Efficient verifiable delay functions

Benjamin Wesolowski

École Polytechnique Fédérale de Lausanne, EPFL IC LACAL, Switzerland

Abstract. We construct a verifiable delay function (VDF). A VDF is a function whose evaluation requires to run 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 a 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 both a single element of the group), and the verification of correctness is very efficient.

Note. The present version is an update of a preprint submitted on 20 June 2018. The terminology has been updated to match with the recent work [4], and the construction has been simplified.

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]. These functions should be computable in a prescribed amount of time $\varDelta$, 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 cores), and the result should be easy to verify (i.e., for a cost $\mathrm{polylog}(\varDelta)$). These special functions are used in [11] (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 unpredictible. These randomness beacons, introduced by Rabin in [13], are a valuable tool in a public, decentralised setting, as it is not trivial for someone to flip a coin and convince their peers that the outcome was not rigged. The VDF proposed in [11], *sloth*, is not asymptotically efficiently verifiable: the verification procedure (given x and y, verify that $\mathtt{sloth}(x) = y$) is faster than the evaluation procedure (given x, compute the value $\mathtt{sloth}(x)$) only by a linear factor. The new construction provides an exponentially faster verification.

The paper [4] was developed independently from the present work, yet we adopt their terminology for verifiable delay functions, for the sake of uniformity. In addition to compiling a variety of interesting applications of such functions in decentralised systems (notably for resource-efficient blockchains), the authors of [4] propose practical constructions that also achieve an exponential gap between evaluation and verification. These constructions, however, do not strictly achieve the requirements of a VDF. For one of them, the evaluation requires an amount $\mathrm{polylog}(\varDelta)$ of parallelism to run in parallel time $\varDelta$. The other one is insecure against an adversary that can run a large (but feasible) pre-computation, so the setup must be regularly updated. The construction we propose is secure against pre-computation attacks, and the evaluation requires a small, constant amount of parallelism.

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

We propose a practical construction based on groups G of unknown order (such as an RSA group $(\mathbf{Z}/N\mathbf{Z})^{\times}$, where N is a product of two large primes, or the class group of an imaginary quadratic field). The trapdoor is the order of the group. The security of the construction is proven assuming the classic time-lock assumption of [14] (but in G instead of necessarily in an RSA group), and the difficulty of extracting roots in G.

Deriving a verifiable delay function. Suppose that a public key pk for a trapdoor VDF is given without any known associated private 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 are 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 [15]), 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 contributing some secret random seeds (as done in [5]), but a third party would have to assume that the parties involved in this computation did not collude to retrieve the secret. A better approach would be to use the class group of an imaginary quadratic order. Indeed, 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 in [8], exploiting the difficulty of computing their orders (and the fact that this order problem is closely related to the discrete logarithm problem and the root problem in this group). To this day, the best know 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 [10] and [16].

Contribution. Fix a timing parameter Δ, a security level k (say, $128, 192$, or 256), and a well-chosen group G. The construction consists in solving an instance of the time-lock puzzle of [14] in the group G (for a timing parameter Δ), and compute a proof of correctness. Our construction has the following properties.

1. It is Δ-sequential (meaning that it takes Δ sequential steps to evaluate) assuming the classic time-lock assumption of [14] in the group G.

2. It is sound (meaning that one cannot produce a 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.
4. The verification of correctness requires essentially two exponentiations in the group G, with exponents of bit-length the security parameter k.
5. The proof can be produced in time about $\varDelta$.

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

Note that since the method we describe to compute the proof takes time $\varDelta$, there is an interval between the guaranteed sequential work $\varDelta$ and the total work $2\varDelta$. In general, this is of no importance since anyway, computational models do not capture well small constant factors with respect to real-world running time: this factor 2 is smaller than the speedup gained from a low-end CPU to a high-end CPU, or from any CPU to some piece of specialised hardware. Precise timing seems unlikely to be achievable without resorting to trusted hardware, thus applications of VDF's are designed not to be too sensitive to these small factors. If despite these facts precise timing is still considered critical in some application, one could first publish the result of the evaluation with a different, more efficiently computable, temporary proof (which might be heavier and less efficiently verifiable than our construction) and then replace it with the lighter and more efficiently verifiable proof when available. A candidate for such temporary proofs could be [12] (developed independently from the present work, and discussed in Section 1.1), assuming their security claims still hold in other groups than the particular RSA groups they require. The proofs of [12] are compatible with the present construction; they are much heavier and take longer to verify (by a factor $O(\log(\varDelta))$), but they can be produced in time $O(\sqrt{\varDelta}\log(\varDelta))$.

