Cryptanalysis of Algebraic Verifiable Delay Functions

Alex Biryukov Ben Fisch Gottfried Herold Dmitry Khovratovich
Gaëtan Leurent María Naya-Plasencia Benjamin Wesolowski

CRYPTO 2024

Verifiable Delay Functions [Boneh, Bonneau, Bünz & Fisch, CRYPTO'18]

Function public function $f : X \to Y$

Delay $f(x)$ cannot be computed faster than T, for random x (security claim)

Verifiable comes with a proof for fast verification of correctness

Security claim: sequentiality

- There exist an evaluation algorithm in time $(1+\varepsilon)T$ with few processors
- There is no evaluation algorithm faster than T, even with many processors

Example usage: Randomness beacons in blockchains

- Users contribute inputs x_i
- A party computes hash of inputs and publishes output

- Problem: last user to contribute can brute-force output to bias it
- Biasing the output requires fast evaluation ⇒ VDF

Algebraic VDF

- Construct hash function using algebraic operations in a large field $\mathbb{F}_p$
  - Additions, multiplications
  - Huge number of rounds to make it slow (*e.g.* 2^{40})
- Use SNARK to make it verifiable
- S-Box candidate: a-th root for small a $\qquad x \mapsto \sqrt[a]{x}$
  - Permutation when $\gcd(a, p-1) = 1$
  - High degree, somewhat slow, efficient ZK proofs

Evaluation of $\sqrt[a]{\cdot}$

- Fermat's little theorem: $\sqrt[a]{x} = x^{1/a \bmod p-1}$
- Fast exponentiation: $\log_2(p)$ squaring and multiply
- Latency $\log_2(p)$ with 2 processors

ZK proof for $\sqrt[a]{\cdot}$

- $y = \sqrt[a]{x} \iff y^a = x$
- $y^a = x$ has low degree

Algebraic VDF

- Construct hash function using algebraic operations in a large field $\mathbb{F}_p$
  - Additions, multiplications
  - Huge number of rounds to make it slow (*e.g.* 2^{40})
- Use SNARK to make it verifiable
- S-Box candidate: a-th root for small a $\qquad x \mapsto \sqrt[a]{x}$
  - Permutation when $\gcd(a, p-1) = 1$
  - High degree, somewhat slow, efficient ZK proofs

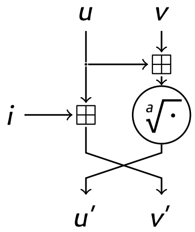
Examples

- Sloth++ [Boneh, Bonneau, Bünz & Fisch, CRYPTO'18]
- Veedo [StarkWare, 2020]
- MinRoot [Khovratovich, Maller & Tiwari, ePrint 2022/1626]

MinRoot [Khovratovich, Maller & Tiwari, ePrint 2022/1626]

MinRoot

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



- Two elements in $\mathbb{F}_p$ — $p = 2^{254} + 2^{32} \cdot \texttt{0x224698fc094cf91b992d30ed} + 1$
- Using 5-th root in $\mathbb{F}_p$ — $a = 5$

- Planned for use in Ethereum's consensus protocol and Filecoin
- ASIC developed by Supranational

Security claim

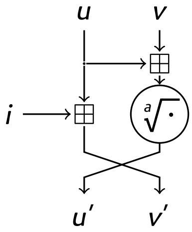
- Even with 2^{128} processors and 2^{128} memory, speedup at most 2

VDF cryptanalysis

- Slow hash function, over a large field, with an unusual security claim
- Security claim: high delay even with massive parallelism and precomputation
  - Delay is measured as *latency*: time between receiving input and computing output
  - Complexity in number of operations can be large

Cryptanalysis targets

1. Can we find shortcuts in the iteration of n rounds?
2. Can we compute the round function faster in parallel?



Computing roots in $\mathbb{F}_p$

- We focus on root computation: $x \mapsto \sqrt[a]{x}$
  - Most expensive part of the round function
- Can we compute root with low latency using many processors and precomputation?
  - Fast exponentiation has latency $\log_2(p)$ squarings
- We consider two techniques to compute root with low latency
  1. Precomputation
  2. Smoothness

