VDF Proving with SnarkPack

Verifiable delay functions are a core building block to the Eth2 infrastructure. They assume the existence of a deterministic function f

such that $f(x) = y$

takes at least time t

to compute. They further require a short proof π

attesting to the fact that $f(x)$

has been computed correctly. This is to allow for fast verification.

The purpose of this document is to discuss how SnarkPack can be used to generate the proof π

. Other options include using hidden order groups such as 1 or 2, STARKs, or recursive SNARKs such as Halo. The code relating to this post can be found here.

At a high level, the function f

is iterative and essentially computes the same smaller function over and over again: so $F(x) = f(f(f(f(\dots)))) = y$

. We can prove with a general purpose Snark that $F(x) = y$

directly. However, this is expensive to do, to the point where proving correct computation of the function takes many times longer than the computation itself, even using the best hardware and a high degree of parallelism. Our idea instead is to prove with a Groth16 Snark that e.g. $f(f(f(x_1))) = x_2$

, and then that $f(f(f(x_2))) = x_3$

, and continue until enough iterations of the function have been computed. This presents two advantages: (1) the computation of f(f(f(x))) is much smaller that the computation of $F(X)$

so the Groth16 prover is much easier to parallelise; (2) we can begin computing a proof for $f(f(f(x)))$

before we know the full outcome $F(x) = y$

. Once we have our numerous “in between” proofs, we can then aggregate these using SnarkPack into one small proof for the entire computation. SnarkPack is already heavily parellelised and so our aggregator can take advantage of this.

SnarkPack has the following advantages for a VDF proof:

- The aggregation prover is concretely very fast. Thus it does not present a bottleneck even though it can only be computed after the full computation is complete.
- The prover is easy to parellise. The prover can first compute F(x) by keeping the intermediate results and compute in parallel all the Groth16 proofs.
- The majority of intermediary results do not need to be stored meaning that the approach is memory efficient.
- The verifier is succinct and runs in time $\mathcal{O}(\log(\frac{n}{t}))$

where n

is the total number of iterations of f

and t

is the number of iterations of f

ran inside each Groth16 proof.

- The proof size is succinct and equal to $\mathcal{O}(\log(\frac{n}{t}))$

.

- There already exist high quality implementations of Groth16.

SnarkPack has the following disadvantages for a VDF proof:

- The proof consists of target group elements, and the verifier must compute target group exponentiations.
- The Groth16 proofs require a trusted setup. We would need to gather a set of at least 100 participants, but preferably more like 1000 to take part. Coordinating trusted setups has been done before by Zcash, Aztec, Celo, Filecoin, Perpetual Powers of Tau and some others. Thus we know that it is possible but that it is unlikely to be easy.
- The Groth16 trusted setup is circuit specific therefore we must be certain of our constraints before it is run, and we will not be able to make changes after it has been run. Potentially the damage of this downside is limitted in our sloth application because the specialist hardware is already circuit specific.
- The solution is not quantum safe.

Technical

We use the MinRoot function by Feist and Khovratovich. Other candidates include the original Sloth function by Lenstra and Wesolowski, Sloth++ by Boneh et al., and Veedo by Starkware. Note that the original MinRoot suggestion computed cube-roots. We compute fifth-roots because in the field Fr of BLS12-381 the cube-root does not always exist.

In the MinRoot VDF we have that a single iteration of f

is given by

f(x_i, y_i) = ((x_i + y_i)^{1/5}, x_i)

Then the MinRoot VDF computes f

over n

iterations where each new iteration takes as input the result of the previous iteration.

minroot(x_0, y_0): for i in 0..n: (x_{i+1}, y_{i+1}) = ((x_i + y_i)^{1/5}, x_i) mod p return (x_n, y_n)

Proving with Groth16

We prove that minroot has been correctly computed using Groth16 implemented over BLS381-12. To do this we must encode minroot into the R1CS constraint system.

Constraint System

Public Inputs:

We take as public inputs (x_0, y_0, x_n, y_n)

where (x_0, y_0)

are the first inputs to f

and (x_n, y_n)

are the final outputs of f

.

