J Cryptol (2020) 33:2113–2147
https://doi.org/10.1007/s00145-020-09364-x

Journal of
CRYPTOLOGY

Efficient Verifiable Delay Functions

Benjamin Wesolowski
Univ. Bordeaux, CNRS, Bordeaux INP, IMB, UMR 5251, 33400 Talence, France
INRIA, IMB, UMR 5251, 33400 Talence, France
Cryptology Group, CWI, Amsterdam, The Netherlands
benjamin.wesolowski@math.u-bordeaux.fr

Communicated by Masayuki Abe

Received 21 October 2019 / Revised 27 July 2020
Online publication 9 September 2020

Abstract. We construct a verifiable delay function (VDF). A VDF is a function whose evaluation requires running a given number of sequential steps, yet the result can be efficiently verified. They have applications in decentralised systems, such as the generation of trustworthy public randomness in a trustless environment, or resource-efficient blockchains. To construct our VDF, we actually build a *trapdoor* VDF. A trapdoor VDF is essentially a VDF which can be evaluated efficiently by parties who know a secret (the trapdoor). By setting up this scheme in a way that the trapdoor is unknown (not even by the party running the setup, so that there is no need for a trusted setup environment), we obtain a simple VDF. Our construction is based on groups of unknown order such as an RSA group or the class group of an imaginary quadratic field. The output of our construction is very short (the result and the proof of correctness are each a single element of the group), and the verification of correctness is very efficient.

Keywords. Time-lock puzzle, VDF, Randomness beacon, RSA, Class group.

1. Introduction

We describe a function that is slow to compute and easy to verify: a *verifiable delay function* (henceforth, VDF) in the sense of [4].[1] These functions should be computable in a prescribed amount of time Δ, but not faster (the *time* measures an amount of sequential work, that is work that cannot be performed faster by running on a large number of parallel cores), and the result should be easy to verify (i.e. for a cost $\mathrm{polylog}(\Delta)$). These special functions are used in [25] (under the name of *slow-timed hash functions*) to construct a trustworthy randomness beacon: a service producing publicly verifiable random numbers, which are guaranteed to be unbiased and unpredictable. These randomness beacons, introduced by Rabin [30], are a valuable tool in a public, decentralised setting,

[1] The paper [4] was developed independently of the present work, yet we adopt their terminology for verifiable delay functions, for the sake of uniformity.

as it is not trivial for someone to flip a coin and convince their peers that the outcome was not rigged. A number of interesting applications of VDFs have recently emerged—see [4] for an overview. Most notably, they can be used to design resource-efficient blockchains [12], eliminating the need for massively power-consuming mining farms. VDFs play a key role in the Chia blockchain design (`chia.net`), and the Ethereum Foundation (`ethereum.org`) and Protocol Labs (`protocol.ai`) are teaming up to investigate the technology of VDFs which promise to play a key role in their respective platforms.

There is thereby a well-motivated need for an efficient construction. This problem was left open in [4], and we address it here with a new, simple, and efficient VDF.

1.1. *Contribution*

An efficient construction The starting point of our construction is the time-lock puzzle of Rivest, Shamir and Wagner [31]: given as input an RSA group $(\mathbf{Z}/N\mathbf{Z})^\times$, where N is a product of two large, secret primes, a random element $x \in (\mathbf{Z}/N\mathbf{Z})^\times$, and a timing parameter t, compute x^{2^t}. Without the factorisation of N, this task is assumed to require t sequential squarings in the group. More generally, one could work with any group G of unknown order. This construction is only a time-lock puzzle and not a VDF, because given an output y, there is no efficient way to verify that $y = x^{2^t}$.

The new VDF construction consists in solving an instance of the time-lock puzzle of [31], and computing a proof of correctness, which allows anyone to efficiently verify the result. Fix a timing parameter $\varDelta$, a security level k (say, 128,192, or 256), and a group G. Our construction has the following properties:

1. It is $\varDelta$-sequential (meaning that it requires $\varDelta$ sequential steps to evaluate) assuming the classic repeated squaring assumption of [31] in the group G.
2. It is sound (meaning that one cannot produce a valid proof for an incorrect output) under some group theoretic assumptions on G, believed to be true for RSA groups and class groups of quadratic imaginary number fields.
3. The output and the proof of correctness are each a single element of the group G (also, the output can be recovered from the proof and a $2k$-bit integer, so it is possible to transmit a single group element and a small integer instead of 2 group elements).
4. The verification of correctness requires essentially two exponentiations in the group G, with exponents of bit-length $2k$.
5. The proof can be produced in $O(\varDelta/\log(\varDelta))$ group operations.

For applications where a lot of these proofs need to be stored, widely distributed, and repeatedly verified, having very short and efficiently verifiable proofs is invaluable.

Following discussion about the present work at the August 2018 workshop at Stanford hosted by the Ethereum Foundation and the Stanford Center for Blockchain Research, we note that our construction features two other useful properties: the proofs can be *aggregated* and *watermarked*. Aggregating consists in producing a single short proof that simultaneously proves the correctness of several VDF evaluations. Watermarking consists in tying a proof to the evaluator's identity; in a blockchain setting, this allows to give credit (and a reward) to the party who spent time and resources evaluating the VDF. These properties are discussed in Sect. 7.

Note that the method we describe to compute the proof requires an amount $O(\Delta/\log(\Delta))$ group operations. Hence, there is an interval between the guaranteed sequential work Δ and the total work $(1+\varepsilon)\Delta$, where $\varepsilon = O(1/\log(\Delta))$. For practical parameters, this ε is in the order of 0.05, and this small part of the computation is easily parallelisable, so that the total evaluation time with s cores is around $(1 + 1/(20s))\Delta$. This gap should be of no importance since anyways, computational models do not capture well small constant factors with respect to real-world running time. Precise timing is unlikely to be achievable without resorting to trusted hardware; thus, applications of VDFs are designed not to be too sensitive to these small factors.

If despite these facts it is still problematic in some applications to know the output of the VDF slightly before having the proof, it is possible to eliminate this gap by artificially considering the proof as part of the output (the output is now a pair of group elements, and the proof is empty). The resulting protocol is still Δ-sequential (trivially), and as noted in Remark 5, it is also sound (note that this is specific to our construction: generically, adding the proof to the output does not preserve soundness). We also propose a second method in Sect. 8.1 which allows to exponentially reduce the overhead of the proof computation at the cost of lengthening the resulting proof by a few group elements.

Trapdoor verifiable delay function The construction proposed is actually a *trapdoor* VDF, from which we can derive an actual VDF. A party, Alice, holds a secret key sk (the trapdoor), and an associated public key pk. Given a piece of data x, a trapdoor VDF allows to compute an output y from x such that anyone can easily verify that either y has been computed by Alice (i.e. she used her secret trapdoor), or the computation of y required an amount of time at least Δ (where, again, time is measured as an amount of sequential work). The verification that y is the correct output of the VDF for input x should be efficient, with a cost polylog(Δ).

Deriving a verifiable delay function Suppose that a public key pk for a trapdoor VDF is given without any known associated secret key. This results in a simple VDF, where the evaluation requires a prescribed amount of time Δ for everyone (because there is no known trapdoor).

Now, how to publicly generate a public key without any known associated private key? In the construction we propose, this amounts to the public generation of a group of unknown order. A standard choice for such groups is RSA groups, but it is hard to generate an RSA number (a product of two large primes) with a strong guarantee that nobody knows the factorisation. It is possible to generate a random number large enough that with high probability it is divisible by two large primes (as done in [32]), but this approach severely damages the efficiency of the construction, and leaves more room for parallel optimisation of the arithmetic modulo a large integer, or for specialised hardware acceleration. It is also possible to generate a modulus by a secure multiparty execution of the RSA key generation procedure among independent parties, each contributing some secret random seeds (as done in [7]). However, in this scenario, a third party would have to assume that the parties involved in this computation did not collude to retrieve the secret. Instead, we propose to use the class group of an imaginary quadratic order. One can easily generate an imaginary quadratic order by choosing a random discriminant, and when the discriminant is large enough, the order of the class group cannot be computed.

These class groups were introduced in cryptography by Buchmann and Williams [10], exploiting the difficulty of computing their orders (and the fact that this order problem is closely related to the discrete logarithm and the root problems in this group). To this day, the best known algorithms for computing the order of the class group of an imaginary quadratic field of discriminant d are still of complexity $L_{|d|}(1/2)$ under the generalised Riemann hypothesis, for the usual function $L_t(s) = \exp\left(O\left(\log(t)^s \log\log(t)^{1-s}\right)\right)$, as shown in [22,34].

Circumventing classic impossibility results Finally, we further motivate the notion of *trapdoor* VDF by showing that it constitutes an original tool to circumvent classic impossibility results. We illustrate this in Sect. 9 with a simple and efficient identification protocol with surprising zero-knowledge and deniability properties.

1.2. *Time-Sensitive Cryptography and Related Work*

Rivest et al. [31] introduced in 1996 the use of *time-locks* for encrypting data that can be decrypted only in a predetermined time in the future. This was the first time-sensitive cryptographic primitive taking into account the parallel power of possible attackers. Other timed primitives appeared in different contexts: Bellare and Goldwasser [1,2] suggested *time capsules* for key escrowing in order to counter the problem of early recovery. Boneh and Naor [8] introduced *timed commitments*: a hiding and binding commitment scheme, which can be *forced open* by a procedure of determined running time. More recently, and as already mentioned, the notion of slow-timed hash function was introduced in [25] as a tool to provide trust to the generation of public random numbers.

Verifiable delay functions These slow-timed hash functions were recently revisited and formalised by Boneh et al. [4] under the name of verifiable delay functions. The function proposed in [25], *sloth*, is not asymptotically efficiently verifiable: the verification procedure (given x and y, verify that $\mathsf{sloth}(x) = y$) is faster than the evaluation procedure (given x, compute the value $\mathsf{sloth}(x)$) only by a constant factor. The authors of [4] proposed practical constructions that achieve an exponential gap between evaluation and verification, but do not strictly achieve the requirements of a VDF. For one of them, the evaluation requires an amount $\mathrm{polylog}(\Delta)$ of parallelism to run in parallel time Δ. The other one is insecure against an adversary that can run a large (but feasible) precomputation, so the setup must be regularly updated. The new construction we propose does not suffer these disadvantages.

Succinct arguments The core of the new construction is a method to produce constant sized and efficiently verifiable proofs that a triple (g, h, t) satisfies $h = g^{2^t}$, where g and h are two elements from a group of unknown order. The verification is significantly more efficient than direct computation, and proofs for a false statement are practically impossible to find: these proofs are *succinct arguments*. The first feasibility results for such constructions appear in seminal works of Kilian [24] and Micali [27].

Pietrzak's verifiable delay function Independently from the present work, another efficient VDF was proposed in [29]. The author describes an elegant construction, provably secure under the classic repeated squaring assumption of [31] when implemented over an RSA group $(\mathbf{Z}/N\mathbf{Z})^{\times}$ where N is a product of two safe primes. The philosophy of [29] is close to our construction: it consists in solving the puzzle of [31] (for a timing parameter Δ), and computing a proof of correctness. Their proofs can be computed with $O(\sqrt{\Delta}\log(\Delta))$ group multiplications. However, the proofs obtained are much longer (they consist of $O(\log(\Delta))$ group elements, versus a single group element in our construction), and the verification procedure is less efficient (it requires $O(\log(\Delta))$ group exponentiations, versus essentially two group exponentiations in our construction—for exponents of bit-length the security level k in both cases).

In the example given in [31], the group G is an RSA group for a 2048 bit modulus, and the time Δ is set to 2^{40} sequential squarings in the group, so the proofs are 10 KB long. In comparison, in the same setting, our proofs are 0.25 KB long.

Subsequent work Since the first version of this article, our new method to prove correct exponentiation in a group of unknown order has found applications beyond the construction of VDFs. Observing that the method allows to prove statements on the form $h = g^e$ for arbitrary integer exponents e (it is a *proof of exponentiation*, or *PoE*), Boneh et al. [6] expand on this technique to construct a *proof of knowledge of an integer exponent* (or *PoKE*). From these, they obtain cryptographic accumulators and vector commitment schemes that significantly improve over previous constructions.

Extending further in this direction, and still employing the new technique for PoE in a group of unknown order, Bünz et al. [11] construct a new polynomial commitment scheme for univariate and multivariate polynomials over finite fields, with no trusted setup. As an application, they build the first universal SNARKs (*succinct non-interactive arguments of knowledge*) with practical proof sizes and verification times that do not require a trusted setup.

1.3. *Notation*

Throughout this paper, the integer k denotes a security level (typically 128,192, or 256), and the map $H : \{0,1\}^* \rightarrow \{0,1\}^{2k}$ denotes a secure cryptographic hash function. For simplicity of exposition, the function H is regarded as a map from $\mathcal{A}^*$ to $\{0,1\}^{2k}$, where $\mathcal{A}^*$ is the set of strings over some alphabet $\mathcal{A}$ such that $\{0,1\} \subset \mathcal{A}$. The alphabet $\mathcal{A}$ contains at least all nine digits and twenty-six letters, and a special character $\star$. Given two strings $s_1, s_2 \in \mathcal{A}^*$, denote by $s_1||s_2$ their concatenation, and by $s_1|||s_2$ their concatenation separated by $\star$. The function $\mathtt{int} : \{0,1\}^* \rightarrow \mathbf{Z}_{\geq 0}$ maps $x \in \{0,1\}^*$ in the canonical manner to the non-negative integer with binary representation x. The function $\mathtt{bin} : \mathbf{Z}_{\geq 0} \rightarrow \{0,1\}^*$ maps any nonzero integer to its binary representation with no leading $\mathtt{0}$-characters, and $\mathtt{bin}(0) = \mathtt{0}$.