Randomization

- Roots and power function are homomorphisms: $\sqrt[a]{xy} = \sqrt[a]{x} \cdot \sqrt[a]{y}$
- Given input x, we can randomize it with r: $y = x \cdot r^a \bmod p$
- And deduce root of x from root of y: $\sqrt[a]{x} = \sqrt[a]{y} \cdot r^{-1} \bmod p$
- Precompute r^a and r^{-1}

Computing roots in $\mathbb{F}_p$

- We focus on root computation: $x \mapsto \sqrt[a]{x}$
  - Most expensive part of the round function
- Can we compute root with low latency using many processors and precomputation?
  - Fast exponentiation has latency $\log_2(p)$ squarings
- We consider two techniques to compute root with low latency
  1. Precomputation
  2. Smoothness

Randomization

- Roots and power function are homomorphisms: $\sqrt[a]{xy} = \sqrt[a]{x} \cdot \sqrt[a]{y}$
- Given input x, we can randomize it with r: $y = x \cdot r^a \bmod p$
- And deduce root of x from root of y: $\sqrt[a]{x} = \sqrt[a]{y} \cdot r^{-1} \bmod p$
- Precompute r^a and r^{-1}

Idea 1: Precomputation

- Precompute roots of small values $T[i] = \sqrt[a]{i}$ for $i < \sqrt{p}$
- Randomization: $y = x \cdot r^a \bmod p$, with $\sqrt{p}$ different values r
  - With high probabiliy, match between y and i
  - Fetch $\sqrt[a]{y} = T[y]$ and deduce $\sqrt[a]{x}$
- Similar to baby-step giant-step algorithm for discrete logarithm

Online algorithm

Input: $x \in \mathbb{F}_p$

  for $0 \leq r < \sqrt{p}$ **do**

    $y \leftarrow x \cdot r^a \bmod p$

    if $y \leq \sqrt{p}$ **then**

      return $\sqrt[a]{y} \cdot r^{-1} \bmod p$

Idea 1: Precomputation

- Precompute roots of small values $T[i] = \sqrt[a]{i}$ for $i < \sqrt{p}$
- Randomization: $y = x \cdot r^a \bmod p$, with $\sqrt{p}$ different values r
  - With high probabiliy, match between y and i
  - Fetch $\sqrt[a]{y} = T[y]$ and deduce $\sqrt[a]{x}$
- Similar to baby-step giant-step algorithm for discrete logarithm
- Parallel implementation
  - $\sqrt{p}$ processors, each processor only does a few operation
  - $\sqrt{p}$ memory (only one CPU makes an access)
  - Latency: 2 `Mul` + 1 `Lookup`

$r = 2$	$r = 3$	…	r
$y \leftarrow x \cdot 2^a \bmod p$	$y \leftarrow x \cdot 3^a \bmod p$		$y \leftarrow x \cdot r^a \bmod p$
If $y \le \sqrt{p}$	**If** $y \le \sqrt{p}$		**If** $y \le \sqrt{p}$
Ret $\sqrt[a]{y} \cdot 2^{-1} \bmod p$	**Ret** $\sqrt[a]{y} \cdot 3^{-1} \bmod p$		**Ret** $\sqrt[a]{y} \cdot r^{-1} \bmod p$

Idea 1: Precomputation

- Precompute roots of small values $T[i] = \sqrt[a]{i}$ for $i < \sqrt{p}$
- Randomization: $y = x \cdot r^a \bmod p$, with $\sqrt{p}$ different values r
  - With high probabiliy, match between y and i
  - Fetch $\sqrt[a]{y} = T[y]$ and deduce $\sqrt[a]{x}$
- Similar to baby-step giant-step algorithm for discrete logarithm
- Parallel implementation
  - $\sqrt{p}$ processors, each processor only does a few operation
  - $\sqrt{p}$ memory (only one CPU makes an access)
  - Latency: 2 `Mul` + 1 `Lookup`
- Concrete parameters

