinRoot paper. Compared with the "concrete security" definition of [KMT22, Section 2.4], α is now ε, σ is now $1/A$, T_0 is now T, and n_E is roughly B.

Alternative but discarded definition. One notable alternative consideration was to swap the reasoning. It would state that, "if there exists an adversary that can run a system in t' time and p' processors, then there must be a legitimate system that can run it it $t = t'A$ time and in p'/B processors." However, the difference is that the adversary will get a random challenge and the legitimate system operates for any x. Thus approach does not seem to work as desired.

Bibliography

[Adl79] Leonard M. Adleman. 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*, pages 55–60. IEEE Computer Society, 1979.

[AH99] Leonard M. Adleman and Ming-Deh A. Huang. Function field sieve method for discrete logarithms over finite fields. *Inf. Comput.*, 151(1-2):5–16, 1999.

[AK88] Leonard M. Adleman and Kireeti Kompella. Using smoothness to achieve parallelism (abstract). In *STOC*, pages 528–538. ACM, 1988.

[AMP+18] Amir H. Atabaki, Sajjad Moazeni, Fabio Pavanello, Hayk Gevorgyan, Jelena Notaros, Luca Alloatti, Mark T. Wade, Chen Sun, Seth A. Kruger, Huaiyu Meng, Kenaish Al Qubaisi, Imbert Wang, Bohan Zhang, Anatol Khilo, Christopher V. Baiocco, Miloš A. Popović, Vladimir M. Stojanović, and Rajeev J. Ram. Integrating photonics with silicon nanoelectronics for the next generation of systems on a chip. *Nature*, 556(7701):349–354, Apr 2018.

[Bac88] Eric Bach. How to generate factored random numbers. *SIAM J. Comput.*, 17(2):179–193, 1988.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 757–788. Springer, 2018.

[BK82] Richard P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, 31(3):260–264, 1982.

[BR85] Richard P Brent and HT Rung. A systolic algorithm for integer gcd computation. In *1985 IEEE 7th Symposium on Computer Arithmetic (ARITH)*, pages 118–125. IEEE, 1985.

[BS07] Daniel J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit Chinese remainder theorem. *Math. Comput.*, 76(257):443–454, 2007.

[CHM20] Dror Chawin, Iftach Haitner, and Noam Mazor. Lower bounds on the time/memory tradeoff of function inversion. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part III*, volume 12552 of *Lecture Notes in Computer Science*, pages 305–334. Springer, 2020.

[Ear65] J Earle. Latched carry-save adder. *IBM Technical Disclosure Bulletin*, 7(10):909–910, 1965.

[Gor93] Daniel M. Gordon. Discrete logarithms in *GF(P)* using the number field sieve. *SIAM J. Discret. Math.*, 6(1):124–138, 1993.

[KMT22] Dmitry Khovratovich, Mary Maller, and Pratyush Ranjan Tiwari. Minroot: Candidate sequential function for ethereum VDF. *IACR Cryptol. ePrint Arch.*, page 1626, 2022.

[Len87] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[LW15] Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptol. ePrint Arch.*, page 366, 2015.

[PAB+22] Enrico Piccione, Samuele Andreoli, Lilya Budaghyan, Claude Carlet, Siemen Dhooghe, Svetla Nikova, George Petrides, and Vincent Rijmen. An optimal universal construction for the threshold implementation of bijective s-boxes. *IACR Cryptol. ePrint Arch.*, page 1141, 2022.

[Pan96] Victor Y. Pan. Parallel computation of polynomial GCD and some related parallel computations over abstract fields. *Theor. Comput. Sci.*, 162(2):173–223, 1996.

[PH78] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance (corresp.). *IEEE Trans. Inf. Theory*, 24(1):106–110, 1978.

[RSS20] Lior Rotem, Gil Segev, and Ido Shahaf. Generic-group delay functions require hidden-order groups. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 155–180. Springer, 2020.

[Sha71] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proc. Symp. Math. Soc., 1971*, volume 20, pages 415–440, 1971.

[Sha99] Adi Shamir. Factoring large numbers with the twinkle device (extended abstract). In *CHES*, volume 1717 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 1999.

[SHT22] Kavya Sreedhar, Mark Horowitz, and Christopher Torng. A fast large-integer extended GCD algorithm and hardware design for verifiable delay functions and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):163–187, 2022.

[Sor94] Jonathan Sorenson. Two fast GCD algorithms. *J. Algorithms*, 16(1):110–144, 1994.

[Sta20] StarkWare. Presenting: VeeDo. `https://medium.com/starkware/presenting-veedo-e4bbff77c7ae`, june 2020.

[Val82] Leslie G. Valiant. A scheme for fast parallel communication. *SIAM J. Comput.*, 11(2):350–361, 1982.

[Wal64] Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electron. Comput.*, 13(1):14–17, 1964.

[Wan81] Paul S. Wang. A p-adic algorithm for univariate partial fractions. In Paul S. Wang, editor, *Proceedings of the Symposium on Symbolic and Algebraic Manipulation, SYMSAC 1981, Snowbird, Utah, USA, August 5-7, 1981*, pages 212–217. ACM, 1981.

[Wie04] Michael J. Wiener. The full cost of cryptanalytic attacks. *J. Cryptol.*, 17(2):105–124, 2004.