Constraints:

We prove the following constraints:

1. The first witness value is equal to x_0
2. The second witness value is equal to y_0
3. The penultimate witness value is equal to x_n
4. The final witness value is equal to y_n
5. For 0 < i < n

, a temporary witness value \mathsf{tmp2}_{i}

is equal to x_{i+1}^2

.

1. For $0 < i < n$, a temporary witness value $\mathsf{tmp3}_{i}$ is equal to $\mathsf{tmp2}_{i}^2$.

1. For $0 < i < n$, we have that $(x_i + y_i) = x_{i+1} * \mathsf{tmp3}_{i}$.

Efficiency:

Our encoding requires $3n + 4$ constraints where n is the number of iterations of f. There are 5 public inputs (including an initial public input that is always set to 1).

Efficiency

Using (a fork of) the Bellman library in rust we run the Groth16 prover on an Intel Core i5 with 2.3 GHz using $n = 10000$. We find that the computation of minroot took on average 106 milliseconds over 128 trials. In comparison the computation of the Groth16 proof toook on average 1345 milliseconds. Thus it took 12.7 times longer to prove than to compute. We will not make any meaningful conclusions from these numbers because they do not indicate the performance of specialist hardware.

Aggregating with SnarkPack

From our Groth16 provers, we obtain a list of inputs, outputs, and proofs of the form

$$[(1, x_0, y_0, x_1, y_1), \pi_0], [(1, x_1, y_1, x_2, y_2), \pi_1], \ldots, [(1, x_{m-1}, y_{m-1}, x_m, y_m), \pi_{m-1}]$$

where π_i is a Groth16 proof attesting to the correctness of $(x_{i+1}, y_{i+1}) = \mathsf{minroot}(x_i, y_i)$. In other words (x_{i+1}, y_{i+1}) is the result of applying n iterations of f to (x_i, y_i).

Denoting the instances by

$$(1, a_0, b_0, c_0, d_0), \ldots, (1, a_{m-1}, b_{m-1}, c_{m-1}, d_{m-1})$$

we have the VDF evaluation is correct if and only if the following constraints hold

1. The first (a_0, b_0) are exactly equal to the public input of the VDF (x_0, y_0).

1. The last (c_{m-1}, d_{m-1})

are exactly equal to the public output of the VDF (x_m, y_m)

.

1. It is true that $(c_j, d_j) = \mathsf{minroot}(a_j,b_j)$
2. The jth

output (c_j , d_j)

is equal to the (j+1)th

input (a_{j+1}, b_{j+1})

.

The proofs $(\pi_{0}, \ldots, \pi_{m-1})$

attest to the fact that the third point holds. We look to aggregate the proofs using SnarkPack. Denote $\pi_j = (A_j, B_j, C_j)$

and $Z_{AB} = \prod_{j=0}^{m-1} e(A_j^{r^j}, B_j)$

and $Z_C = \prod_{j=0}^{m-1} C_j^{r^j}$

for some random value r

determined after $[(x_0, y_0, x_1, y_1), \pi_0]$

, $\ldots$

, $[(x_{m-1}, y_{m-1}, x_m, y_m), \pi_{m-1}]$

have been fixed. The proof π_{agg}

attests to the fact that Z_{AB}

and Z_C

have been computed correctly. The proof π_{agg}

is of size $\mathcal{O}(\log(n))$

and can be verified using $\mathcal{O}(\log(n))$

target group exponentiations.