2. Trapdoor Verifiable Delay Functions

Let $\Delta : \mathbf{Z}_{>0} \rightarrow \mathbf{R}_{>0}$ be a function of the (implicit) security parameter k. This Δ is meant to represent a time duration, and what is precisely meant by *time* is explained in Sect. 3 (essentially, it measures an amount of sequential work). A party, Alice, has a public key pk and a secret key sk. Let x be a piece of data. Alice, thanks to her secret key sk, is able to quickly evaluate a function $\mathsf{trapdoor}_{\mathsf{sk}}$ on x. On the other hand, other parties knowing only pk can compute $\mathsf{eval}_{\mathsf{pk}}(x)$ in time Δ, but not faster (and importantly parallel computing power does not give a substantial advantage in going faster, remember that Δ measures the sequential work), such that the resulting value $\mathsf{eval}_{\mathsf{pk}}(x)$ is the same as $\mathsf{trapdoor}_{\mathsf{sk}}(x)$.

More formally, a trapdoor VDF consists of the following components (very close to the classic VDF defined in [4]):

$\mathsf{keygen} \rightarrow (\mathsf{pk}, \mathsf{sk})$ is a key generation procedure, which outputs Alice's public key pk and secret key sk. As usual, the public key should be publicly available, and the secret key is meant to be kept secret.

$\mathsf{trapdoor}_{\mathsf{sk}}(x, \Delta) \rightarrow (y, \pi)$ takes as input the data $x \in \mathcal{X}$ (for some input space $\mathcal{X}$), and uses the secret key sk to produce the output y from x, and a (possibly empty) proof π. The parameter Δ is the amount of sequential work required to compute the same output y without knowledge of the secret key.

$\mathsf{eval}_{\mathsf{pk}}(x, \Delta) \rightarrow (y, \pi)$ is a procedure to evaluate the function on x using only the public key pk, for a targeted amount of sequential work Δ. It produces the output y from x and a (possibly empty) proof π. This procedure is meant to be infeasible in time less than Δ (this will be expressed precisely in the security requirements).

$\mathsf{verify}_{\mathsf{pk}}(x, y, \pi, \Delta) \rightarrow \mathsf{true}$ or false is a procedure to check if y is indeed the correct output for x, associated with the public key pk and the evaluation time Δ, possibly with the help of the proof π.

Note that the security parameter k is implicitly an input to each of these procedures. Given any key pair $(\mathsf{pk}, \mathsf{sk})$ generated by the keygen procedure, the functionality of the scheme is the following. Given any input x and time parameter Δ, let $(y, \pi) \leftarrow \mathsf{eval}_{\mathsf{pk}}(x, \Delta)$ and $(y', \pi') \leftarrow \mathsf{trapdoor}_{\mathsf{sk}}(x, \Delta)$. Then, $y = y'$ and the procedures $\mathsf{verify}_{\mathsf{pk}}(x, y, \pi, \Delta)$ and $\mathsf{verify}_{\mathsf{pk}}(x, y', \pi', \Delta)$ both output true.

We also require the protocol to be *sound*, as in [4]. Intuitively, we want that if y' is not the correct output of $\mathsf{eval}_{\mathsf{pk}}(x, \Delta)$ then $\mathsf{verify}_{\mathsf{pk}}(x, y', \Delta)$ outputs false. We however allow the holder of the trapdoor to generate misleading values y'.

Definition 1 (*Soundness*) A trapdoor VDF is *sound* if any polynomially bounded algorithm solves the following *soundness-breaking* game with negligible probability (in k): given as input the public key pk, output a message x, a value y' and a proof π' such that $y' \neq \mathsf{eval}_{\mathsf{pk}}(x, \Delta)$, and $\mathsf{verify}_{\mathsf{pk}}(x, y', \pi', \Delta) = \mathsf{true}$.

The second security property is that the correct output cannot be produced in time less than Δ without knowledge of the secret key sk. This is formalised in the next section via the Δ-*evaluation race* game. A trapdoor VDF is Δ-*sequential* if any polynomially bounded adversary wins the Δ-evaluation race game with negligible probability.

3. Wall-Clock Time and Computational Assumptions

Primitives such as verifiable delay functions or time-lock puzzles wish to deal with the delicate notion of real-world time. This section discusses how to formally handle this concept, and how it translates in practice.

3.1. *Theoretical Model*

A precise notion of wall-clock time is difficult to capture formally. However, we can get a first approximation by choosing a model of computation and defining *time* as an amount of sequential work in this model. A model of computation is a set of allowable operations, together with their respective costs. For instance, working with circuits with gates $\vee$, $\wedge$ and $\neg$ which each have cost 1, the notion of time complexity of a circuit $\mathcal{C}$ can be captured by its depth $d(\mathcal{C})$, i.e. the length of the longest path in $\mathcal{C}$. The time complexity of a Boolean function f is then the minimal depth of a circuit implementing f, but this does not reflect the time it might take to actually compute f in the real world where one is not bound to using circuits. A random access machine might perform better, or maybe a quantum circuit.

A good model of computation for analysing the actual time it takes to solve a problem should contain all the operations that one could use in practice (in particular the adversary). From now on, we suppose the adversary works in a model of computation $\mathcal{M}$. We do not define exactly $\mathcal{M}$, but only assume that it allows all operations a potential adversary could perform, and that it comes with a cost function c and a time-cost function t. For any algorithm $\mathcal{A}$ and input x, the cost $C(\mathcal{A}, x)$ measures the overall cost of computing $\mathcal{A}(x)$ (i.e. the sum of the costs of all the elementary operations that are executed), while the time-cost $T(\mathcal{A}, x)$ abstracts the notion of time it takes to run $\mathcal{A}(x)$ in the model $\mathcal{M}$. For the model of circuits, one could define the cost as the size of the circuit and the time-cost as its depth. For concreteness, one can think of the model $\mathcal{M}$ as the model of parallel random-access machines.

All forthcoming security claims are (implicitly) made with respect to the model $\mathcal{M}$. The repeated squaring assumption of Rivest, Shamir and Wagner [31] can be expressed as Assumption 1.

Definition 2 ((δ, t)-*squaring game*) Let $k \in \mathbf{Z}_{>0}$ be a difficulty parameter, and $\mathcal{A}$ be an algorithm playing the game. The parameter t is a positive integer, and $\delta : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ is a function. The (δ, t)-*squaring game* goes as follows:

1. An RSA modulus N is generated at random by an RSA key-generation procedure, for the security parameter k;
2. $\mathcal{A}(N)$ outputs an algorithm $\mathcal{B}$;
3. An element $g \in \mathbf{Z}/N\mathbf{Z}$ is generated uniformly at random;
4. $\mathcal{B}(g)$ outputs $h \in \mathbf{Z}/N\mathbf{Z}$.

Then, $\mathcal{A}$ wins the game if $h = g^{2^t} \mod N$ and $T(\mathcal{B}, g) < t\delta(k)$.

Assumption 1 (*Repeated squaring assumption*) There is a cost function $\delta : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ and a small constant $\varepsilon > 0$ such that the following two statements hold:

1. There is an algorithm $\mathcal{S}$ such that for any modulus N generated by an RSA key-generation procedure with security parameter k, and any element $g \in \mathbf{Z}/N\mathbf{Z}$, the output of $\mathcal{S}(N, g)$ is the square of g, and $T(\mathcal{S}, (N, g)) < (1 + \varepsilon)\delta(k)$;

2. For any $t \in \mathbf{Z}_{>0}$, no algorithm $\mathcal{A}$ of polynomial cost[2] wins the (δ, t)-squaring game with non-negligible probability (with respect to the difficulty parameter k).

The function δ encodes the time-cost of computing a single modular squaring, and Assumption 1 expresses that without knowledge of the factorisation of N, there is no faster way to compute $g^{2^t} \mod N$ than performing t sequential squarings.

With this formalism, we can finally express the security notion of a trapdoor VDF.

Definition 3 (*Δ-evaluation race game*) Let $\mathcal{A}$ be a party playing the game. The parameter $\Delta : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ is a function of the (implicit) security parameter k. The *Δ-evaluation race game* goes as follows:

1. The random procedure keygen is run and it outputs a public key pk;
2. $\mathcal{A}(\mathsf{pk})$ outputs an algorithm $\mathcal{B}$;
3. Some data $x \in \mathcal{X}$ is generated according to some random distribution of min-entropy at least k;
4. $\mathcal{B}^{\mathcal{O}}(x)$ outputs a value y, where $\mathcal{O}$ is an oracle that outputs the evaluation $\mathsf{trapdoor}_{\mathsf{sk}}(x', \Delta)$ on any input $x' \neq x$.

Then, $\mathcal{A}$ wins the game if $T(\mathcal{B}, x) < \Delta$ and $\mathsf{eval}_{\mathsf{pk}}(x, \Delta)$ outputs y.

Definition 4 (*Δ-sequential*) A trapdoor VDF is *Δ-sequential* if any polynomially bounded player (with respect to the implicit security parameter) wins the above Δ-evaluation race game with negligible probability.

Observe that it is useless to allow $\mathcal{A}$ to adaptively ask for oracle evaluations of the VDF during the execution of $\mathcal{A}(\mathsf{pk})$: for any data x', the procedure $\mathsf{eval}_{\mathsf{pk}}(x', \Delta)$ produces the same output as $\mathsf{trapdoor}_{\mathsf{sk}}(x', \Delta)$, so any such request can be computed by the adversary in time $O(\Delta)$.

Remark 1 Suppose that the input x is hashed as $H(x)$ (by a secure cryptographic hash function) before being evaluated (as is the case in the construction we present in the next section), i.e.

$$\mathsf{trapdoor}_{\mathsf{sk}}(x, \Delta) = t_{\mathsf{sk}}(H(x), \Delta),$$

for some procedure t, and similarly for eval and verify. Then, it becomes unnecessary to give to $\mathcal{B}$ access to the oracle $\mathcal{O}$. We give a proof in Appendix A when H is modelled as a random oracle.

We choose to give $\mathcal{B}$ oracle access to $\mathcal{O}$ (thereby differing from the sequentiality defined in [4]) because it provides stronger security, ruling out undesirable properties like the VDF being a group homomorphism (see Remark 3).

Remark 2 At the third step of the game, the bound on the min-entropy is fixed to k. The exact value of this bound is arbitrary, but forbidding low entropy is important: if x has a

[2] i.e. $C(\mathcal{A}, g) = O(f(\mathrm{len}(g)))$ for a polynomial f, with $\mathrm{len}(g)$ the binary length of g.

good chance of falling in a small subset of $\mathcal{X}$, the adversary can simply precompute the VDF for all the elements of this subset.

3.2. *Timing Assumptions in the Real World*

Given an algorithm, or even an implementation of this algorithm, its actual running time will depend on the hardware on which it is run. If the algorithm is executed independently on several single-core general purpose CPUs, the variations in running time between them will be reasonably small as overclocking records on clock-speeds barely achieve 9 GHz (cf. [14]), only a small factor higher than a common personal computer. Assuming the computation is not parallelisable, using multiple CPUs would not allow to go faster. However, specialised hardware could be built to perform a certain computation much more efficiently than on any general purpose hardware.

For these reasons, the theoretical model developed in Sect. 3.1 has a limited accuracy. To resolve this issue, and evaluate precisely the security of a timing assumption like Assumption 1, one must estimate the speed at which the current state of technology allows to perform a certain task, given a possibly astronomical budget. To this end, the Ethereum Foundation and Protocol Labs [19] are currently investigating extremely fast hardware implementations of RSA multiplication, and hope to construct a piece of hardware close enough to today's technological limits, with the goal of using the present construction in their future platforms. Similarly, the Chia Network has opened a competition for very fast multiplication in the class group of a quadratic imaginary field, a choice of group with several advantages over RSA groups, as discussed in Example 2.

4. Construction of the Verifiable Delay Function

Let $x \in \mathcal{A}^*$ be the input at which the VDF is to be evaluated. Alice's secret key sk is the order of a finite group G, and her public key is a description of G allowing to compute the group multiplication efficiently. We also assume that any element g of G can efficiently be represented in a canonical way as binary strings $\texttt{bin}(g)$. Also part of Alice's public key is a hash function $H_G : \mathcal{A}^* \to G$.

Example 1 (RSA setup) A natural choice of setup is the following: the group G is $(\mathbf{Z}/N\mathbf{Z})^\times$ where $N = pq$ for a pair of distinct prime numbers p and q, where the secret key is $(p-1)(q-1)$ and the public key is N, and the hash function $H_G(x) = \texttt{int}(H(\text{“}\texttt{residue}\text{"}||x)) \mod N$ (where H is a secure cryptographic hash function). For a technical reason explained later in Remark 4, we actually need to work in $(\mathbf{Z}/N\mathbf{Z})^\times/\{\pm 1\}$, and we call this the *RSA setup*.

Example 2 (Class group setup) For a public setup where we do not want the private key to be known by anyone, one could choose G to be the class group of an imaginary quadratic field. This construction is presented in more detail in Sect. 4.3.

Consider a targeted evaluation time given by $\Delta = t\delta$ for a timing parameter t, where δ is the time-cost (i.e. the amount of sequential work) of computing a single squaring in the group G (as done in Assumption 1 for the RSA setup).

To evaluate the VDF on input x, first let $g = H_G(x)$. The basic idea (which finds its origins in [31]) is that for any $t \in \mathbf{Z}_{>0}$, Alice can efficiently compute g^{2^t} with two exponentiations, by first computing $e = 2^t \mod |G|$, followed by g^e. The running time is logarithmic in t. Any other party who does not know $|G|$ can also compute g^{2^t} by performing t sequential squarings, with a running time $t\delta$. Therefore anyone can compute $y = g^{2^t}$ but only Alice can do it fast, and any other party has to spend a time linear in t. However, verifying that the published value y is indeed g^{2^t} is long: there is no shortcut to the obvious strategy consisting in recomputing g^{2^t} and checking if it matches. To solve this issue, we propose the following public-coin succinct argument, for proving that $y = g^{2^t}$. The input of the interaction is (G, g, y, t). Let Primes$(2k)$ denote the set containing the 2^{2k} first prime numbers.

1. The verifier samples a prime ℓ uniformly at random from Primes$(2k)$.
2. The prover computes $\pi = g^{\lfloor 2^t/\ell \rfloor}$ and sends it to the verifier.
3. The verifier computes $r = 2^t \mod \ell$, (the least positive residue of 2^t modulo ℓ), and accepts if $g, y, \pi \in G$ and $\pi^\ell g^r = y$.