1.1 Time-sensitive cryptography and related work

Rivest, Shamir and Wagner [14] 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 [6] 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 [11] as a tool to provide trust to the generation of public random numbers. These slow-timed hash functions were recently revisited and formalised by Boneh *et al.* in [4] under the name of verifiable delay functions.

Pietrzak's verifiable delay function. Independently from the present work, another efficient VDF was proposed in [12]. The author describes an elegant construction, provably secure under the classic time-lock assumption of [14] when implemented over an RSA group $(\mathbf{Z}/N\mathbf{Z})^\times$ where N is a product of two safe primes. As discussed earlier, generating a public RSA modulus is a difficult task without a trusted environment. Note that it might be possible to obtain security proofs for [12] that would work in class groups, under some number-theoretic assumptions, thus eliminating the need for a trusted setup. The philosophy of [12] is close to our construction: it consists in solving the puzzle of [14] (for a timing parameter Δ), and computing a proof of correctness. Their proofs can be computed with $O(\sqrt{\Delta}\log(\Delta))$ group multiplications, with a storage requirement of $O(\sqrt{\Delta})$ group elements. 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 [14], the security parameter is $k = 100$, the group G is an RSA group for a 2048 bit modulus, and the time Δ is set to 2^τ sequential squarings in the group, for $\tau = 40$. The corresponding proofs are $10KB$ long, and the verification requires around $10K$ multiplications in G. In comparison, in the same setting, our proofs are $0.25KB$ long, and the verification requires around 260 multiplications in G.

1.2 Notation

Throughout, the integer k denotes a security level (typically $128, 192$, or 256), and the map $H : \{\mathtt{0},\mathtt{1}\}^* \to \{\mathtt{0},\mathtt{1}\}^{2k}$ denotes a secure cryptographic hash function. For simplicity of exposition, the function H is regarded as a map from $\mathcal{A}^*$ to $\{\mathtt{0},\mathtt{1}\}^{2k}$, where $\mathcal{A}^*$ is the set of strings over some alphabet $\mathcal{A}$ such that $\{\mathtt{0},\mathtt{1}\} \subset \mathcal{A}$. 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 $\mathtt{int} : \{\mathtt{0},\mathtt{1}\}^* \to \mathbf{Z}_{\geq 0}$ maps $x \in \{\mathtt{0},\mathtt{1}\}^*$ in the canonical manner to the non-negative integer with binary representation x, and $\mathtt{bin} : \mathbf{Z}_{\geq 0} \to \{\mathtt{0},\mathtt{1}\}^*$ maps any non-zero integer to its binary representation with no leading $\mathtt{0}$-characters, and $\mathtt{bin}(0) = \mathtt{0}$.

2 Trapdoor verifiable delay functions

Let $\Delta : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ 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 Section 3 (essentially, it measures an amount of sequential work). A party, Alice, has a public key pk and a secret key sk. Let m be a piece of data. Alice, thanks to her secret key sk, is able to quickly evaluate a function $\mathsf{trapdoor}_{\mathsf{sk}}$ on m. On the other hand, other parties knowing only pk can compute $\mathsf{eval}_{\mathsf{pk}}(m)$ in time Δ, but not faster (and important parallel computing power does not give a substantial advantage in going faster; remember that Δ measures the sequential work), such that the resulting value $\mathsf{eval}_{\mathsf{pk}}(m)$ is the same as $\mathsf{trapdoor}_{\mathsf{sk}}(m)$.

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

$\mathsf{keygen} \to (\mathsf{pk}, \mathsf{sk})$ 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.