  $p \approx 2^{256}$
  - 2^{128} processors, 2^{128} memory, speedup 32

Idea 2: Smoothness

- Precompute roots of small primes $q_i \leq B$
- Randomization: $y = x \cdot r^a \bmod p$
  - Lift y to integers, and check if B-smooth: $y = \prod q_i$ with $q_i \leq B$
  - Deduce $\sqrt[a]{y} = \prod \sqrt[a]{q_i}$
- Similar to index calculus for discrete logarithm
  - Probability of $y \leq p$ to be B-smooth $\approx \rho(\log_2(p)/\log_2(B))$ [Dickman, 1930]
  - Sub-exponential complexity

Online algorithm

Input: $x \in \mathbb{F}_p$
loop
$y \leftarrow x \cdot r^a \bmod p$
if $y = \prod q_i$, with $q_i \leq B$ **then**
return $\sqrt[a]{y} \cdot r^{-1} \bmod p$

Idea 2: Smoothness

- Parallel implementation
  - Groups of $\pi(B)$ processors (subexponential complexity)
  - Latency: 1 `Mul` + 1 `TrialDiv` + a few `Mul`

$r = 2$

$y \leftarrow x \cdot 2^a \bmod p$

$\forall q : z_q \leftarrow 1$

$q = 2$
If $2 \mid y$
$z_2 \leftarrow \sqrt[a]{2}$

$q = 3$
If $3 \mid y$
$z_3 \leftarrow \sqrt[a]{3}$

...

q
If $q \mid y$
$z_q \leftarrow \sqrt[a]{q}$

$z \leftarrow \prod z_q \bmod p$

If $z^a \bmod p = y$

Ret $z \cdot 2^{-1} \bmod p$

...

r

$y \leftarrow x \cdot r^a \bmod p$

$\forall q : z_q \leftarrow 1$

...

q
If $q \mid y$
$z_q \leftarrow \sqrt[a]{q}$

$z \leftarrow \prod z_q \bmod p$

If $z^a \bmod p = y$

Ret $z \cdot r^{-1} \bmod p$

Idea 2: Smoothness

- Precompute roots of small primes $q_i \le B$
- Randomization: $y = x \cdot r^a \bmod p$
  - Lift y to integers, and check if B-smooth: $y = \prod q_i$ with $q_i \le B$
  - Deduce $\sqrt[a]{y} = \prod \sqrt[a]{q_i}$
- Similar to index calculus for discrete logarithm
  - Probability of $y \le p$ to be B-smooth $\approx \rho(\log_2(p)/\log_2(B))$ [Dickman, 1930]
  - Sub-exponential complexity
- Parallel implementation
  - Groups of $\pi(B)$ processors (subexponential complexity)
  - Latency: 1 `Mul` + 1 `TrialDiv` + a few `Mul`
- Concrete parameters $p \approx 2^{256}$
  - $B = 2^{35}$, $\pi(B) \approx 2^{30.5}$, $\Pr[\text{smooth}] \approx 2^{-21.6}$
  - $2^{54.5}$ processors, speedup 42
- Already known for parallel modular exponentiation! [Adleman & Kompella, STOC'88]
  - Coming next: improvements to this technique, and application to concrete VDF

Improvements 1: Almost-smoothness

- Almost-smoothness: Assume that y has small factors, and a medium factor:

$$y = q' \cdot \prod q_i, \quad \text{with } q_i \le B, q' \le B'$$

  - Remove small factors with trial division, check is remaining value is small
  - Deduce $\sqrt[a]{y} = \sqrt[a]{q'} \cdot \prod \sqrt[a]{q_i}$
- Precompute and tabulate roots of medium primes $q' \le B'$
- Parallel implementation
  - Latency: 2 `Mul` + 1 `TrialDiv` + 1 `Lookup` + a few `Mul`
- Concrete parameters $p \approx 2^{256}$
  - $B = 2^{32}$, $B' = 2^{65}$ $\Pr[\text{almost-smooth}] \approx 2^{-18}$
  - 2^{48} processors, $2^{59.5}$ memory, speedup 20

Improvements 1: Almost-smoothness

- Parallel implementation
  - Latency: 2 `Mul` + 1 `TrialDiv` + 1 `Lookup` + a few `Mul`