Now, it might not be clear how Alice or a third party should compute $\pi = g^{\lfloor 2^t/\ell \rfloor}$. For Alice, it is simple: she can compute $r = 2^t \mod \ell$. Then we have $\lfloor 2^t/\ell \rfloor = (2^t - r)/\ell$, and since she knows the order of the group, she can compute $q = (2^t - r)/\ell \mod |G|$ and $\pi = g^q$. We explain in Sect. 4.1 how anyone else can compute π without knowing $|G|$, with a total of $O(t/\log(t))$ group multiplications.

This protocol is made non-interactive using the Fiat–Shamir transformation, by letting $\ell = H_{\texttt{prime}}(\texttt{bin}(g)|||\texttt{bin}(y))$, where $H_{\texttt{prime}}$ is a hash function which sends any string s to an element of Primes$(2k)$. We assume in the security analysis below that this function is a uniformly distributed random oracle. The procedures trapdoor, verify, and eval are fully described in Algorithms 1, 2, and 3, respectively.

Remark 3 Instead of hashing the input x into the group G as $g = H_G(x)$, one could be tempted to consider $x \in G$. However, the function $x \mapsto x^{2^t}$ being a group homomorphism, bypassing the hashing step has undesirable consequences. Note that being a group homomorphism renders the construction insecure under Definition 3, where $\mathcal{B}$ is given oracle access to the trapdoor: given the outputs corresponding to xy and y^{-1} for any $y \in G \backslash \{1_G, x^{-1}\}$, an adversary can recover the output for x.

Verification It is straightforward to check that the verification condition $\pi^\ell g^r = y$ holds if the evaluator is honest. Now, what can a dishonest evaluator do? That question is answered formally in Sect. 6, but the intuitive idea is easy to understand. We will show that given x, finding a pair (y, π) different from the honest one amounts to solve a root-finding problem in the underlying group G (supposedly hard for anyone who does not know the secret order of the group). As a result, only Alice can produce misleading proofs.

Consider the above interactive succinct argument, and suppose that the verifier accepts, i.e. $\pi^\ell g^r = y$, where r is the least residue of 2^t modulo ℓ. Since $r = 2^t - \ell\lfloor 2^t/\ell \rfloor$, the verification condition is equivalent to

$$yg^{-2^t} = \left(\pi g^{-\lfloor 2^t/\ell \rfloor}\right)^\ell.$$

```
Data: a public key $\mathsf{pk} = (G, H_G)$ and a secret key $\mathsf{sk} = |G|$, some input $x \in \mathcal{A}^*$, a targeted evaluation
      time $\Delta = t\delta$.
Result: the output $y$, and the proof $\pi$.
$g \leftarrow H_G(x) \in G$;
$e \leftarrow 2^t \mod |G|$;
$y \leftarrow g^e$;
$\ell \leftarrow H_{\texttt{prime}}(\texttt{bin}(g)|||\texttt{bin}(y))$;
$r \leftarrow$ least residue of $2^t$ modulo $\ell$;
$q \leftarrow (2^t - r)\ell^{-1} \mod |G|$;
$\pi \leftarrow g^q$;
return $(y, \pi)$;
```

Algorithm 1: $\mathsf{trapdoor}_{\mathsf{sk}}(x, t) \rightarrow (y, \pi)$

```
Data: a public key $\mathsf{pk} = (G, H_G)$, some input $x \in \mathcal{A}^*$, a targeted evaluation time $\Delta = t\delta$, a VDF
      output $y$ and a proof $\pi$.
Result: true if $y$ is the correct evaluation of the VDF at $x$, false otherwise.
$g \leftarrow H_G(x)$;
$\ell \leftarrow H_{\texttt{prime}}(\texttt{bin}(g)|||\texttt{bin}(y))$;
$r \leftarrow$ least residue of $2^t$ modulo $\ell$;
if $\pi^\ell g^r = y$ then
    return true;
else
    return false;
end
```

Algorithm 2: $\mathsf{verify}_{\mathsf{pk}}(x, y, \pi, t) \rightarrow$ true or false

```
Data: a public key $\mathsf{pk} = (G, H_G)$, some input $x \in \mathcal{A}^*$, a targeted evaluation time $\Delta = t\delta$.
Result: the output value $y$ and a proof $\pi$.
$g \leftarrow H_G(x) \in G$;
$y \leftarrow g^{2^t}$;                               // via $t$ sequential squarings
$\ell \leftarrow H_{\texttt{prime}}(\texttt{bin}(g)|||\texttt{bin}(y))$;
$\pi \leftarrow g^{\lfloor 2^t/\ell \rfloor}$;      // following simple Algorithm 4, or faster Algorithm 5
return $(y, \pi)$;
```

Algorithm 3: $\mathsf{eval}_{\mathsf{pk}}(x, t) \rightarrow (y, \pi)$

Before the generation of ℓ, the left-hand side $\alpha = yg^{-2^t}$ is already determined. Once ℓ is revealed, the evaluator is able to compute $\beta = \pi g^{-\lfloor 2^t/\ell \rfloor}$, which is an ℓ-th root of α. For a prover to succeed with good probability, he must be able to extract ℓ-th roots of α for random values of ℓ. This is hard in our groups of interest, unless $\alpha = \beta = 1_G$, in which case (y, π) is the honest output.

Remark 4 Observe that in the RSA setup, this task is easy if $\alpha = \pm 1$, i.e. $y = \pm g^{2^t}$. It is however a difficult problem, given an RSA modulus N, to find an element $\alpha \mod N$ other than ± 1 from which ℓ-th roots can be extracted for any ℓ. This explains why we need to work in the group $G = (\mathbf{Z}/N\mathbf{Z})^\times/\{\pm 1\}$ instead of $(\mathbf{Z}/N\mathbf{Z})^\times$ in the RSA setup. This problem is formalised (and generalised to other groups) in Definition 7.

```
Data: an element $g$ in a group $G$ (with identity $1_G$), a prime number $\ell$ and a positive integer $t$.
Result: $g^{\lfloor 2^t/\ell \rfloor}$.
$x \leftarrow 1_G \in G$;
$r \leftarrow 1 \in \mathbf{Z}$;
for $i \leftarrow 0$ to $T - 1$ do
    $b \leftarrow \lfloor 2r/\ell \rfloor \in \{0, 1\}$;
    $r \leftarrow$ least residue of $2r$ modulo $\ell$;
    $x \leftarrow x^2 g^b$;
end
return $x$;
```

Algorithm 4: Simple algorithm to compute $g^{\lfloor 2^t/\ell \rfloor}$, with an on-the-fly long division [5].

4.1. *Computing the Proof π in $O(t/\log(t))$ Group Operations*

In this section, we describe how to compute the proof $\pi = g^{\lfloor 2^t/\ell \rfloor}$ with a total of $O(t/\log(t))$ group multiplications. First, we mention a very simple algorithm to compute π, which simply computes the long division $\lfloor 2^t/\ell \rfloor$ on the fly, as pointed out by Boneh et al. [5], but requires between t and $2t$ group operations. It is given in Algorithm 4.

We now describe how to perform the same computation with only $O(t/\log(t))$ group operations. Fix a parameter κ. The idea is to express $\lfloor 2^t/\ell \rfloor$ in base 2^κ as

$$\lfloor 2^t/\ell \rfloor = \sum_i b_i 2^{\kappa i} = \sum_{b=0}^{2^\kappa - 1} b \left(\sum_{i \text{ such that } b_i = b} 2^{\kappa i} \right).$$

Similarly to Algorithm 4, each coefficient b_i can be computed as

$$b_i = \left\lfloor \frac{2^\kappa (2^{t-\kappa(i+1)} \mod \ell)}{\ell} \right\rfloor,$$

where $2^{t-\kappa(i+1)} \mod \ell$ denotes the least residue of $2^{t-\kappa(i+1)}$ modulo ℓ. For each κ-bits integer $b \in \{0, \ldots, 2^\kappa - 1\}$, let $I_b = \{i \mid b_i = b\}$. We get

$$g^{\lfloor 2^t/\ell \rfloor} = \prod_{b=0}^{2^\kappa - 1} \left(\prod_{i \in I_b} g^{2^{\kappa i}} \right)^b. \tag{1}$$

Suppose first that all the values $g^{2^{\kappa i}}$ have been memorised (from the sequential computation of the value $y = g^{2^t}$). Then, each product $\prod_{i \in I_b} g^{2^{\kappa i}}$ can be computed in $|I_b|$ group multiplications (for a total of $\sum_b |I_b| = t/\kappa$ multiplications), and full product (1) can be deduced with about $\kappa 2^\kappa$ additional group operations. In total, this strategy requires about $t/\kappa + \kappa 2^\kappa$ group operations. Choosing, for instance, $\kappa = \log(t)/2$, we get about $t \cdot 2/\log(t)$ group operations. Of course, this would require the storage of t/κ group elements.

We now show that the memory requirement can easily be reduced to, for instance, $O(\sqrt{t})$ group elements, for essentially the same speedup. Instead of memorising each

κ element of the sequence g^{2^i}, only memorise every $\kappa\gamma$ element (i.e. the elements $g^{2^0}, g^{2^{\kappa\gamma}}, g^{2^{2\kappa\gamma}}, \ldots$), for some parameter γ (we will show that $\gamma = O(\sqrt{t})$ is sufficient). For each integer j, let $I_{b,j} = \{i \in I_b \mid i \equiv j \mod \gamma\}$. Now,

$$g^{\lfloor 2^t/\ell \rfloor} = \prod_{b=0}^{2^\kappa-1}\left(\prod_{j=0}^{\gamma-1}\prod_{i\in I_{b,j}} g^{2^{\kappa i}}\right)^b = \prod_{j=0}^{\gamma-1}\left(\prod_{b=0}^{2^\kappa-1}\left(\prod_{i\in I_{b,j}} g^{2^{\kappa(i-j)}}\right)^b\right)^{2^{\kappa j}}.$$

In each factor of the final product, $i - j$ is divisible by γ, so $g^{2^{\kappa(i-j)}}$ is one of the memorised values. A straightforward approach allows to compute this product with a total amount of group operations about $t/\kappa + \gamma\kappa 2^\kappa$, yet one can still do better. Write $y_{b,j} = \prod_{i\in I_{b,j}} g^{2^{\kappa(i-j)}}$, and split κ into two halves, as $\kappa_1 = \lfloor \kappa/2 \rfloor$ and $\kappa_0 = \kappa - \kappa_1$. Now, observe that for each index j,

$$\prod_{b=0}^{2^\kappa-1} y_{b,j}^b = \prod_{b_1=0}^{2^{\kappa_1}-1}\left(\prod_{b_0=0}^{2^{\kappa_0}-1} y_{b_1 2^{\kappa_0}+b_0,j}\right)^{b_1 2^{\kappa_0}} \cdot \prod_{b_0=0}^{2^{\kappa_0}-1}\left(\prod_{b_1=0}^{2^{\kappa_1}-1} y_{b_1 2^{\kappa_0}+b_0,j}\right)^{b_0}$$

The right-hand side provides a way to compute the product with a total of about $2(2^\kappa + \kappa 2^{\kappa/2})$ (instead of $\kappa 2^\kappa$ as in the more obvious strategy). The full method is summarised in Algorithm 5 (on page 37) and requires about $t/\kappa + \gamma 2^{\kappa+1}$ group multiplications.

The algorithm requires the storage of about $t/(\kappa\gamma) + 2^\kappa$ group elements. Choosing, for instance, $\kappa = \log(t)/3$ and $\gamma = \sqrt{t}$, we get about $t \cdot 3/\log(t)$ group operations, with the storage of about $\sqrt{t}$ group elements. This algorithm can also be parallelised.

Finally, we note that following the first version of the present work, algorithms to compute π have been revisited and optimised, and the most efficient variant to date appears to be Algorithm 6, proposed in [33]. It requires $t/\kappa + \gamma 2^\kappa$ group operations, and the storage of $t/(\kappa\gamma)$ precomputed group elements.

4.2. *A Simple Bandwidth and Storage Improvement*

Typically, an *evaluator* would compute the output y and the proof π, and send the pair (y, π) to the *verifiers*. Each verifier would compute the Fiat–Shamir challenge

$$\ell \leftarrow H_{\texttt{prime}}(\texttt{bin}(g)|||\texttt{bin}(y)),$$

then check $y = \pi^\ell g^{2^t \mod \ell}$. Instead, it is possible for the evaluator to transmit (ℓ, π), which has almost half the size (typically, ℓ is in the order of hundreds of bits, while group elements are in the order of thousands of bits). The verifiers would recover

$$y \leftarrow \pi^\ell g^{2^t \mod \ell},$$

and then verify that $\ell = H_{\texttt{prime}}(\texttt{bin}(g)|||\texttt{bin}(y))$. The two strategies are equivalent, but the second divides almost by 2 the bandwidth and storage footprint.

```
Data: an element $g$ in a group $G$ (with identity $1_G$), a prime number $\ell$, a positive integer $t$, two
      parameters $\kappa, \gamma > 0$, and a table of precomputed values $C_i = g^{2^{i\kappa\gamma}}$, for $i = 0, \dots, \lceil t/(\kappa\gamma) \rceil$.
Result: $g^{\lfloor 2^t/\ell \rfloor}$.
// define a function get_block such that $\lfloor 2^t/\ell \rfloor = \sum_i \text{get\_block}(i) 2^{\kappa i}$
get_block $\leftarrow$ the function that on input $i$ returns $\lfloor 2^\kappa (2^{t-\kappa(i+1)} \mod \ell)/\ell \rfloor$;
// split $\kappa$ into two halves
$\kappa_1 \leftarrow \lfloor \kappa/2 \rfloor$;
$\kappa_0 \leftarrow \kappa - \kappa_1$;
$x \leftarrow 1_G \in G$;
for $j \leftarrow \gamma - 1$ to 0 (descending order) do
    $x \leftarrow x^{2^\kappa}$;
    for $b \leftarrow 0$ to $2^\kappa - 1$ do
        $y_b \leftarrow 1_G \in G$;
    end
    for $i \leftarrow 0$ to $\lceil t/(\kappa\gamma) \rceil - 1$ do
        $b \leftarrow \text{get\_block}(i\gamma + j)$; // this could easily be optimised by computing
        the blocks iteratively as in Algorithm 4 (but computing blocks
        of $\kappa$ bits and taking steps of $\kappa\gamma$ bits), instead of computing
        them one by one.
        $y_b \leftarrow y_b \cdot C_i$;
    end
    for $b_1 \leftarrow 0$ to $2^{\kappa_1} - 1$ do
        $z \leftarrow 1_G \in G$;
        for $b_0 \leftarrow 0$ to $2^{\kappa_0} - 1$ do
            $z \leftarrow z \cdot y_{b_1 2^{\kappa_0} + b_0}$;
        end
        $x \leftarrow x \cdot z^{b_1 2^{\kappa_0}}$;
    end
    for $b_0 \leftarrow 0$ to $2^{\kappa_0} - 1$ do
        $z \leftarrow 1_G \in G$;
        for $b_1 \leftarrow 0$ to $2^{\kappa_1} - 1$ do
            $z \leftarrow z \cdot y_{b_1 2^{\kappa_0} + b_0}$;
        end
        $x \leftarrow x \cdot z^{b_0}$;
    end
end
return $x$;
```

Algorithm 5: Faster algorithm to compute $g^{\lfloor 2^t/\ell \rfloor}$, given some precomputations.