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

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

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

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 data m and time parameter Δ, let $(\mathsf{h}, \mathsf{p}) \leftarrow \mathsf{eval}_{\mathsf{pk}}(m, \Delta)$ and $(\mathsf{h}', \mathsf{p}') \leftarrow \mathsf{trapdoor}_{\mathsf{sk}}(m, \Delta)$. Then, $\mathsf{h} = \mathsf{h}'$ and the procedures $\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{h}, \mathsf{p}, \Delta)$ and $\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{h}', \mathsf{p}', \Delta)$ both output true.

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

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* m*, a value* h' *and a proof* p' *such that* $\mathsf{h}' \neq \mathsf{eval}_{\mathsf{pk}}(m, \Delta)$*, and* $\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{h}', \mathsf{p}', \Delta) = \mathsf{true}$*.*

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. 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 different single-core general purpose CPUs, the variations in running time between them will be reasonably small as overclocking records on clock-speeds barely achieve 9GHz (cf. [9]), only a small factor higher than a common personal computer. Then, parallelization has to be taken into consideration. Some parallelizable algorithms can run significantly faster on multiple parallel cores, up to a threshold where additional cores do not improve the running time anymore. Then, specialized hardware can be built to run an algorithm much more efficiently than any general purpose hardware.

Therefore a precise notion of wall-clock time is difficult to capture formally. However, for most applications, a good enough approximation is sufficient. Such an approximation can be obtain based on the choice of 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 time-lock assumption of Rivest, Shamir and Wagner [14] can be expressed as Assumption 1 below.

Definition 2 ((δ, t)-time-lock 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)-time-lock 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 $x \in \mathbf{Z}/N\mathbf{Z}$ is generated uniformly at random;*
4. *$\mathcal{B}(x)$ outputs $y \in \mathbf{Z}/N\mathbf{Z}$.*

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

Assumption 1 (Time-lock assumption) *There is a cost function $\delta : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ 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 $x \in \mathbf{Z}/N\mathbf{Z}$, the output of $\mathcal{S}(N, x)$ is the square of x, and $T(\mathcal{S}, (N, x)) < \delta(k)$;*
2. *For any $t \in \mathbf{Z}_{>0}$, no algorithm $\mathcal{A}$ of polynomial cost*[1] *wins the (δ, t)-time-lock 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 $x^{2^t} \mod N$ 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 $m \in \mathcal{M}$ is generated according to some random distribution of min-entropy at least k;*

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

4. $\mathcal{B}^{\mathcal{O}}(m)$ *outputs a value* h*, where* $\mathcal{O}$ *is an oracle that outputs the evaluation* $\mathsf{trapdoor}_{\mathsf{sk}}(m', \Delta)$ *on any input* $m' \neq m$*.*

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

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 m', the procedure $\mathsf{eval}_{\mathsf{pk}}(m', \Delta)$ produces the same output as $\mathsf{trapdoor}_{\mathsf{sk}}(m', \Delta)$, so any such request can be computed by the adversary in time $O(\Delta)$.

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

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

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.

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 forbiding low entropy is important: if m has a good chance of falling in a small subset of $\mathcal{M}$, the adversary can simply precompute the VDF for all the elements of this subset.

4 Construction of a verifiable delay function

Let $m \in \mathcal{A}^*$ be the message at which the VDF is to be evaluated. Alice's secret key sk is the order of a 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 $\mathtt{bin}(g)$. Also part of Alice's public key is a hash function $H_G : \mathcal{A}^* \to G$.

Remark 3 (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 $H_G(m) = \mathtt{int}(H(\text{“}\mathtt{residue}\text{”}||m)) \mod N$. For a technical reason explained later in Remark 5, we actually need to work in $(\mathbf{Z}/N\mathbf{Z})^\times/\{\pm 1\}$, and we call this the *RSA setup.*

Remark 4 (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. The construction is simple. 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 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 be computed if the factorisation of d is known. Therefore one should choose d to be a negative prime, which ensures that $|\mathrm{Cl}(d)|$ is odd. See [7] for a review of the arithmetic in class groups of imaginary quadratic orders, and a discussion on the choice of cryptographic parameters.