$r = 2$

$y \leftarrow x \cdot 2^a \bmod p$
$\forall q :\ z_q \leftarrow 1$

$q = 2$	$q = 3$	…	q
If $2 \mid y$ $z_2 \leftarrow \sqrt[a]{2}$	**If** $3 \mid y$ $z_3 \leftarrow \sqrt[a]{3}$		**If** $q \mid y$ $z_q \leftarrow \sqrt[a]{q}$

$z \leftarrow \prod z_q \bmod p$
$w \leftarrow y/z$
If $w \le B'$
  Ret $z \cdot \sqrt[a]{w} \cdot 2^{-1} \bmod p$

…

r

$y \leftarrow x \cdot r^a \bmod p$
$\forall q :\ z_q \leftarrow 1$

…	q
	If $q \mid y$ $z_q \leftarrow \sqrt[a]{q}$

$z \leftarrow \prod z_q \bmod p$
$w \leftarrow y/z$
If $w \le B'$
  Ret $z \cdot \sqrt[a]{w} \cdot r^{-1} \bmod p$

Improvements 1: Almost-smoothness

- Almost-smoothness: Assume that y has small factors, and a medium factor:

$$y = q' \cdot \prod q_i \,, \quad \text{with } q_i \leq B, q' \leq B'$$

  - Remove small factors with trial division, check is remaining value is small
  - Deduce $\sqrt[a]{y} = \sqrt[a]{q'} \cdot \prod \sqrt[a]{q_i}$
- Precompute and tabulate roots of medium primes $q' \leq B'$
- Parallel implementation
  - Latency: 2 `Mul` + 1 `TrialDiv` + 1 `Lookup` + a few `Mul`
- Concrete parameters

$p \approx 2^{256}$

  - $B = 2^{32}$, $B' = 2^{65}$ Pr[almost-smooth] $\approx 2^{-18}$
  - 2^{48} processors, $2^{59.5}$ memory, speedup 20

Improvements 2: Prefiltering

- Observation: Randomizing step uses a single CPU per group
- Improvement: Try a set of values r_i, keep most promising $y_i = x \cdot r_i \bmod p$ in each group
  - Simple filter: keep smallest y_i
  - Advanced filter: trial division with small bound $B_0 < B$, keep y with large B_0-smooth part
- Filtering improves the probability that y_i is (almost)-smooth
- Parallel implementation
  - Latency: 2 `Mul` + 2 `TrialDiv` + 1 `Lookup` + a few `Mul`
- Concrete parameters $p \approx 2^{256}$
  - $B = 2^{32}$, $B' = 2^{65}$, $B_0 = 2^{20}$, Pr[almost-smooth | filter] $\approx 2^{-9.5}$
  - 2^{40} processors, $2^{59.5}$ memory, speedup 18

Improvements 2: Prefiltering

- Parallel implementation
  - Latency: 2 `Mul` + 2 `TrialDiv` + 1 `Lookup` + a few `Mul`

$r_i = 2$
$y_i \leftarrow x \cdot 2^a \bmod p$
If y_i **has smooth part**
$y \leftarrow y_i$, $r \leftarrow 2$

$r_i = 3$
$y_i \leftarrow x \cdot 3^a \bmod p$
If y_i **has smooth part**
$y \leftarrow y_i$, $r \leftarrow 3$

...

$q = 2$
If $2 \mid y$
$z_2 \leftarrow \sqrt[a]{2}$

$q = 3$
If $3 \mid y$
$z_3 \leftarrow \sqrt[a]{3}$

...

q
If $q \mid y$
$z_q \leftarrow \sqrt[a]{q}$

$z \leftarrow \prod z_q \bmod p$
$w \leftarrow y/z$
If $w \leq B'$
Ret $z \cdot \sqrt[a]{w} \cdot 2^{-1} \bmod p$

...