4.3. *Class Groups of Imaginary Quadratic Fields*

Number theory provides a simple way to generate groups of which no one knows the order: class groups of imaginary quadratic fields. Choose a random, negative, square-free integer d, of large absolute value, and such that $d \equiv 1 \mod 4$. Then, let $G = \mathrm{Cl}(d)$ be the class group of the imaginary quadratic field $\mathbf{Q}(\sqrt{d})$. Just as we wish, there is no known algorithm to efficiently compute the order of this group: the best known algorithms, all variants of the algorithm of Hafner and McCurley [22], have complexity $L_{|d|}(1/2)$. The multiplication can be performed efficiently, and each class can be represented canonically by its reduced ideal. Note that the even part of $|\mathrm{Cl}(d)|$ can easily be computed if the factorisation of d is known. Therefore, one should choose d to be a negative prime, which

```
Data: an element $g$ in a group $G$ (with identity $1_G$), a prime number $\ell$, a positive integer $t$, two
      parameters $\kappa, \gamma > 0$, and a table of precomputed values $C_i = g^{2^{i\kappa\gamma}}$, for $i = 0, \ldots, \lceil t/(\kappa\gamma) \rceil$.
Result: $g^{\lfloor 2^t/\ell \rfloor}$.
get_block ← the function that on input $i$ returns $\lfloor 2^\kappa (2^{t-\kappa(i+1)} \mod \ell)/\ell \rfloor$;
$x \leftarrow 1_G \in G$;
for $j \leftarrow \gamma - 1$ to 0 (descending order) do
    for $b \in \{0, \ldots, 2^\kappa - 1\}$ do
        $A_b \leftarrow []$;                                  // the empty list
    end
    for $i \leftarrow \lfloor t/(\kappa\gamma) \rfloor$ to 0 do
        $b \leftarrow$ get_block$(i\gamma + j)$;
        $A_b \leftarrow \mathrm{append}(A_b, i)$;
    end
    $y \leftarrow x$;
    for $b \leftarrow 2^\kappa - 1$ to 1 do
        for $i \in A_b$ do
            $y \leftarrow y \cdot C_i$;
        end
        $x \leftarrow x \cdot y$;
    end
end
return $x$;
```

.]Method to compute the proof proposed in [33, Algorithm 4].

Algorithm 6: Argonreport

ensures that $|\mathrm{Cl}(d)|$ is odd [13, Proposition 3.11]. Class groups of imaginary quadratic orders were first used as a platform for building cryptographic schemes by Buchmann and Williams in 1988 [10]. See [9] for a review of the arithmetic in such groups, and a discussion on the choice of cryptographic parameters.

The theory of class groups of quadratic number fields can equivalently be presented in the language of binary quadratic form. The latter provides a more explicit perspective, well suited for computational tasks.

Definition 5 (*Binary quadratic form*) An *(integral) binary quadratic form* is a bivariate polynomial

$$f(x, y) = ax^2 + bxy + cy^2 \in \mathbb{Z}[x, y],$$

where $a, b, c \in \mathbb{Z}$ are not all zero. Its *discriminant* is the quantity $\Delta(f) = b^2 - 4ac$.

Following the first version of the present article, Long [26] published an elementary introduction to the arithmetic of binary quadratic forms, specifically tailored for readers interested in implementing a VDF in the class group setup. In a nutshell, there is an equivalence relation on binary quadratic forms, and the set of all (primitive positive definite) forms of fixed negative discriminant d up to equivalence form the class group $C(d)$, isomorphic to $\mathrm{Cl}(d)$. Each class can be represented uniquely and efficiently by its *reduced form* [26, Sect. 5.2], and the group operation (the *composition*) is efficiently computable [26, Sect. 6].

Finally, we point out that one can efficiently hash into the group. Exploiting the heuristic that the classes of ideals of small prime norm behave like uniformly distributed random classes, one could proceed as follows: from the hash input, use a cryptographic hash function to derive a random-looking odd prime number a (with sufficient entropy, so at least 2^{2k} bits to get collision resistance) such that d is a (nonzero) quadratic residue modulo a. Then, let b be the smallest odd integer such that $b^2 \equiv d \mod a$, and let $c = \frac{b^2-d}{4a}$. Return at random the class of one of the two binary quadratic forms $ax^2 \pm bxy + cy^2$. In the language of ideals, this procedure returns the class of one of the two prime ideals above the split prime number a. To get a rigorous guarantee that the resulting class is (almost) uniformly distributed, one should ensure a is large enough to apply some explicit form of Chebotarev's density theorem. We leave the rigorous analysis, as well as potential heuristic optimisations of the method, as open research directions.

5. Analysis of the Sequentiality

In this section, the proposed construction is proven to be $(t\delta)$-sequential, meaning that no polynomially bounded player can win the associated $(t\delta)$-evaluation race game with non-negligible probability (in other words, the VDF cannot be evaluated in time less than $t\delta$). For the RSA setup, it is proved under the classic repeated squaring assumption of Rivest, Shamir, and Wagner [31] (formalised in Assumption 1), and more generally, it is secure for groups where a generalisation of this assumption holds (Assumption 2).

5.1. *Generalised Repeated Squaring Assumptions*

The following game generalises the classic repeated squaring assumption to arbitrary families of groups of unknown orders.

Definition 6 (*Generalised* (δ, t)-*squaring game*) Consider a sequence of the form $(\mathcal{G}_k)_{k\in\mathbf{Z}_{>0}}$, where each $\mathcal{G}_k$ is a set of finite groups (supposedly of unknown orders), associated with a "difficulty parameter" k. Let keygen be a procedure to generate a random group from $\mathcal{G}_k$, according to the difficulty k.

Fix the difficulty parameter $k \in \mathbf{Z}_{>0}$, and let $\mathcal{A}$ be an algorithm playing the game. The parameter t is a positive integer, and $\delta : \mathbf{Z}_{>0} \rightarrow \mathbf{R}_{>0}$ is a function. The (δ, t)-*squaring game* goes as follows:

1. A group G is generated by keygen;
2. $\mathcal{A}(G)$ outputs an algorithm $\mathcal{B}$;
3. An element $g \in G$ is generated uniformly at random;
4. $\mathcal{B}(g)$ outputs $h \in G$.

Then, $\mathcal{A}$ wins the game if $h = g^{2^t}$ and $T(\mathcal{B}, g) < t\delta(k)$.

Assumption 2 (*Generalised repeated squaring assumption*) The generalised repeated squaring assumption for a given family of groups $(\mathcal{G}_k)_{k\in\mathbf{Z}_{>0}}$ is the following. There is a cost function $\delta : \mathbf{Z}_{>0} \rightarrow \mathbf{R}_{>0}$ and a small constant $\varepsilon > 0$ such that the following two statements hold for any k:

1. There is an algorithm $\mathcal{S}$ such that for any group $G \in \mathcal{G}_k$ (for the difficulty parameter k), and any element $g \in G$, the output of $\mathcal{S}(G, g)$ is the square of g, and $T(\mathcal{S}, (G, g)) \leq (1 + \varepsilon)\delta(k)$;
2. For any $t \in \mathbf{Z}_{>0}$, no algorithm $\mathcal{A}$ of polynomial cost wins the (δ, t)-squaring game with non-negligible probability (with respect to the difficulty parameter k).

The function δ encodes the time-cost of computing a single squaring in a group of $\mathcal{G}_k$, and Assumption 2 expresses that without more specific knowledge about these groups (such as their orders), there is no faster way to compute g^{2^t} than performing t sequential squarings.

5.2. *Sequentiality in the Random Oracle Model*

Theorem 1 ($t\delta$-sequentiality of the trapdoor VDF in the random oracle model) *Let $\mathcal{A}$ be a player winning with probability p_{win} the $(t\delta)$-evaluation race game associated with the proposed construction, assuming H_G and $H_{\mathtt{prime}}$ are random oracles and $\mathcal{A}$ is limited to q oracle queries.*[3] *Then, there is a player $\mathcal{C}$ for the (generalised) (δ, t)-squaring game, with winning probability $p \geq (1 - q/2^k)p_{\mathsf{win}}$, and with same running time as $\mathcal{A}$ (up to a constant factor*[4]*).*

Proof Build $\mathcal{C}$ as follows. Upon receiving the group G, $\mathcal{C}$ starts running $\mathcal{A}$ on input G. The random oracles H_G and $H_{\mathtt{prime}}$ are simulated in a straightforward manner, maintaining a table of values, and generating a random outcome for any new request (with distribution uniform in G and in Primes$(2k)$, respectively). When $\mathcal{A}(G)$ outputs an algorithm $\mathcal{B}$, $\mathcal{C}$ generates a random $x \in \mathcal{X}$ (according to the same distribution as the $(t\delta)$-evaluation race game). If x has been queried by the oracle already, $\mathcal{C}$ aborts; this happens with probability at most $q/2^k$, since the min-entropy of the distribution of messages in the $(t\delta)$-evaluation race game is at least k. Otherwise, $\mathcal{C}$ outputs the following algorithm $\mathcal{B}'$. When receiving as input the challenge g, $\mathcal{B}'$ adds g to the table of oracle H_G, for the input x (i.e. $H_G(x) = g$). As discussed in Remark 1, we can assume that the algorithm $\mathcal{B}$ does not call the oracle $\mathsf{trapdoor}_{\mathsf{sk}}(-, \Delta)$. Then $\mathcal{B}'$ can invoke $\mathcal{B}$ on input x while simulating the oracles H_G and $H_{\mathtt{prime}}$. Whenever $\mathcal{B}$ outputs y, $\mathcal{B}'$ outputs y, which equals g^{2^t} whenever y is the correct evaluation of the VDF at x. We assume that simulating the oracle has a negligible cost, so $\mathcal{B}'(g)$ has essentially the same time-cost as $\mathcal{B}(x)$. Then, $\mathcal{C}$ wins the (δ, t)-squaring game with probability $p \geq p_{\mathsf{win}}(1 - q/2^k)$. □

[3] In this game, the output of $\mathcal{A}$ is another algorithm $\mathcal{B}$. When we say that $\mathcal{A}$ is limited to q queries, we limit the total number of queries by $\mathcal{A}$ and $\mathcal{B}$ combined. In other words, if $\mathcal{A}$ did $x \leq q$ queries, then its output $\mathcal{B}$ is limited to $q - x$ queries.

[4] Note that this constant factor does not affect the chances of $\mathcal{C}$ to win the (δ, t)-squaring game, since it concerns only the running time of $\mathcal{C}$ itself and not of the algorithm output by $\mathcal{C}(G)$.

6. Analysis of the Soundness

In this section, the proposed construction is proven to be sound, meaning that no polynomially bounded player can produce a misleading proof for an invalid output of the VDF. For the RSA setup, it is proved under a new number theoretic assumption expressing that it is hard to find an integer $u \neq 0, \pm 1$ for which ℓ-th roots modulo an RSA modulus N can be extracted for ℓ-values sampled uniformly at random from $\mathrm{Primes}(2k)$, when the factorisation of N is unknown. More generally, the construction is sound if a generalisation of this assumption holds in the group of interest.

6.1. *The Root Finding Problem*

The following game formalises the root finding problem.

Definition 7 (*The root finding game* $\mathcal{G}^{\mathsf{root}}$) Let $\mathcal{A}$ be a party playing the game. The *root finding game* $\mathcal{G}^{\mathsf{root}}(\mathcal{A})$ goes as follows: first, the keygen procedure is run, resulting in a group G which is given to $\mathcal{A}$ (G is supposedly of unknown order). The player $\mathcal{A}$ then outputs an element u of G. An integer ℓ is sampled uniformly from $\mathrm{Primes}(2k)$ and given to $\mathcal{A}$. The player $\mathcal{A}$ outputs an element v of G and wins the game if $v^{\ell} = u \neq 1_G$.

This game has been proved to be difficult in the generic group model [6]. In the RSA setup, the group G is the quotient $(\mathbf{Z}/N\mathbf{Z})^{\times}/\{\pm 1\}$, where N is a product of two random large prime numbers. It is not known if this problem can easily be reduced to a standard assumption such as the difficulty of factoring N or the RSA problem, for which the best known algorithms have complexity $L_N(1/3)$.[5] The standard problem of extracting e-th roots in RSA groups is however strongly believed to be difficult. When $e = 2$, it is known to be equivalent to factoring N. For arbitrary e, the best known algorithms have complexity $L_N(1/3)$ even when given access to some helping oracle [23].