Given any string s, we denote by $H_{\mathtt{prime}}(s)$ the first odd prime number in the sequence $H_k(\text{“}\mathtt{prime}\text{”}||\mathtt{bin}(j)|||s)$, for $j \in \mathbf{Z}_{\geq 0}$, where H_k are the first k bits of H (recall that k is the security parameter). Consider a targeted evaluation time given by $\Delta = \tau\delta$ for a timing parameter τ, 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 at m, first let $h = H_G(m)$. The basic idea (which finds its origins in [14]) is that for any $\tau \in \mathbf{Z}_{>0}$, Alice can efficiently compute h^{2^τ} with two exponentiations, by first computing $x = 2^\tau \mod |G|$, followed by h^x. The running time is logarithmic in τ. Any other party who does not know $|G|$ can also compute h^{2^τ} by performing τ sequential squarings, with a running time $\tau\delta$. Therefore anyone can compute h^{2^τ} but only Alice can do it fast, and any other party has to spend a time linear in τ. However, verifying that the published value is indeed h^{2^τ} is long: there is no shortcut to the obvious strategy consisting in recomputing h^{2^τ} and checking if it matches. To solve this issue, we propose the following:

1. First compute $a = h^{2^\tau}$.
2. Compute the prime number $B = H_{\mathtt{prime}}(\mathtt{bin}(h)|||\mathtt{bin}(a))$.
3. Then compute $b = h^{\lfloor \frac{2^\tau}{B} \rfloor}$.
4. The output of the VDF is $(\mathsf{h}, \mathsf{p}) = (a, b)$.

Now, it might not be clear how Alice or a third party should compute $b = h^{\lfloor \frac{2^\tau}{B} \rfloor}$. For Alice, it is simple: she can compute $r = 2^\tau \mod B$. Then we have $\lfloor \frac{2^\tau}{B} \rfloor = \frac{2^\tau - r}{B}$, and since she knows the order of the group, she can compute $q = (2^\tau - r)B^{-1} \mod |G|$ and $b = h^q$. We explain in Section 4.1 how to compute b without knowing $|G|$. The procedures trapdoor, verify and eval are fully described in Algorithms 1, 2 and 3 respectively.

Verification. The verification consists in checking that $b^B h^r = a$, where B is the prime $H_{\mathtt{prime}}(\mathtt{bin}(h)|||\mathtt{bin}(a))$, and r is the remainder of 2^τ divided by B. It is straightforward to check that this holds if the evaluator is honest.

Now, what can a dishonest evaluator do? That question is answered formally in Section 6, but the intuitive idea is easy to understand. We will show that given m, finding a pair (a, b) 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.

Suppose that instead of setting $B = H_{\mathtt{prime}}(\mathtt{bin}(h)|||\mathtt{bin}(a))$, we consider a protocol where the evaluator first computes a, then a challenge B is received, then b is computed according to this B (the non-interactive version is then a Fiat-Shamir transformation). Suppose that the proof passes the verification, i.e., $b^B h^r = a$, where r is the least residue of 2^τ modulo B. Since $r = 2^\tau - B\lfloor \frac{2^\tau}{B} \rfloor$, the verification condition is equivalent to

$$ah^{-2^\tau} = \left(bh^{-\lfloor \frac{2^\tau}{B} \rfloor}\right)^B.$$

Before the generation of B, the left-hand side $\alpha = ah^{-2^\tau}$ is already determined. Once B is revealed, the evaluator is able to compute $\beta = bh^{-\lfloor \frac{2^\tau}{B} \rfloor}$, which is a B-th root of α. For an

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

Algorithm 1: $\mathsf{trapdoor}_{\mathsf{sk}}(m, \tau) \to (\mathsf{h}, \mathsf{p})$

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

Algorithm 2: $\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{h}, \mathsf{p}, \tau) \to$ true or false

evaluator to succeed with good probability, he must be able to extract B-th roots of α for arbitrary values of B. This is hard in our groups of interest, unless $\alpha = \beta = 1_G$, in which case (a, b) is the honest output.