We then have that the Groth16 verifier is satisfied if and only if

$EQ_1: \quad Z_{AB} = e(g^{\alpha \left(\sum_{j=0}^{m-1} r^j \right)}, h^{\beta}) e(\prod_{k = 0}^{\ell} S_k^{\sum_{j=0}^{m-1} a_{j,k} r^j}, h^{\gamma}) e(Z_C, h^{\delta})$

Handling Public Inputs to SnarkPack

From our SnarkPack prover, we obtain a list of inputs, outputs and aggregated proofs of the form

$[(x_0, y_0, x_1, y_1)], [(x_1, y_1, x_2, y_2)], \ldots, [(x_{m-1}, y_{m-1}, x_m, y_m)], Z_{AB}, Z_C, \pi_{\mathsf{agg}}$

where π_{agg}

attests to the correctness of Z_{AB}

and Z_C

. This still has size $\mathcal{O}(m)$

due to the inclusion of the instance values. We now look to avoid giving out the intermediary values

$(x_1, y_1), \ldots, (x_{m-1}, y_{m-1})$

in order to achieve logarithmic costs. However our verifier must remain confident that there exists such intermediary values satisfying our security requirements 1, 2, 3, and 4.

To do this we define the polynomials

$p(X) = x_0 + x_1 X + \cdots + x_{m-1} X^{m-1}$

and

$q(X) = y_0 + y_1 X + \cdots + y_{m-1} X^{m-1}$

according to the public instances and publicly give out the values

$z_1 = \sum_{j=0}^{m-1} x_{j} r^j = p(r) \quad \mbox{and} \quad z_2 = \sum_{j=0}^{m-1} y_{j} r^j = q(r)$

We design a constant sized proof π_{inputs}

(which we will describe shortly) that attests that z_1

and z_2

have been computed correctly. In other words it will enforce that $p(X)$ is a degree $m-1$

polynomial with x_0

as the zero'th coefficient and that $q(X)$

is a degree $(m-1)$

with y_0

as the zero'th coefficient.

To enforce that the last (c_{m-1}, d_{m-1})

are exactly equal to the public output of the VDF and that the jth

output (c_j , d_j)

is equal to the $(j+1)$th

input (a_{j+1}, b_{j+1})

, we have the verifier compute

$z_3 = \sum_{j=0}^{m-1} x_{j+1} r^j = p(r) \quad \mbox{and} \quad z_4 = \sum_{j=0}^{m-1} y_{j + 1} r^j = q(r)$

for itself. It can do this using the information (z_1, z_2, x_m, y_m)

because z_3

and z_4

are a shift of $p(r)$

and $q(r)$

respectively.

Prover

Then the VDF proof consists of the values

$(x_0, y_0), (x_m, y_m), Z_{AB}, Z_C, z_1, z_2,\pi_{\mathsf{agg}}, \pi_{\mathsf{inputs}}$

Verifier:

The VDF verifier checks that π_{agg}

and π_{inputs}

both verify and then computes two additional values

$z_3 = \frac{1}{r}(z_1 - x_0) + r^{m-1} x_m \quad \mbox{ and } \quad z_4 = \frac{1}{r}(z_2 - y_0) + r^{m-1} y_m$

It then sets $r_{\mathsf{sum}} = \sum_{j=0}^{m-1} r^j = \frac{r^m - 1}{r - 1}$

and checks that

$Z_{AB} = e(g^{\alpha r_{\mathsf{sum}} }, h^{\beta}) e(S_0^{r_{\mathsf{sum}}} S_1^{z_1}S_2^{z_2}S_3^{z_3}S_4^{z_4}, h^{\gamma}) e(Z_C, h^{\delta})$

The verifier returns true if and only if all checks pass. See that the verifiers equation corresponds exactly to EQ_1

.

Proving Correct Inputs

To prove that z_1

and z_2

are computed correctly we will use theKZG10 polynomial commitment scheme. The techniques used here are similar to those by Tyagi et al.. We set a polynomial commitment to $p(X)$

and $q(X)$

as

$\mathsf{com}_p = g^{ \sum_{j=0}^{m-1} x_j \tau^j} \quad \mbox{ and } \quad \mathsf{com}_q = g^{ \sum_{j=0}^{m-1} y_j \tau^j}$

where τ

is secret.

The commitments com_p

and com_q

must be chosen before the randomness r

used to compute Z_{AB}

and Z_C

is selected. This is to ensure that the polynomials $p(X)$

and $q(X)$

cannot depend on r

, which is equivalent to ensuring that $(x_0, y_0, x_1, y_1), \ldots, (x_{m-1}, y_{m-1}, x_m, y_m)$

are fixed before r

is determined. We thus include com_p

and com_q

in the Fiat-Shamir hash that is used to compute r

.