Similarly, in the class group setting, this problem is not known to reduce to a standard assumption, but it is closely related to the order problem and the root problem (which are tightly related to each other, see [3, Theorem 3]), for which the best known algorithms have complexity $L_{|d|}(1/2)$ where d is the discriminant of the imaginary quadratic field. *Necessity of the difficulty of the root finding problem.* The root finding problem must be hard for the protocol to be sound. Indeed, as demonstrated in [5, p. 7], winning the root finding game allows to produce proofs for wrong outputs of the VDF. Let $\mathcal{A}$ be an adversary for the root finding game. It outputs an element $u \neq 1_G$ of G. Now, given any $g \in G$, and $y = g^{2^t}$, one can convince the verifier that uy is the correct VDF output as follows. The verifier outputs a random prime ℓ. Using $\mathcal{A}$, one can find an element v such that $v^{\ell} = u$. Now, with $2^t = q\ell + r$, and $0 \leq r < \ell$, the element $\pi = vg^q$ is a valid proof for the output uh since $\pi^{\ell} g^r = v^{\ell} g^{q\ell + r} = uy$. Note that such an attack would break the soundness, but not the sequentiality: the proof does not guarantee that the claimed output is the unique correct output, but the attacker still needs to perform the sequential computation $y = g^{2^t}$.

[5] Recall the L-notation $L_t(s) = \exp\left(O\left(\log(t)^s \log\log(t)^{1-s}\right)\right)$.

6.2. *Soundness in the Random Oracle Model*

Before proving the soundness, we prove that to win the game $\mathcal{G}^{\mathsf{root}}$, it is sufficient to win the following game $\mathcal{G}_X^{\mathsf{root}}$, which is more convenient for our analysis.

Definition 8 (*The oracle root finding game* $\mathcal{G}_X^{\mathsf{root}}$) Let $\mathcal{A}$ be a party playing the game. Let X be a function that takes as input a group G and a string s in $\mathcal{A}^*$ and outputs an element $X(G, s) \in G$. Let $\mathcal{O} : \mathcal{A}^* \to \mathrm{Primes}(2k)$ be a random oracle with the uniform distribution. The player has access to the random oracle $\mathcal{O}$. The *oracle root finding game* $\mathcal{G}_X^{\mathsf{root}}(\mathcal{A}, \mathcal{O})$ goes as follows: first, the keygen procedure is run and the resulting group G is given to $\mathcal{A}$. The player $\mathcal{A}$ then outputs a string $s \in \mathcal{A}^*$, and an element v of G. The game is won if $v^{\mathcal{O}(s)} = X(G, s) \neq 1_G$.

Lemma 1 *If there is a function X and an algorithm $\mathcal{A}$ limited to q queries to the oracle $\mathcal{O}$ winning the game $\mathcal{G}_X^{\mathsf{root}}(\mathcal{A}, \mathcal{O})$ with probability p_{win}, there is an algorithm $\mathcal{B}$ winning the game $\mathcal{G}^{\mathsf{root}}(\mathcal{B})$ with probability at least $p_{\mathsf{win}}/(q + 1)$, and same running time, up to a small constant factor.*

Proof Let $\mathcal{A}$ be an algorithm limited to q oracle queries, and winning the game with probability p_{win}. Build an algorithm $\mathcal{A}'$ which does exactly the same thing as $\mathcal{A}$, but with possibly additional oracle queries at the end to make sure the output string s' is always queried to the oracle, and the algorithm always does exactly $q + 1$ (distinct) oracle queries.

Build an algorithm $\mathcal{B}$ playing the game $\mathcal{G}^{\mathsf{root}}$, using $\mathcal{A}'$ as follows. Upon receiving $\mathsf{pk} = G$, $\mathcal{B}$ starts running $\mathcal{A}'$ on input pk. The oracle $\mathcal{O}$ is simulated as follows. First, an integer $i \in \{1, 2, \dots, q + 1\}$ is chosen uniformly at random. For the first $i - 1$ (distinct) queries from $\mathcal{A}'$ to $\mathcal{O}$, the oracle value is chosen uniformly at random from $\mathrm{Primes}(2k)$. When the ith string $s \in \mathcal{A}^*$ is queried to the oracle, the algorithm $\mathcal{B}$ outputs $u = X(G, s)$, concluding the first round of the game $\mathcal{G}^{\mathsf{root}}$. The game continues as the integer ℓ is received (uniform in $\mathrm{Primes}(2k)$). This ℓ is then used as the value for the ith oracle query $\mathcal{O}(s)$, and the algorithm $\mathcal{A}'$ can continue running. The subsequent oracle queries are handled like the first $i - 1$ queries, by picking random primes in $\mathrm{Primes}(2k)$. Finally, $\mathcal{A}'$ outputs a string $s' \in \mathcal{A}^*$ and an element v of G. To conclude the game $\mathcal{G}^{\mathsf{root}}(\mathcal{B})$, $\mathcal{B}$ returns v.

Since $\mathcal{O}$ simulates a random oracle with uniform outputs in $\mathrm{Primes}(2k)$, $\mathcal{A}'$ outputs with probability p_{win} a pair (s', v) such that $v^{\mathcal{O}(s')} = X(G, s') \neq 1_G$; denote this event $\mathsf{win}_{\mathcal{A}'}$. If $s = s'$, this condition is exactly $v^{\ell} = u \neq 1_G$, where $u = X(G, s)$ is the output for the first round of $\mathcal{G}^{\mathsf{root}}$, and $\mathcal{O}(s) = \ell$ is the input for the second round. If these conditions are met, the game $\mathcal{G}^{\mathsf{root}}(\mathcal{B})$ is won. Therefore

$$\Pr[\mathcal{B} \text{ wins } \mathcal{G}^{\mathsf{root}}] \geq p_{\mathsf{win}} \cdot \Pr\left[s = s' | \mathsf{win}_{\mathcal{A}'}\right].$$

Let $\mathcal{Q} = \{s_1, s_2, \dots, s_{q+1}\}$ be the $q + 1$ (distinct) strings queried to $\mathcal{O}$ by $\mathcal{A}'$, indexed in chronological order. By construction, we have $s = s_i$. Let j be such that $s' = s_j$ (recall that $\mathcal{A}'$ makes sure that $s' \in \mathcal{Q}$). Then,

$$\Pr\left[s = s' | \mathsf{win}_{\mathcal{A}'}\right] = \Pr\left[i = j | \mathsf{win}_{\mathcal{A}'}\right]$$

The integer i is chosen uniformly at random in $\{1, 2, \ldots, q + 1\}$, and the values given to $\mathcal{A}'$ are independent from i (the oracle values are all independent random variables). So $\Pr\left[i = j | \mathsf{win}_{\mathcal{A}'}\right] = 1/(q + 1)$. Therefore $\Pr[\mathcal{B}$ wins $\mathcal{G}^{\mathsf{root}}] \geq p_{\mathsf{win}}/(q + 1)$. Since $\mathcal{B}$ mostly consists in running $\mathcal{A}$ and simulating the random oracle, it is clear than both have the same running time, up to a small constant factor. □

Theorem 2 (Soundness of the trapdoor VDF in the random oracle model) *Let $\mathcal{A}$ be a player winning with probability p_{win} the soundness-breaking game associated with the proposed scheme, assuming $H_{\texttt{prime}}$ is a random oracle and $\mathcal{A}$ is limited to q oracle queries.*[6] *Then, there is a player $\mathcal{D}$ for the root finding game $\mathcal{G}^{\mathsf{root}}$ with winning probability $p \geq p_{\mathsf{win}}/(q+1)$, and with same running time as $\mathcal{A}$ (up to a constant factor).*

Proof Instead of directly building $\mathcal{D}$, let $\mathcal{O}$ as in Definition 8, and we build an algorithm $\mathcal{D}'$ playing the game $\mathcal{G}_X^{\mathsf{root}}(\mathcal{D}', \mathcal{O})$, and invoke Lemma 1. Define the function X as follows. Recall that for any group G that we consider in the construction, each element $g \in G$ admits a canonical binary representation $\texttt{bin}(g)$. For any such group G, any elements $g, h \in G$, let

$$X(G, \texttt{bin}(g)|||\texttt{bin}(h)) = h/g^{2^t},$$

and let $X(G, s) = 1_G$ for any other string s. When receiving pk, $\mathcal{D}'$ starts running $\mathcal{A}$ with input pk. The oracle $H_{\texttt{prime}}$ queried by $\mathcal{A}$ is instantiated with the oracle $\mathcal{O}$. The algorithm $\mathcal{A}$ outputs a message x, and a pair $(y, \pi) \in G \times G$ (if it is not of this form, abort). Output $s = \texttt{bin}(H_G(x))|||\texttt{bin}(y)$ and $v = \pi/H_G(x)^{\lfloor 2^t/\mathcal{O}(s)\rfloor}$. If $\mathcal{A}$ won the simulated soundness-breaking game, the procedure did not abort, and $v^{\mathcal{O}(s)} = X(G, s) \neq 1_G$, so $\mathcal{D}'$ wins the game. If furthermore $\mathcal{O}(s)$ is coprime to $|G|$, we have $v^{\mathcal{O}(s)} \neq 1_G$ so $\mathcal{D}'$ wins the game. Since $\mathcal{O}(s)$ is one of at most q random prime numbers of bit-length $2k$, it is coprime to $|G|$ with probability $1 - \varepsilon$ with $\varepsilon = \operatorname{negl}\left(\frac{k}{\log\log(|G|)\log(q)}\right)$. Hence $\mathcal{D}'$ has winning probability p_{win}. Since $\mathcal{A}$ was limited to q oracle queries, $\mathcal{D}'$ also does not do more than q queries. Applying Lemma 1, there is an algorithm $\mathcal{D}$ winning the game $\mathcal{G}^{\mathsf{root}}(\mathcal{B})$ with probability $p \geq p_{\mathsf{win}}/(q + 1)$. □

Remark 5 The construction remains sound if instead of considering the output y and the proof π, we consider the output to be the pair (y, π), with an empty proof. The winning condition for $\mathcal{A}$ playing the soundness-breaking game changes: $\mathcal{A}$ is now allowed to break soundness even for the honest output y, by producing a proof π different from the honest one. The winning probability of $\mathcal{D}$ in Theorem 2 becomes $p \geq p_{\mathsf{win}}(1-\varepsilon)/(q+1)$, where $\varepsilon = \operatorname{negl}\left(\frac{k}{\log\log(|G|)\log(q)}\right)$, by accounting for the unlikely event that the large random prime $\mathcal{O}(s)$ is a divisor of $|G|$.

[6] Here again, the output of $\mathcal{A}$ is another algorithm $\mathcal{B}$. When we say that $\mathcal{A}$ is limited to q queries, we limit the total number of queries by $\mathcal{A}$ and $\mathcal{B}$ combined.

7. Aggregating and Watermarking Proofs

In this section, we present two useful properties of the VDF: the proofs can be aggregated and watermarked. The methods of this section follow from discussions at the August 2018 workshop at Stanford hosted by the Ethereum Foundation and the Stanford Center for Blockchain Research. The author wishes to thank the participants for their contribution.

7.1. *Aggregation*

If the VDF is evaluated at multiple inputs, it is possible to produce a single proof $\widetilde{\pi} \in G$ that simultaneously proves the validity of all the outputs. Suppose that n inputs are given, $x_1, \dots, x_n$. For each index i, let $g_i = H_G(x_i)$. The following public-coin interactive succinct argument allows to prove that a given list $(y_i)_{i=1}^n$ satisfies $y_i = g_i^{2^t}$:

1. The verifier samples a prime ℓ uniformly at random from $\text{Primes}(2k)$, and n uniformly random integers $(\alpha_i)_{i=1}^n$ of k bits.
2. The prover computes
$$\widetilde{\pi} = \left(\prod_{i=1}^n g_i^{\alpha_i}\right)^{\lfloor 2^t/\ell \rfloor}$$
and sends it to the verifier.
3. The verifier computes $r = 2^t \mod \ell$, (the least positive residue of 2^t modulo ℓ), and accepts if
$$\widetilde{\pi}^\ell \left(\prod_{i=1}^n g_i^{\alpha_i}\right)^r = \prod_{i=1}^n y_i^{\alpha_i}.$$

The single group element $\widetilde{\pi}$ serves as proof for the whole list of n statements $y_i = g_i^{2^t}$: it is an aggregated proof. The protocol can be made non-interactive by a Fiat–Shamir transformation: let

$$s = \texttt{bin}(g_1)|||\texttt{bin}(g_2)|||\dots|||\texttt{bin}(g_n)|||\texttt{bin}(y_1)|||\texttt{bin}(y_2)|||\dots|||\texttt{bin}(y_n),$$

and let $\ell = H_{\texttt{prime}}(s)$, and for each index i, let $\alpha_i = \texttt{int}(H(\texttt{bin}(i)|||s))$ (where H is a secure cryptographic hash function). For simplicity, we prove the soundness in the interactive setup (the non-interactive soundness then follows from the Fiat–Shamir heuristic).

Remark 6 One could harmlessly fix $\alpha_1 = 1$, leaving only α_i to be chosen at random for $i > 1$. We present the protocol as above for simplicity, to avoid dealing with $i = 1$ as a special case in the proof below.