...

r_i
$y_i \leftarrow x \cdot r_i^a \bmod p$
If y_i **has smooth part**
$y \leftarrow y_i$, $r \leftarrow r_i$

...

q
If $q \mid y$
$z_q \leftarrow \sqrt[a]{q}$

$z \leftarrow \prod z_q \bmod p$
$w \leftarrow y/z$
If $w \leq B'$
Ret $z \cdot \sqrt[a]{w} \cdot 2^{-1} \bmod p$

Improvements 2: Prefiltering

- Observation: Randomizing step uses a single CPU per group
- Improvement: Try a set of values r_i, keep most promising $y_i = x \cdot r_i \bmod p$ in each group
  - Simple filter: keep smallest y_i
  - Advanced filter: trial division with small bound $B_0 < B$, keep y with large B_0-smooth part
- Filtering improves the probability that y_i is (almost)-smooth
- Parallel implementation
  - Latency: 2 `Mul` + 2 `TrialDiv` + 1 `Lookup` + a few `Mul`
- Concrete parameters $p \approx 2^{256}$
  - $B = 2^{32}$, $B' = 2^{65}$, $B_0 = 2^{20}$, Pr[almost-smooth | filter] $\approx 2^{-9.5}$
  - 2^{40} processors, $2^{59.5}$ memory, speedup 18

Improvement 3: Parallel smoothness test

- Additive randomization: $y = x + rp$ (as integers), instead of $y = x \cdot r^a \bmod p$
  - Lift y to integers, and check if B-smooth: $y = \prod q_i$ with $q_i \leq B$
  - Deduce $\sqrt[a]{x} = \sqrt[a]{y} = \prod \sqrt[a]{q_i}$
- Advantage: we can test all values y for smoothness simultaneously
  - $q \mid x + rp \iff r \equiv -x \cdot p^{-1} \bmod q$
  - Precompute $p^{-1} \bmod q$
- Parallel implementation
  - Latency: 2 `Mul` + 1 `ModRed` + 1 `Lookup` + a few `Mul`
- Concrete parameters $p \approx 2^{256}$
  - $B = 2^{32}$, $B' = 2^{45}$, $B_0 = 2^{20}$, Pr[almost-smooth | filter] $\approx 2^{-24}$
  - 2^{29} processors, 2^{40} memory, speedup 20

Improvement 3: Parallel smoothness test

- Parallel implementation
  - Latency: 2 `Mul` + 1 `ModRed` + 1 `Lookup` + a few `Mul`

$\forall q, \forall i : z_q^i \leftarrow 1$

$q = 2$

$\bar{x} \leftarrow x \bmod 2$
ForAll $i \equiv -\bar{x} \cdot p^{-1} \bmod 2$
$z_2^i \leftarrow \sqrt[a]{2}$

$q = 3$

$\bar{x} \leftarrow x \bmod 3$
ForAll $i \equiv -\bar{x} \cdot p^{-1} \bmod 3$
$z_3^i \leftarrow \sqrt[a]{3}$

…

q

$\bar{x} \leftarrow x \bmod q$
ForAll $i \equiv -\bar{x} \cdot p^{-1} \bmod q$
$z_q^i \leftarrow \sqrt[a]{q}$

$r = 0, y = x$

$z \leftarrow \prod_q z_q^0 \bmod p$
$w \leftarrow y/z$
If $w \leq B'$
Ret $z \cdot \sqrt[a]{w} \bmod p$

$r = 1, y = x + p$

$z \leftarrow \prod_q z_q^1 \bmod p$
$w \leftarrow y/z$
If $w \leq B'$
Ret $z \cdot \sqrt[a]{w} \bmod p$

…

$r, \quad y = x + rp$

$z \leftarrow \prod_q z_q^r \bmod p$
$w \leftarrow y/z$
If $w \leq B'$
Ret $z \cdot \sqrt[a]{w} \bmod p$

Improvement 3: Parallel smoothness test

- Additive randomization: $y = x + rp$ (as integers), instead of $y = x \cdot r^a \bmod p$
  - Lift y to integers, and check if B-smooth: $y = \prod q_i$ with $q_i \leq B$
  - Deduce $\sqrt[a]{x} = \sqrt[a]{y} = \prod \sqrt[a]{q_i}$
- Advantage: we can test all values y for smoothness simultaneously
  - $q \mid x + rp \iff r \equiv -x \cdot p^{-1} \bmod q$
  - Precompute $p^{-1} \bmod q$
- Parallel implementation
  - Latency: 2 `Mul` + 1 `ModRed` + 1 `Lookup` + a few `Mul`
- Concrete parameters