The prover now needs to show that

1. com_p

has zero coefficient x_0

and com_q

has zero coefficient y_0

.

1. com_p

and com_q

both have degree no higher than $m-1$

.

1. com_p

evaluates at r

to z_1

and com_q

evaluates at r

to z_2

.

To prove that com_p

has zero coefficient x_0

and com_q

has zero coefficient y_0

, the prover include values $W_{p,0}$

and $W_{q,0}$

equal to

$$W_{p,0} = g^{ \sum_{j=1}^{m-1} x_j \tau^{j-1}} \quad \mbox{ and } \quad W_{q,0} = g^{ \sum_{j=1}^{m-1} y_j \tau^{j-1}}.$$

The verifier checks that

$$e(\mathsf{com}p \cdot g^{-x_0}, h) = e(W{p,0}, h^{\tau}) \quad \mbox{ and } \quad e(\mathsf{com}q \cdot g^{-y_0}, h) = e(W{q,0}, h^{\tau}).$$

If these checks hold but x_0

and y_0

are not the zero'th coefficients of $p(X)$

and $q(X)$

respectively, then $W_{p,0}$

or $W_{q,0}$

will contain a polynomial with a non-zero coefficient in X^{-1}

. However $g^{\tau^{-1}}$

is not given out in the SRS and thus this cannot happen in the GGM.

To prove that com_p

and com_q

both have degree no higher than $m-1$

, the prover includes values $W_{p,d}$

and $W_{q,d}$

equal to

$$W_{p,d} = g^{ \sum_{j=0}^{m-1} x_j \tau^{j + d - m + 1 }} \quad \mbox{ and } \quad W_{q,d} = g^{ \sum_{j=0}^{m-1} y_j \tau^{j + d - m + 1}}$$

where d

is the maximum power of τ

given out in the SRS. The verifier checks that

e(\mathsf{com}p, h^{\tau^{d - m + 1}}) = e(W{p,d}, h) \quad \mbox{ and } \quad e(\mathsf{com}q , h^{\tau^{d - m + 1}}) = e(W{q,d}, h).

If these checks hold but p(X)

or q(X)

have degree greater than d

, then W_{p,d}

or W_{q,d}

will contain a polynomial with a non-zero coefficient of degree greater than X^{d}

. However this is a higher power of tau than the maximum given out in the SRS and thus this cannot happen in the GGM.

To prove that p(X)

evaluates to z_1

at r

and q(X)

evaluates to z_2

at r

, the prover computes KZG opening proofs. They set

\omega_{p,r}(X) = \frac{p(X) - z_1}{X - r} \quad \mbox{ and } \quad \omega_{q,r}(X) = \frac{q(X) - z_2}{X - r}

and compute

W_{p,r} = g^{ \omega_{p,r}(\tau)} \quad \mbox{ and } \quad W_{q,r} = g^{ \omega_{p,r}(\tau)}.

The verifier checks that

e(\mathsf{com}p \cdot g^{-z_1}, h) = e(W{p,r}, h^{\tau - r}) \quad \mbox{ and } \quad e(\mathsf{com}q \cdot g^{-z_2}, h) = e(W{q,r}, h^{\tau - r}).

Efficiency:

The proof

\pi_{\mathsf{inputs}} = (\mathsf{com}p, \mathsf{com}_q, W{p,0}, W_{q,0}, W_{p,d}, W_{q,d}, W_{p,r}, W_{q,r})

consists of 8

\mathbb{G}_1

elements. In BLS12-381 this is 381

bytes. The verifier computes 12

pairings. We have not optimised \pi_{\mathsf{inputs}}

because it is not a bottleneck, however there are some improvements that could be made in reducing the proof sizes and verifier time.

Some Preliminary Benchmarks

We implement the VDF prover and verifier using theBellperson library by Filecoin. In the table below we give some benchmarks. These were computed on an Intel Core i5 with 2.3 GHz.