Theorem 3 *If there is a malicious prover $\mathcal{P}$ breaking the soundness of the above interactive succinct argument with probability p, then there is a player $\mathcal{B}$ winning the* *root finding game $\mathcal{G}^{\mathsf{root}}$ with probability at least $(p^2-2^{1-k})/3$, with essentially the same running time as $\mathcal{P}$.*

Proof Let $\mathcal{I} = \{0, 1, \dots, 2^k - 1\}$, and let $\mathcal{Z} = \mathcal{I}^{n-1} \times \mathrm{Primes}(2k)$. Let $Z = (\alpha_2, \dots, \alpha_n, \ell)$ be a uniformly distributed random variable in $\mathcal{Z}$, and let α_1 and α_1' be two independent, uniformly distributed random variables in $\mathcal{I}$. Let win and win' be the events that $\mathcal{P}$ breaks soundness when given $(\alpha_1, \alpha_2, \dots, \alpha_n, \ell)$ and $(\alpha_1', \alpha_2, \dots, \alpha_n, \ell)$, respectively. We wish to estimate the probability of the event $\mathsf{double\_win} = \mathsf{win} \wedge \mathsf{win}' \wedge (\alpha_1 \not\equiv \alpha_1' \mod \ell)$. Observe that conditioning over $Z = z$ for an arbitrary, fixed $z \in \mathcal{Z}$, the events win and win' are independent and have same probability, so

$$\Pr[\mathsf{win} \wedge \mathsf{win}'] = \frac{1}{|Z|}\sum_{z\in\mathcal{Z}} \Pr[\mathsf{win} \wedge \mathsf{win}' \mid Z = z] = \frac{1}{|Z|}\sum_{z\in\mathcal{Z}} \Pr[\mathsf{win} \mid Z = z]^2.$$

From the Cauchy–Schwarz inequality, we get

$$\frac{1}{|Z|}\sum_{z\in\mathcal{Z}} \Pr[\mathsf{win} \mid Z = z]^2 \geq \left(\frac{1}{|Z|}\sum_{z\in\mathcal{Z}} \Pr[\mathsf{win} \mid Z = z]\right)^2 = \Pr[\mathsf{win}]^2 = p^2.$$

Therefore $\Pr[\mathsf{win} \wedge \mathsf{win}'] \geq p^2$. Furthermore,

$$\begin{aligned}\Pr[\alpha_1 \equiv \alpha_1' \mod \ell] &\leq \Pr[\alpha_1 \equiv \alpha_1']\Pr[\ell > 2^k] + \Pr[\ell < 2^k]\\ &\leq \Pr[\alpha_1 \equiv \alpha_1'] + \Pr[\ell \in \mathrm{Primes}(k)]\\ &\leq 2^{1-k}.\end{aligned}$$

We conclude that $\Pr[\mathsf{double\_win}] \geq p^2 - 2^{1-k}$.

With these probabilities at hand, we can now construct the player $\mathcal{B}$ for the root finding game $\mathcal{G}^{\mathsf{root}}$. Run $\mathcal{P}$, which outputs values g_i and y_i. If $y_i = g_i^{2^t}$ for all i, abort. Up to some reindexing, we can now assume $y_1 \neq g_1^{2^t}$. Draw $\alpha_1, \alpha_1', \alpha_2, \dots, \alpha_n$ uniformly at random from $\mathcal{I}$. Define

$$x_0 = y_1/g_1^{2^t},\ x_1 = \prod_{i=1}^{n}(y_i^{\alpha_i}/g_i^{2^t})^{\alpha_i},\ x_2 = (y_1/g_1^{2^t})^{\alpha_1'}\prod_{i=2}^{n}(y_i^{\alpha_i}/g_i^{2^t})^{\alpha_i}.$$

Let b be a uniformly random element of $\{0, 1, 2\}$. The algorithm $\mathcal{B}$ outputs x_b. We get back a challenge ℓ. Run the prover $\mathcal{P}$ twice, independently, for the challenges $(\alpha_1, \alpha_2, \dots, \alpha_n, \ell)$ and $(\alpha_1', \alpha_2, \dots, \alpha_n, \ell)$, and suppose that both responses break soundness, and $\alpha_1 \not\equiv \alpha_1' \mod \ell$ (i.e. the event $\mathsf{double\_win}$ occurs). If $x_1 \neq 1_G$ or $x_2 \neq 1_G$, the winning responses from $\mathcal{P}$ allow to extract an ℓ-th root of either x_1 or x_2, respectively. Otherwise, we have $x_1 = x_2$, which implies that $x_0^{\alpha_1-\alpha_1'} = 1_G$, so x_0 is an element of order dividing $\alpha_1 - \alpha_1'$, and one can easily extract any ℓ-th root of x_0. In conclusion, under the event $\mathsf{double\_win}$, one can always extract an ℓ-th root of either x_0, x_1 or x_2, so the total winning probability of algorithm $\mathcal{B}$ is at least $(p^2 - 2^{1-k})/3$. □

7.2. *Watermarking*

When using a VDF to build a decentralised randomness beacon (e.g. as a backbone for an energy-efficient blockchain design), people who spent time and energy evaluating the VDF should be rewarded for their effort. Since the output of the VDF is supposed to be unique, it is hard to reliably identify the person who computed it. A naive attempt of the evaluator to sign the output would not prevent theft: since the output is public, a dishonest party could as easily sign it and claim it their own.

Let the evaluator's identity be given as a string id. One proposed method (see [18]) essentially consists in computing the VDF twice: once on the actual input, and once on a combination of the input with the evaluator's identity id. Implemented carefully, this method could allow to reliably reward the evaluators for their work, but it also doubles the required effort. In the following, we sketch two cost-effective solutions to this problem.

The first cost-effective approach consists in having the evaluator prove that he knows some hard-to-recover intermediate value of the computation of the VDF. Since the evaluation of our proposed construction requires computing in sequence the elements $g_i = g^{2^i}$ for $i = 1, \dots, t$, and only the final value $y = g_t$ of the sequence is supposed to be revealed, one can prove that they performed the computation by proving that they know g_{t-1} (it is a square root of y; hence, the fastest way for someone else to recover it would be to recompute the full sequence). A simple way to do so would be for the evaluator to reveal the value $c_{\mathsf{id}} = g_{t-1}^{p_{\mathsf{id}}}$ (a *certificate*), where $p_{\mathsf{id}} = H_{\mathtt{prime}}(\mathsf{id})$. The validity of the certificate can be checked via the equation $y^{p_{\mathsf{id}}} = c_{\mathsf{id}}^2$. The security claim is the following: given the input x, the output y, the proof π, and the certificate c_{id}, the cost for an adversary with identifier id' (distinct from id) to produce a valid certificate $c_{\mathsf{id}'}$ is as large as actually recomputing the output of the VDF by themself.

The above method is cost-effective as it does not require the evaluator to perform much more work than evaluating the VDF. However, it makes the output longer by adding an extra group element: the certificate. Another approach consists in producing a single group element that plays simultaneously the role of the proof and the certificate. This element is a *watermarked proof*, tied to the evaluator's identity. This can be done easily with our construction. In evaluation procedure (Algorithm 3), replace the definition of the prime ℓ by $H_{\mathtt{prime}}(\mathsf{id}|||\mathtt{bin}(g)|||\mathtt{bin}(y))$ (and the corresponding change must be made in the verification procedure). The resulting proof π_{id} is now inextricably tied to id. Informally, the security claim is the following: given the input x, the output y, and the watermarked proof π_{id}, the cost for an adversary with identifier id' (distinct from id) to produce a valid proof $\pi_{\mathsf{id}'}$ is about as large as reevaluating the VDF altogether. Indeed, a honest prover, after having computed the output y, can compute π_{id} at a reduced cost thanks to some precomputed intermediate values. But an adversary does not have these intermediate values, so they would have to compute $\pi_{\mathsf{id}'}$ from scratch. This is an exponentiation in G, with exponent of bit-length close to t; without any intermediate values, it requires in the order of t sequential group operations, which is the cost of evaluating the VDF.

8. Trade-Offs Between Proof Shortness and Prover Efficiency

In this section, we propose and compare two trade-offs between proof shortness and prover efficiency. The first construction, called the *iterated prover*, iteratively applies the VDF proposed in the present paper. The second construction, called the *hybrid prover*, combines the VDF of this paper with the Pietrzak scheme [29], in an attempt to benefit from both the shortness of our proofs and the efficiency of the Pietrzak prover.

8.1. *The Iterated Prover*

The evaluation of the VDF, i.e. the computation of $y = g^{2^t}$, takes time $T = \delta t$, where δ is the time of one squaring in the underlying group. As demonstrated in Sect. 4.1, the proof π can be computed in $O(t/\log(t))$ group operations. Say that the total time of computing the proof is a fraction T/ω; considering Algorithm 5, one can think of $\omega = 20$, a reasonable value for practical parameters. One potential issue with the proposed VDF is that the computation of π can only start after the evaluation of the VDF output g^{2^t} is completed. So after the completion of the VDF evaluation, there still remains a total amount T/ω of work to compute the proof. We call *overhead* these computations that must be done after the evaluation of $y = g^{2^t}$. Even though this part of the computation can be parallelised, it might be advantageous for some applications to reduce the overhead to a negligible amount of work.

We show in the following that using only two parallel threads, the overhead can be reduced to a total cost of about T/ω^n, at the cost of lengthening the proofs to n group elements (instead of a single one), and $n - 1$ small prime numbers. Note that the value of ω varies with T, yet for simplicity of exposition, we assume that it is constant in the following analysis (a reasonable approximation for practical purposes).

The idea is to start proving some intermediate results before the full evaluation is over. For instance, consider $t_1 = t\frac{\omega}{\omega+1}$. Run the evaluator, and when the intermediate value $g_1 = g^{2^{t_1}}$ is reached, store it (but keep the evaluator running in parallel), and compute the proof π_1 for the statement $g_1 = g^{2^{t_1}}$. The computation of this proof takes time about $\delta t_1/\omega = T/(\omega+1)$, which is the time it takes to finish the full evaluation (i.e. going from g_1 to $y = g^{2^t} = g_1^{2^{t/(\omega+1)}}$). Therefore, the evaluation of y and the first proof π_1 finish at the same time. It only remains to produce a proof π_2 for the statement $y = g_1^{2^{t/(\omega+1)}}$, which can be done in total time $\frac{\delta t}{\omega(\omega+1)} \leq T/\omega^2$. Therefore, the overhead is at most T/ω^2. At first glance, it seems the verification requires the triple (g_1, π_1, π_2), but in fact, the value g_1 can be recovered from π_1 and the prime number $\ell_1 = H_{\texttt{prime}}(\texttt{bin}(g)|||\texttt{bin}(g_1))$ via $g_1 = \pi_1^{\ell_1} g^{t_1 \bmod \ell_1}$, as done is Sect. 4.2. Therefore, the proof can be compressed to (ℓ_1, π_1, π_2).

More generally, one could split the computation into n segments of length $t_i = t\omega^{n-i}\frac{\omega-1}{\omega^n-1}$, for $i = 1, \ldots, n$. We have that $t = \sum_{i=1}^n t_i$, and $t_i = t_{i-1}/\omega$, so during the evaluation of each segment (apart from the first), one can compute the proof corresponding to the previous segment. The overhead is only the proof of the last segment, which takes time $T\frac{\omega-1}{\omega(\omega^n-1)} \leq T/\omega^n$. The proof consists of the n intermediate proofs and the $n - 1$ intermediate prime challenges.

The above construction has since been generalised by Döttling et al. [17], who show that this trick can in fact be applied to a large family of VDFs (*self-composable* VDFs): they describe a general compiler that transforms a non-tight VDF (with a $O(t)$ overhead) into a tight VDF (with a $O(1)$ overhead).

8.2. *The Hybrid Prover*

The hybrid prover consists in running a few rounds of the Pietrzak prover, then finishing the proof with a short instance of our prover. In the following protocol, the input is (g, h, T) where g and h are two elements of a group G, and T is a positive integer. For simplicity, we assume T is a power of 2. The prover tries to convince the verifier that $h = g^{2^T}$. Let N be a positive integer.

1. The verifier checks that $g, h \in G$, and rejects if not.
2. Set $g_0 \leftarrow g$, $h_0 \leftarrow h$, $t \leftarrow T$ and $n \leftarrow 0$.
3. If $n < N$ do:
   (a) Set $t \leftarrow t/2$ and $n \leftarrow n + 1$.
   (b) The prover computes $v_n \leftarrow g_{n-1}^{2^t}$ and sends v_n to the verifier. The verifier checks that $v_n \in G$, and rejects if not.
   (c) The verifier sends a random $r_n \in \{1, \ldots, 2^k\}$.
   (d) Both compute $g_n \leftarrow g_{n-1}^{r_n} v_n$ and $h_n \leftarrow v_n^{r_n} h_{n-1}$.
   (e) Go to Step 3.
4. The verifier sends to the prover a random prime ℓ from $\mathrm{Primes}(2k)$.
5. The prover computes $\pi \leftarrow g_N^{\lfloor 2^t/\ell \rfloor}$ and sends it to the verifier.
6. The verifier computes $r \leftarrow 2^t \mod \ell$ and accepts if $\pi \in G$ and $h_N = \pi^\ell g_N^r$.

We refer to the loop in Step 3 as the *Pietrzak rounds* and to the three final steps as the *final round*. Making this protocol non-interactive with the Fiat–Shamir transform, the proof produced consists of $N + 1$ group elements.

Security. The security of this construction follows directly from the security of the Pietrzak scheme and of our scheme. Suppose that $h \neq g^{2^T}$. The security of the Pietrzak scheme ensures that at each round, for any efficient prover, we have $h_n \neq g_n^{2^{T/2^n}}$ with overwhelming probability. Then, if $h_N \neq g_N^{2^{T/2^N}}$, the security of our scheme ensures that for any efficient adversary, the final verification fails with overwhelming probability. More precisely, we have the following security theorem.