Remark 5. Observe that in the RSA setup, this task is easy if $\alpha = \pm 1$, i.e. $a = \pm h^{2^\tau}$. It is however a difficult problem, given an RSA modulus N, to find an element $\alpha \mod N$ other than ± 1 from which B-th roots can be extracted for any B. 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 formalized (and generalised to other groups) in Definition 6.

4.1 Evaluation in total time $O(\tau)$

This section shows that the procedure $\mathsf{eval}_{\mathsf{pk}}(m, \tau)$ described in Algorithm 3 is correct and runs in total time $O(\tau)$. The asymptotic estimates in this section are given in terms of the time parameter τ, for a fixed security parameter k.

As observed in the previous section, since $B^{-1} \mod |G|$ cannot be computed without knowledge of the private key, an alternative strategy is needed for the evaluation without the trapdoor. Assuming the correctness of the function $\mathsf{eval\_rec}(h, B, i)$ (described in Algorithm 4), which computes h^q where q is the quotient of the euclidean division of 2^τ by B, it is easy to check that the function $\mathsf{eval}_{\mathsf{pk}}(m, \tau)$ is correct.

The function $\mathsf{eval\_rec}(h, B, \tau)$ deserves more explanations. A straightforward strategy would be to actually compute the quotient q of 2^τ by B and then the exponentiation h^q.

```
Data: a public key $\mathsf{pk} = (G, H_G)$, some data $m \in \mathcal{A}^*$, a targeted evaluation time $\Delta = \tau\delta$.
Result: the output value $\mathsf{h}$ and a proof $\mathsf{p}$.
$h \leftarrow H_G(m) \in G$;
$a \leftarrow h^{2^\tau}$;                                        // via $\tau$ sequential squarings
$B \leftarrow H_{\texttt{prime}}(\texttt{bin}(h)|||\texttt{bin}(a))$;
$b \leftarrow \mathsf{eval\_rec}(h, B, \tau)$ ;                   // this function is described in Algorithm 4
$(\mathsf{h}, \mathsf{p}) \leftarrow (a, b)$;
return $(\mathsf{h}, \mathsf{p})$;
```

Algorithm 3: $\mathsf{eval}_{\mathsf{pk}}(m, \tau) \to (\mathsf{h}, \mathsf{p})$

```
Data: an element $h$ in a group $G$ (with identity $1_G$), a prime number $B$ and a positive integer $\tau$. The
      integer MEMORY_BOUND is a limit on the storage requirement (it does not need to be large for the
      algorithm to perform efficiently, see analysis of Section 4.1).
Result: $h^q$, where $q$ is the quotient of the euclidean division of $2^\tau$ by $B$.
if $2^\tau < B$ then
    return $1_G$;
else if $\tau <$ MEMORY_BOUND then
    $q \leftarrow$ quotient of the euclidean division of $2^\tau$ by $B$ ;    // using Newton-Raphson
    return $h^q$;
else if $\tau$ is even then
    $\tau' \leftarrow \tau/2$;
    $x \leftarrow \mathsf{eval\_rec}(h, B, \tau')$;
    $y \leftarrow \mathsf{eval\_rec}(x, B, \tau')$;
    $r' \leftarrow$ least residue of $2^{\tau'}$ modulo $B$;
    $s \leftarrow$ quotient of the euclidean division of $r'^2$ by $B$;
    return $y^B x^{2r'} h^s$;
else
    $\tau' \leftarrow (\tau - 1)/2$;
    $x \leftarrow \mathsf{eval\_rec}(h, B, \tau')$;
    $y \leftarrow \mathsf{eval\_rec}(x, B, \tau')$;
    $r' \leftarrow$ least residue of $2^{\tau'}$ modulo $B$;
    $s \leftarrow$ quotient of the euclidean division of $2r'^2$ by $B$;
    return $y^{2B} x^{4r'} h^s$;
end
```

Algorithm 4: $\mathsf{eval\_rec}(h, B, \tau) \to h^q$

This could be done in time essentially linear in τ with a Newton-Raphson division, but it requires as much memory. It might not be a problem for most platforms if the time parameter is not excessively large. It is however possible to use a logarithmic amount of memory, in about the same amount of parallel time with three cores. To do so, we proceed by recursion on τ. The base case is easy: if 2^τ is smaller than B, then q is zero, so h^q is the identity element of the group. In particular, if $\tau = 0$, then $2^\tau = 1 < B$.