  $p \approx 2^{256}$
  - $B = 2^{32}$, $B' = 2^{45}$, $B_0 = 2^{20}$, Pr[almost-smooth | filter] $\approx 2^{-24}$
  - 2^{29} processors, 2^{40} memory, speedup 20

Application to MinRoot and Veedo

- Speedup of root computation directly applicable to MinRoot and Veedo
  - Various trade-offs between latency and number of processors
  - More improvements in the paper
- Concrete parameters for MinRoot ($p \approx 2^{256}$):

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

Application to Sloth++

- Sloth++ uses square roots in $\mathbb{F}_{p^2}$
  - Smoothness not directly applicable in $\mathbb{F}_{p^2}$
- Assume $\mathbb{F}_{p^2}$ is constructed as $\mathbb{F}_p[X]/\left(X^2 + \alpha\right)$ (elements are polynomials)
- Square root $z_0 + z_1 X$ of $b_0 + b_1 X$ satisfies:

$$\left(z_0 + z_1 X\right)^2 = b_0 + b_1 X \iff \begin{cases} 2z_0 z_1 & = b_1 \\ z_0^2 - \alpha z_1^2 & = b_0 \end{cases}$$

$$\iff \begin{cases} z_0 & = b_1 / 2z_1 \quad \text{(assuming } z_1 \neq 0\text{)} \\ \frac{a_1^2}{4z_1^2} - \alpha z_1^2 & = b_0 \quad \Rightarrow \text{quadratic equation in } z_1^2 \end{cases}$$

- Solve with quadratic formula, deduce z_1^2 then z_1 by computing square roots in $\mathbb{F}_p$.

Practical limitations

- In theory, this clearly breaks the security model
- In practice, communication is the bottleneck

- We need a billion CPU, with high speed communication
  - At each round, one CPU computes the root and sends result to all CPUs
  - Communication must be faster than computing root naively: 230ns (Supranational)

- Obviously not practical with current technology
- Does not seem to break laws of physics

- More work needed to evaluate practical impact

Conclusion

- Computing roots in $\mathbb{F}_p$ is not sequential
  - Various trade-offs between latency and number of processors
  - Breaks security claims of MinRoot: speedup 20 with 2^{29} CPU and 2^{40} memory
  - Almost practical for Veedo (128-bit prime): 2^{13} CPU 2^{40} memory
  - Extension to $\mathbb{F}_{p^2}$ (Sloth++)
- Strong link to discrete logarithm
  - Techniques similar to DL algorithms
  - Reduction from a class of parallel power-function algorithms to DL
- Open questions
  - Can we use more advanced discrete logarithm algorithm in this context? (ECM, NFS, ...)
  - What is the difficulty of parallel discrete logarithm?

Additional slides

Possible countermeasures

Modeling latency

Possible countermeasures for VDF construction

1. Make a weaker delay claim
   - 1 operation per round rather than $\log_2(p)$
2. Use $x \mapsto x^a$ instead of $x \mapsto \sqrt[a]{x}$ for the S-Box
   - Warning: some ideas for parallel evaluation of low-degree powers in the paper
3. Use a larger prime
   - Number of processors for our attack is sub-exponential
4. Use more complex groups
   - Index calculus only works in $\mathbb{F}_p$, but more advanced algorithms might be applicable

- More cryptanalysis needed!

Modeling latency

- Simple model for the latency of operations
  - Concrete values strongly dependent on architecture and technology
- Basic operations have latency $\mathcal{O}(\log(\log(p)))$ using optimized hardware $\Rightarrow$ 1 unit
  - Modular addition (up to $\log(p)$ operands)
  - Modular multiplication, Multiply-and-add
  - Trial division by a constant
- Lookup in a small table with k entries has latency $\log(k)$ $\Rightarrow$ 1 unit if $k \lesssim \log_2(p)$
- Memory access has larger latency $\Rightarrow \approx 6$ units
- We ignore latency of communication
  - $\mathcal{O}(\log(n))$ latency for n processors with hypercube tolopology [Valiant, 1982]