Theorem 4 *If there is a malicious prover $\mathcal{P}$ breaking the soundness of the above interactive succinct argument with probability p, then there is a player $\mathcal{B}$ winning the root finding game $\mathcal{G}^{\mathsf{root}}$ with probability at least $p^2/(8N) - p/2^{k+2}$, with essentially the same running time as $\mathcal{P}$.*

Proof Let $\mathcal{P}_1$ be a prover that plays like $\mathcal{P}$, but abandons if $h_N = g_N^{2^{T/2^N}}$, and similarly, $\mathcal{P}_2$ plays like $\mathcal{P}$ but abandons if $h_N \neq g_N^{2^{T/2^N}}$. Intuitively, $\mathcal{P}_1$ tries to cheat during the Pietrzak rounds, and $\mathcal{P}_2$ tries to cheat during the final round. Let p_1 and p_2 be their respective winning probabilities. We then have $p_1 + p_2 \geq p$. From the proof of [5, Theorem 1], the prover $\mathcal{P}_1$ allows us to construct an algorithm $\mathcal{A}_1$ breaking the low-order assumption with probability $p_1' \geq p_1^2/N - p_1/2^k$. Recall from [5, Definition 1] that breaking the low-order assumption consists in finding an element $\mu \in G$, and an integer $d < 2^\lambda$ such that $\mu \neq 1_G$ and $\mu^d = 1_G$. Such a pair (μ, d) immediately allows to solve the root finding game. Similarly, from [5, Theorem 2], the prover $\mathcal{P}_2$ allows us to construct an algorithm $\mathcal{A}_2$ succeeding at the root finding game with probability $p_2' \geq p_2$. Let $\mathcal{A}$ be the algorithm that chooses uniformly at random $b \in \{1, 2\}$ and then plays the root finding game with $\mathcal{A}_b$. Then, $\mathcal{A}$ wins with probability at least

$$(p_1' + p_2')/2 \geq (p_1^2/N - p_1/2^k + p_2)/2.$$

The facts that either $p_1 \geq p/2$ or $p_2 \geq p/2$, and that $x \mapsto \max(x^2/N - x/2^k, 0)$ is monotonically increasing, allow us to conclude. □

Efficiency We have the formula

$$v_n = \prod_{b_1,\ldots,b_{n-1} \in \{0,1\}} \left(g^{2^{[b_1 \ldots b_{n-1} 1]_2 T/2^n}}\right)^{\prod_i r_i^{1-b_i}},$$

where $[b_1 \ldots b_n]_2$ is the integer with binary representation $b_1 \ldots b_n$. It can easily be proved inductively, by simultaneously proving the formula

$$g_n = \prod_{b_1,\ldots,b_n \in \{0,1\}} \left(g^{2^{[b_1 \ldots b_n]_2 T/2^n}}\right)^{\prod_i r_i^{1-b_i}}.$$

Therefore, the values $v_1, \ldots, v_N$ can be computed in about 2^N small group exponentiations, so the Pietrzak rounds can be performed efficiently for small values of the number of rounds N.

Now, there are two options to run the final phase. Let $t = T/2^N$. First, one could simply compute $g_n^{\lfloor 2^t/\ell \rfloor}$ in a straightforward manner, by computing the exponent as a Euclidean division then exponentiating with a square-and-multiply algorithm. That would require $(1+o(1))t$ group operations. Second, it is possible to run the optimised prover discussed in Sect. 4.1, but that requires to get values $C_i = g_N^{2^{i\kappa\gamma}}$ for some parameters κ and γ, for each $i = 0, \ldots, \lceil t/(\kappa\gamma) \rceil$. The values $g^{2^{i\kappa\gamma}}$ can be precomputed at no additional cost since they are intermediate steps of the VDF evaluation. We can use them to compute the values C_i as fast as possible. Observe that

$$C_i = g_N^{2^{i\kappa\gamma}} = \prod_{b_1,\ldots,b_N \in \{0,1\}} \left(g^{2^{i\kappa\gamma + [b_1 \ldots b_N]_2 t}}\right)^{\prod_i r_i^{1-b_i}}.$$

Suppose we have precomputed and stored all the values $g^{2^{i\kappa\gamma + [b_1 \ldots b_N]_2 t}}$.

Lemma 2 *Given n integers $r_0, \dots, r_{n-1}$ and 2^n group elements $x_0, \dots x_{2^n-1}$, one can compute the product*

$$C = \prod_{a=0}^{2^n-1} x_a^{\prod_{i=0}^{n-1} r_i^{\mathtt{bit}_i(a)}}$$

at the cost of $2^n - 1$ group exponentiations by r_i-values, and $2^n - 1$ additional group multiplications.

Proof We can compute C as follows. First, let $C_0(a) = x_a$. For every $j = 1, \dots, n$ and every $a = 0, \dots, 2^{n-j} - 1$, compute the value

$$C_j(a) = C_{j-1}(a) \cdot C_{j-1}(a + 2^{n-j})^{r_{n-j}},$$

for a total cost of $2^n - 1$ group exponentiations by r_i-values and $2^n - 1$ additional group multiplications. A simple induction shows that for each j we have

$$C = \prod_{a=0}^{2^{n-j}-1} C_j(a)^{\prod_{i=0}^{n-j-1} r_i^{\mathtt{bit}_i(a)}}.$$

In particular, we get $C = C_n(1)$. □

We can compute C_i as in Lemma 2 with the values $x_a = g^{2^{i\kappa\gamma+\tilde{a}t}}$, where $a = \sum_{i=0}^{N-1} a_i 2^i$ is the binary expansion of a and $\tilde{a} = \sum_{i=0}^{N-1}(1-a_{N-i})2^i$. In total, computing C_i with this strategy requires $2^N - 1$ exponentiations by r_i-values and as many additional group multiplications. Using a standard square-and-multiply exponentiation amounts to about $1.5k2^N$ group operations (yet this factor 1.5 can be decreased in practice, for instance with a sliding window approach).

After this update, computing π following Argon Design's strategy (Algorithm 6) costs about $t/\kappa + \gamma 2^\kappa$ group operations. In total, this also requires the storage of about $2^N t/\kappa\gamma$ precomputed values $C_{i,0}(x)$. The total cost of computing the proof is then about

$$t/\kappa + \gamma 2^\kappa + 1.5k2^N t/\kappa\gamma$$

group operations. For design purposes, one would fix the parameters T and N (leading to proofs of $N + 1$ group elements), let $t = T/2^N$ and compute the optimal parameters γ and κ to minimise the cost of computing the proof, under the constraint that $2^N t/\kappa\gamma$ precomputed group elements must be stored in memory.

8.3. *Comparison with the Iterated Prover*

Both the iterated prover and the hybrid prover provide a trade-off between prover efficiency and length of the proofs. A key difference is that the hybrid construction can rather consistently divide the overhead by 2 when N is increased by 1, while the behaviour of the iterated construction is not as easy to control and strongly depends on

Table 1. Comparing the hybrid prover and the iterated prover with the following parameters: the security parameter is $k = 128$, the timing parameter is set to 2^{39} sequential squarings (corresponds to about 9 min of evaluation, assuming the latency of a squaring is 1 ns), the group is an RSA group for a 2048 bit modulus, 8MB of precomputed data is available, and 4/1.2 parallel multiplications can be computed during the latency of a single squaring .

Hybrid prover			Iterated prover		
n	Proof length (B)	Overhead (%)	n	Proof length (B)	Overhead (%)
1	256	2.8	1	256	2.8
2	512	1.5	2	544	0.077
3	768	0.84	3	832	0.0022
4	1024	0.47	4	1120	0.000061
5	1280	0.26	5	1408	0.0000017

The *overhead* is the time it takes to compute the proof as a percentage of the time required by the sequential evaluation of the VDF. In both cases, $n = 1$ corresponds to the plain prover. For the hybrid prover, $n = N + 1$ is the number of Pietrzak rounds plus one; for the iterated prover, n is the number of iterations

the performance of the simple (non-iterated) prover. As explained in Sect. 8.1, if the overhead to compute the proof of a VDF with time parameter T takes time T/ω for some constant ω, the n-iterated variant has an overhead of at most T/ω^n. But in reality, ω is not constant, it depends on T. Worse, analysing the algorithms is not sufficient to derive ω: it strongly depends on the hardware performance. The prover needs to perform a large number of group multiplications, possibly in parallel. If many multiplications can be done (in parallel) during the time it takes to perform a single squaring, then ω gets smaller. On the other hand, if multiplications are significantly costlier than squarings, then ω gets larger.

As all the possible variations of these parameters are hard to predict, we make available a Python script[7] that computes the cost of each strategy for tuneable parameters, including the relative cost of multiplication and squaring and the amount of available memory for the precomputed values. Note that this script estimates the cost of each strategy only in terms of group operations, neglecting the cost of seemingly simpler tasks (like table lookups), which might have an impact in practice if the group arithmetic becomes too efficient. Results for two sets of parameters are presented in Tables 1 and 2. It appears that the overhead of the iterated prover decreases significantly faster than the hybrid prover, for comparable proof length. It should however be emphasised that this efficiency comes at a cost: the iterated prover is more hardware intensive, since the computation of the proof starts before the end of the sequential evaluation (affecting power consumption and cooling).

9. Circumventing Impossibility Results with Timing Assumptions

In addition to the applications mentioned in introduction, we conclude this paper by showing that a *trapdoor* VDF also constitutes a new tool for circumventing classic im-

[7] https://github.com/Calodeon/vdf/blob/master/estimation.py.

Table 2. Comparing the hybrid prover and the iterated prover .

Hybrid prover			Iterated prover		
n	Proof length (B)	Overhead (%)	n	Proof length (B)	Overhead (%)
1	256	9.4	1	256	9.4
2	512	5.1	2	544	0.80
3	768	2.8	3	832	0.075
4	1024	1.6	4	1120	0.0070
5	1280	0.87	5	1408	0.00066

The parameters are the same as Table 1, except that a single multiplication can be computed during the latency of a single squaring

possibility results. We illustrate this through a simple identification protocol constructed from a trapdoor VDF, where a party, Alice, wishes to identify herself to Bob by proving that she knows the trapdoor. Thanks to the VDF timing properties, this protocol features surprising zero-knowledge and deniability properties challenging known impossibility results.

As this discussion slightly deviates from the crux of the article (the construction of a trapdoor VDF), most of the details are deferred to Appendices B and C, and this section only introduces the main ideas. As in the rest of the paper, the parameter k is the security level. The identification protocol goes as follows:

1. Bob chooses a challenge $c \in \{\mathtt{0}, \mathtt{1}\}^k$ uniformly at random. He sends it to Alice, along with a time limit Δ, and starts a timer.
2. Alice responds by sending the evaluation of the VDF on input c (with time parameter Δ), together with the proof. She can respond fast using her trapdoor.
3. Upon receiving the response, Bob stops the timer. He accepts if the verification of the VDF succeeds and the elapsed time is smaller than Δ.

Remark 7 We present here only an identification protocol, but it is easy to turn it into an authentication protocol for a message m by having Alice use the concatenation $c||m$ as input to the VDF.

Since only Alice can respond immediately thanks to her secret, Bob is convinced of her identity. Since anyone else can compute the response to the challenge in time Δ, the exchange is perfectly simulatable, hence perfectly zero-knowledge. It is well-known (and in fact clear from the definition) that a classic interactive zero-knowledge proof [21] cannot have only one round (this would be a challenge-response exchange, and the simulator would allow to respond to the challenge in polynomial time, violating soundness). The above protocol avoids this impossibility thanks to a modified notion of soundness, ensuring that only Alice can respond *fast enough*. This is made formal in Appendix B, via the notion of zero-knowledge timed challenge-response protocol.

Remark 8 Note that this very simple protocol is also efficient: the "time-lock" evaluation of the VDF does not impact any of the honest participants, and it is only meant to be used by the simulator. Only the trapdoor evaluation and the verification are actually executed.

Finally, this protocol has strong deniability properties. Indeed, since anyone can produce in time Δ a response to any challenge, any transcript of a conversation that is older than time Δ could have been generated by anyone. In fact the protocol is *on-line deniable* against any judge that suffers a communication delay larger than $\Delta/2$. Choosing Δ to be as short as possible (while retaining soundness) yields a strongly deniable protocol. Deniable authentication was introduced in [16,20], and Pass [28] proved several possibility and impossibility results for deniable zero-knowledge proofs in the CRS or random oracle model. However, these results focus on a weaker notion of deniability: in a sense, only the final transcript is deniable. Pass [28] shows that this form of deniability can be achieved in the random oracle model for zero-knowledge arguments of knowledge. The stronger notion of on-line deniability was introduced in [15] and proven to be impossible in a PKI. Using a delay assumption, we provide a new way to circumvent this impossibility. This is discussed in more detail in Appendix C.

Acknowledgements

Part of this work was supported by the Ethereum Foundation under Grant FY19-0090 and by the ERC Advanced Grant 740972 (ALGSTRONGCRYPTO). The author wishes to thank a number of people with whom interesting discussions helped improve the present work, in alphabetical order, Dan Boneh, Justin Drake, Alexandre Gélin, Novak Kaluđerović, Arjen K. Lenstra, and Serge Vaudenay.

A. Proof of Remark 1

Model H as a random oracle. Suppose that

$$\begin{aligned}\mathsf{trapdoor}^{H}_{\mathsf{sk}}(x,\Delta) &= t_{\mathsf{sk}}(H(x),\Delta),\\ \mathsf{eval}^{H}_{\mathsf{pk}}(x,\Delta) &= e_{\mathsf{pk}}(H(x),\Delta), \mathit{and}\\ \mathsf{verify}_{\mathsf{pk}}(x,y,\Delta) &= v_{\mathsf{pk}}(H(x),y,\Delta),\end{aligned}$$

for procedures t, e and v that do not have access to H.
Let $\mathcal{A}$ be a player of the Δ-evaluation race game. Assume that the output $\mathcal{B}$ of $\mathcal{A}$ is limited to a number q of queries to $\mathcal{O}$ and H. We are going to build an algorithm $\mathcal{A}'$ that wins with same probability as $\mathcal{A}$ when its output $\mathcal{B}'$ is not given access to $\mathcal{O}$.
Let $(Y_i)_{i=1}^{q}$ be a sequence of random hash values (i.e. uniformly distributed random values in $\{0,1\}^{2k}$). First observe that $\mathcal{A}$ wins the Δ-evaluation race game with the same probability if the last step runs the algorithm $\mathcal{B}^{\mathcal{O}',H'}$ instead of $\mathcal{B}^{\mathcal{O},H}$, where

1. H' is the following procedure: for any new requested input x, if x has previously been requested by $\mathcal{A}$ to H then output $H'(x) = H(x)$; otherwise set $H'(x)$ to be the next unassigned value in the sequence (Y_i);

2. $\mathcal{O}'$ is an oracle that on input x outputs $t_{\mathsf{sk}}(H'(x), \Delta)$.

With this observation in mind, we build $\mathcal{A}'$ as follows. On input pk, $\mathcal{A}'$ first runs $\mathcal{A}^H$ which outputs $\mathcal{A}^H(\mathsf{pk}) = \mathcal{B}$. Let X be the set of inputs of the requests that $\mathcal{A}$ made to H. For any $x \in X$, $\mathcal{A}'$ computes and stores the pair $(H(x), \mathsf{eval}_{\mathsf{pk}}(x, \Delta))$ in a list L. In addition, it computes and stores $(Y_i, e_{\mathsf{pk}}(Y_i, \Delta))$ for each $i = 1, \ldots, q$, and adds them to L.