The number of iterations of f

is the total number of times f

is computed. We use n = 10000

for all benchmarks (the number of iterations proved per Groth16 prover) and m = 32, 64, 128, 256, 512, 1024 (the number of proofs aggregated). This means the number of iterations of f varies between 320,000 and 10,240,000. The corresponding time to compute this many iterations of f is given in seconds.

We urge the reader not to take the prover time too seriously because this is not indicative of the time taken by specialist hardware. Nonetheless we have provided the total proving time, in seconds, as the time to generate all Groth16 proofs and to generate the aggregation and public input proofs. We additionally give the time to generate the aggregation and public input proofs separately because these must wait until the vdf completes before the prover can start.

The verifier time is given in milliseconds. It ranges between 58 ms and 90 ms and is growing logarithmically. This is as expected.

The proof size is given in kilobytes. This varies between 17.1 kbytes and 29.0 kbytes and is growing logarithmically.

iterations of f	compute time (s)	total prove time (s)	aggregation prove time (s)	verify time (ms)	proof size (kbytes)
320,000	3.5	43.2	0.51	58	17.1
640,000	6.8	85.7	1.05	69	20.1
1,280,000	13.5	174.0	1.87	75	

23.1

2,560,000

27.4

352.7

3.72

79

26.1

5,120,000

54.4

673.3

7.30

84

29.0

10,240,000

106.6

1,357.9

14.55

90

32.0

Some Estimated Benchmarks with ASICs and Parallelism

The following estimations were computed by Kelly Olson from Supranational.

In order to estimate the feasibility of using Sloth and SnarkPack as a production VDF, it is important to understand the potential performance of optimized implementations of the VDF evaluation and proof generation functions on alternative hardware platforms. First, in order to be secure and user friendly, VDF evaluation must be as fast as possible. Second, we can use a high degree of parallelism in generating the proofs. With this in mind, we look to understand the complexity and time spent to generate a VDF proof using SnarkPack.

Previous research on RSA VDFs has shown that the performance of an ASIC can be up to 200x faster than a comparable CPU implementation. However, since the modular multiplication operations in the BLS12-381 fields are much smaller than the 2048-bit operations in RSA groups, and therefore much closer to the native 64-bit word size of a modern CPU, we estimate that one round of the Sloth function could be performed in as little as 150 nanoseconds. The baseline implementation took approximately 10 microseconds to evaluate a single iteration of the Sloth function, so with an ASIC we expect an approximate 70x speedup in computing the VDF.

Let us assume arbitrary delay of one minute. While some use cases of VDFs may require a longer, or shorter, delay, the key point to understand is the relative performance between the evaluation and proving function. That is to say the proving must be sufficiently fast such that it does not add a significant amount of overhead on top of the evaluation function. In order to ensure a delay of at least one minute, approximately 425 million iterations of the Sloth function must be performed (assuming 150ns per iteration). Using the (3n+4) formula from above, this implies a total circuit complexity of approximately 1.27B constraints.

To understand the time it would take to prove such a large circuit, we can look at implementations used in practice by projects such as Filecoin. One of the most common SNARKs used on the Filecoin blockchain is a Proof-of-Replication proof, which requires the generation of 10 Groth16 proofs, each of size approximately 130 million, for a total of 1.3B constraints. Recent benchmarks from the Filecoin community suggest that this proof can be created in <4 minutes using a modern CPU and GPU system. This implies a ratio of approximately 1:4 for VDF evaluation to proving time for the proposed function. However, since this VDF construction relies on generating many small proofs which are then aggregated, the proving operation can be easily parallelized by adding more proving hardware, such as a system with 4 GPUs, or by distributing the individual proof generation work across multiple ‘workers’. Given the increase in GPU performance from generation to generation, potential unexplored optimizations, and these high level estimates, it is feasible that a VDF system using the

SnarkPack system could be used in a production environment today.

Acknowledgements

Thanks to Kelly Olson for writing the estimated benchmarks above, Nicolas Gailly for his feedback and help navigating the Bellperson library, and to Dmitry Khovratovich for discussions on representing Minroot in R1CS.