Consider the following procedure $\mathcal{O}'$: on input x, look for the pair of the form $(H'(x), \sigma)$ in the list L, and output σ. The output of $\mathcal{A}'$ is the algorithm $\mathcal{B}' = \mathcal{B}^{\mathcal{O}', H'}$. It does not require access to the oracle $\mathcal{O}$ anymore: all the potential requests are available in the list of precomputed values. Each call to $\mathcal{O}$ is replaced by a lookup in the list L, so $\mathcal{B}'$ has essentially the same running time as $\mathcal{B}$. Therefore $\mathcal{A}'$ wins the Δ-evaluation race game with same probability as $\mathcal{A}$ even when its output $\mathcal{B}'$ is not given access to a evaluation oracle.

B. Timed Challenge-Response Identification Protocols

A timed challenge-response identification protocol has four procedures:

$\mathsf{keygen} \to (\mathsf{pk}, \mathsf{sk})$ is a key generation procedure, which outputs a prover's public key pk and secret key sk.
$\mathsf{challenge} \to c$ which outputs a random challenge.
$\mathsf{respond}_{\mathsf{sk}}(c, \Delta) \to r$ is a procedure that uses the prover's secret key to respond to the challenge c, for the time parameter Δ.
$\mathsf{verify}_{\mathsf{pk}}(c, r, \Delta) \to \mathsf{true}$ or false is a procedure to check if r is a valid response to c, for the public key pk and the time parameter Δ.

The security level k is implicitly an input to each of these procedures. The keygen procedure is used the generate Alice's public and secret keys; then, the identification protocol is as follows:

1. Bob generates a random c with the procedure $\mathsf{challenge}$. He sends it to Alice, along with a time limit Δ, and starts a timer.
2. Alice responds $r = \mathsf{respond}_{\mathsf{sk}}(c, \Delta)$.
3. Bob stops the timer. He accepts if $\mathsf{verify}_{\mathsf{pk}}(c, r, \Delta) = \mathsf{true}$ and the elapsed time is smaller than Δ.

Given a time parameter Δ, a *Δ-response race game* and an associated notion of *Δ-soundness* can be defined in a straightforward manner as follows.

Definition 9 (*Δ-response race game*) Let $\mathcal{A}$ be a party playing the game. The parameter $\Delta : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ is a function of the (implicit) security parameter k. The *Δ-response race game* goes as follows:

1. The random procedure keygen is run and it outputs a public key pk;
2. $\mathcal{A}(\mathsf{pk})$ outputs an algorithm $\mathcal{B}$;
3. A random challenge c is generated according to the procedure $\mathsf{challenge}$;
4. $\mathcal{B}^{\mathcal{O}}(c)$ outputs a value r, where $\mathcal{O}$ is an oracle that outputs the evaluation $\mathsf{respond}_{\mathsf{sk}}$ (c', Δ) on any input $c' \neq c$.

Then, $\mathcal{A}$ wins the game if $T(\mathcal{B}, c) < \Delta$ and $\mathsf{verify}_{\mathsf{pk}}(c, r, \Delta) = \mathsf{true}$.

Definition 10 (Δ-*soundness*) A timed challenge-response identification protocol is Δ-*sound* if any polynomially bounded player (with respect to the implicit security parameter) wins the above Δ-response race game with negligible probability.

It is as immediate to verify that a sound and Δ-sequential trapdoor VDF gives rise to a Δ-sound identification protocol (via the construction of Sect. 9). Similarly, the *completeness* of the identification protocol (that a honest run of the protocol terminates with a successful verification) is straightforward to derive from the fact that the verification of a valid VDF output always outputs true. There simply is one additional requirement: if the procedure $\mathsf{respond}_{\mathsf{sk}}(c, \Delta)$ requires computation time at least ϵ_1, and the channel of communication has a transmission delay at least ϵ_2, we must have $\epsilon_1 + 2\epsilon_2 < \Delta$. Finally the *zero-knowledge* property is defined as follows.

Definition 11 (*Zero-knowledge*) A timed challenge-response identification protocol is (perfectly, computationally, or statistically) zero-knowledge if there is an algorithm $\mathcal{S}$ that on input k, Δ, pk and a random $\mathsf{challenge}(k, \Delta)$ produces an output (perfectly, computationally, or statistically) indistinguishable from $\mathsf{respond}_{\mathsf{sk}}(c, k, \Delta)$, and the running time of $\mathcal{S}$ is polynomial in k.

In a classical cryptographic line of though, this zero-knowledge property is too strong to allow for any soundness, since an adversary can respond to the challenge with a running time polynomial in the security parameter of Alice's secret key. This notion starts making sense when the complexity of the algorithm $\mathcal{S}$ is governed by another parameter, here Δ, independent from Alice's secret.
For the protocol derived from a VDF, the zero-knowledge property is ensured by the fact that anyone can compute Alice's response to the challenge in time polynomial in k, with the procedure eval.

C. Local Identification

The challenge-response identification protocol derived from a VDF in Sect. 9 is totally deniable against a judge, Judy, observing the communication from a long distance. The precise definition of on-line deniability is discussed in [15]. We refer the reader there for the details, but the high-level idea is as follows. Alice is presumably trying to authenticate her identity to Bob. Judy will rule whether or not the identification was attempted. Judy interacts with an informant who is witnessing the identification and who wants to convince Judy that it happened. This informant could also be a misinformant, who is not witnessing any identification, but tries to deceive Judy into believing it happened. The protocol is online deniable if no efficient judge can distinguish whether she is talking to an informant or a misinformant. The (mis)informant is allowed to corrupt Alice or Bob, at which point he learns their secret keys and controls their future actions. When some party is corrupted, Judy learns about it.

It is shown in [15] that this strong deniability property is impossible to achieve in a PKI. To mitigate this issue, they propose a secure protocol in a relaxed setting, allowing incriminating aborts. We propose an alternative relaxation of the setting, where Judy is assumed to be far away from Alice and Bob (more precisely: the travel time of a message between Alice and Bob is shorter than between Alice (or Bob) and Judy[8]). For example, consider a building whose access is restricted to authorised card holders. Suppose the card holders do not want anyone other than the card reader to get convincing evidence that they are accessing the building (even if the card reader is corrupted, it cannot convince anyone else). Furthermore, Alice herself cannot convince anyone that the card reader ever acknowledged her identification attempt. In this context, the card and the card reader benefit from very efficient communications, while a judge farther away would communicate with an additional delay. An identification protocol can exploit this delay to become deniable, and this is achieved by the timed challenge-response identification protocol derived from a VDF.

The idea is the following. Suppose that the distance between Alice and Judy is long enough to ensure that the travel time of a message from Alice to Judy is larger than $\Delta/2$. Then, Judy cannot distinguish a legitimate response of Alice that took some time to reach her from a response forged by a misinformant that is physically close to Judy.

More precisely, considering an informant I who established a strategy with Judy, we can show that there is a misinformant M that Judy cannot distinguish from I. First of all, Bob cannot be incriminated since he is not using a secret key. It all boils down to tracking the messages that depend on Alice's secret key. Consider a run of the protocol with the informant I. Let t_0 be the point in time where Alice computed $s = \mathsf{trapdoor}_{\mathsf{sk}}(c, \Delta)$. The delay implies two things:

1. The challenge c is independent of anything Judy sent after point in time $t_0 - \Delta/2$.
2. The first message Judy receives that can depend on s (and therefore the first message that depends on Alice's secret) arrives after $t_0 + \Delta/2$.

From Point 1, at time $t_0 - \Delta/2$, the misinformant (who is close to Judy) can already generate c (following whichever procedure I and Judy agreed on), and start evaluating $\mathsf{eval}_{\mathsf{pk}}(c, \Delta)$. The output is ready at time $t_0 + \Delta/2$, so from Point 2, the misinformant is on time to send to Judy messages that should depend on the signature s.

In practice. The protocol is deniable against a judge at a certain distance away from Alice and Bob, and the minimal value of this distance depends on Δ. An accurate estimation of this distance would require in the first place an equally accurate estimation of the real time Δ (in s) a near-optimal adversary would need to forge the response. This non-trivial task relates to the discussion of Sect. 3.2.

Assuming reasonable bounds for Δ have been established, one can relate the distance and the communication delay in a very conservative way through the speed of light. We want Judy to stand at a sufficient distance to ensure that any message takes at least $\Delta/2$ s to

[8] A message does not travel directly from Alice (or Bob) to Judy, since Judy is only communicating with the (mis)informant. What is measured here is the sum of the delay between Alice and the (mis)informant and the delay between the (mis)informant and Judy. There is no constraint on the location of the (mis)informant, but we assume a triangular inequality: he could be close to Alice and Bob, in which case his communications with Judy suffer a delay, or he could be close to Judy, in which case his interactions with Alice and Bob are delayed.

travel between them, so Judy should be at least $c\Delta/2$ m away, where $c \approx 3.00 \times 10^8$ m/s is the speed of light. For security against a judge standing 100 m away, one would require $\Delta \approx 0.66\,\mu$s. Alice should be able to respond to Bob's challenge in less time than that. At this point, it seems unreasonable to assume that such levels of precision can be achieved (although in principle, distance bounding protocols do deal with such constraints), yet it remains interesting that such a simple and efficient protocol provides full deniability against a judge that suffers more serious communication delays.

References

[1] M. Bellare and S. Goldwasser. Encapsulated key escrow. Technical report, 1996.

[2] M. Bellare and S. Goldwasser. Verifiable partial key escrow. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, CCS '97, pages 78–91, New York, NY, USA, 1997. ACM.

[3] I. Biehl, J. Buchmann, S. Hamdy, and A. Meyer. A signature scheme based on the intractability of computing roots. *Designs, Codes and Cryptography*, 25(3):223–236, 2002.

[4] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In E. F. Brickell, editor, *Advances in Cryptology – CRYPTO 2018*, pages 757–788. Springer, 2018.

[5] D. Boneh, B. Bünz, and B. Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712, 2018. https://eprint.iacr.org/2018/712.

[6] D. Boneh, B. Bünz, and B. Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In A. Boldyreva and D. Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, volume 11692 of *Lecture Notes in Computer Science*, pages 561–586. Springer, 2019.

[7] D. Boneh and M. Franklin. Efficient generation of shared RSA keys. In *Annual International Cryptology Conference*, pages 425–439. Springer, 1997.

[8] D. Boneh and M. Naor. Timed commitments. In M. Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 236–254. Springer Berlin Heidelberg, 2000.

[9] J. Buchmann and S. Hamdy. A survey on IQ cryptography. In *In Proceedings of Public Key Cryptography and Computational Number Theory*, pages 1–15, 2001.

[10] J. Buchmann and H. C. Williams. A key-exchange system based on imaginary quadratic fields. *Journal of Cryptology*, 1(2):107–118, 1988.

[11] B. Bünz, B. Fisch, and A. Szepieniec. Transparent snarks from DARK compilers. Cryptology ePrint Archive, Report 2019/1229, 2019. https://eprint.iacr.org/2019/1229.

[12] B. Cohen and K. Pietrzak. The Chia network blockchain. Chia network, white paper, 2019. https://vdfresearch.org/assets/P0137-R-004b%20(VDF%20proof%20feasibility%20study).pdf.

[13] D. A. Cox. *Primes of the form x^2+ny^2: Fermat, class field theory, and complex multiplication*, volume 34. John Wiley & Sons, 2011.

[14] CPU-Z OC world records. http://valid.canardpc.com/records.php, 2018.

[15] Y. Dodis, J. Katz, A. Smith, and S. Walfish. *Composability and On-Line Deniability of Authentication*, pages 146–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[16] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM review*, 45(4):727–784, 2003.

[17] N. Döttling, S. Garg, G. Malavolta, and P. N. Vasudevan. Tight verifiable delay functions. Cryptology ePrint Archive, Report 2019/659, 2019. https://eprint.iacr.org/2019/659.

[18] J. Drake. Ethereum 2.0 randomness. August 2018 workshop at Stanford hosted by the Ethereum Foundation and the Stanford Center for Blockchain Research, 2018. https://www.chia.net/assets/ChiaGreenPaper.pdf.

[19] J. Drake. Minimal VDF randomness beacon. Ethereum Research post, 2018. https://ethresear.ch/t/minimal-vdf-randomness-beacon/3566.

[20] C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. *Journal of the ACM (JACM)*, 51(6):851–898, 2004.

[21] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[22] J. L. Hafner and K. S. McCurley. A rigorous subexponential algorithm for computation of class groups. *Journal of the American mathematical society*, 2(4):837–850, 1989.

[23] A. Joux, D. Naccache, and E. Thomé. When *e*-th roots become easier than factoring. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2007.

[24] J. Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 723–732, 1992.

[25] A. K. Lenstra and B. Wesolowski. Trustworthy public randomness with sloth, unicorn and trx. *International Journal of Applied Cryptology*, 2016.

[26] L. Long. Binary quadratic forms. Chia network, Chia VDF Competition Guide, 2019. https://github.com/Chia-Network/vdf-competition/blob/master/classgroups.pdf.

[27] S. Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[28] R. Pass. On deniability in the common reference string and random oracle model. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 316–337. Springer, 2003.

[29] K. Pietrzak. Simple verifiable delay functions. Cryptology ePrint Archive, Report 2018/627, Version 20180626:145529, 2018. https://eprint.iacr.org/2018/627.

[30] M. O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256 – 267, 1983.

[31] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.

[32] T. Sander. Efficient accumulators without trapdoor extended abstract. In *International Conference on Information and Communications Security*, pages 252–262. Springer, 1999.

[33] R. Swarbrick. VDF proof feasibility study. Argon Design, technical report, 2018. https://vdfresearch.org/assets/P0137-R-004b%20(VDF%20proof%20feasibility%20study).pdf.

[34] U. Vollmer. Asymptotically fast discrete logarithms in quadratic number fields. In *International Algorithmic Number Theory Symposium (ANTS)*, pages 581–594. Springer, 2000.