For the general step of the recursion, first assume that τ is even, and let q' be the quotient of the euclidean division of $2^{\tau/2}$ by B, and r' the remainder. Observe that

$$2^\tau = \left(2^{\tau/2}\right)^2 = (q'B + r')^2 = ({q'}^2 B + 2q'r')B + r'^2.$$

Therefore, if s is the quotient of the euclidean division of r'^2 by B, then $q = {q'}^2 B + 2q'r' + s$. We can recursively compute $h^{q'} = \mathsf{eval\_rec}(h, B, \tau/2)$ and $h^{{q'}^2} = \mathsf{eval\_rec}(h^{q'}, B, \tau/2)$. Then

h^q can be computed as

$$h^q = \left(h^{q'^2}\right)^B \left(h^{q'}\right)^{2r'} h^s.$$

The case where τ is odd is similar, except that we use the relation $2^\tau = 2\left(2^{(\tau-1)/2}\right)^2$. Besides the recursive calls, the most expensive steps are the three exponentiations $y^B, x^{2r'}$ and h^s. The processing time of a single step of the recursion is essentially $c = 3 \cdot \log_2(B)\delta$ (assuming that multiplying and squaring in G have similar cost δ).

Let $R(\tau)$ denote the running time of $\mathsf{eval\_rec}(h, B, \tau)$. Observe that it takes $\lfloor \log_2(\tau) - \log_2 \log_2(B) \rfloor$ successive halvings of τ to reach a value τ' smaller than $\log_2(B)$. Therefore,

$$R(\tau) \leq 2^{\lfloor \log_2(\tau) - \log_2 \log_2(B) \rfloor} R(\tau') + \left(2^{\lfloor \log_2(\tau) - \log_2 \log_2(B) \rfloor} - 1\right) c \leq 3\tau\delta.$$

This factor three can be eliminated by running the three exponentiations on three parallel cores. This estimation is experimentally confirmed by our implementation. Now, the need for parallel cores can be eliminated by setting the bound `MEMORY_BOUND` to, for instance, $40\log_2(B)$. Indeed, the bulk of the computation are the large number of calls of eval_rec for small values of τ, which can be done sequentially with a Newton-Raphson division followed by an exponentiation. This change preserves the wall-clock running time, but with a single core, and a small memory requirement. This, again, is confirmed by our implementation.

5 Analysis of the sequentiality

In this section, the proposed construction is proven to be $(\tau\delta)$-sequential, meaning that no polynomially bounded player can win the associated $(\tau\delta)$-evaluation race game with non-negligible probability (in other words, the VDF cannot be evaluated in time less than $\tau\delta$). For the RSA setup, it is proved under the classic time-lock assumption of Rivest, Shamir and Wagner [14] (formalised in Assumption 1), and more generally, it is secure for groups where a generalisation of this assumption holds (Assumption 2).

5.1 Generalised time-lock assumptions

The following game generalises the classic time-lock assumption to arbitrary families of groups of unknown orders.

Definition 5 (Generalised (δ, t)-time-lock game). *Consider a sequence $(\mathcal{G}_k)_{k \in \mathbf{Z}_{>0}}$, where each $\mathcal{G}_k$ is a set of finite groups (supposedly of unknown orders), associated to 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 $\delta : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ is a function. The (δ, t)-*time-lock game *goes as follows:*

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

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

Assumption 2 (Generalised time-lock assumption) *The generalised time-lock assumption for a given family of groups $(\mathcal{G}_k)_{k\in\mathbf{Z}_{>0}}$ is the following. There is a cost function $\delta : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ such that the following two statements hold:*

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 $x \in G$, the output of $\mathcal{S}(G, x)$ is the square of x, and $T(\mathcal{S}, (G, x)) < \delta(k)$;*
2. *For any $t \in \mathbf{Z}_{>0}$, no algorithm $\mathcal{A}$ of polynomial cost wins the (δ, t)-time-lock 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 x^{2^t} than performing t sequential squarings.

5.2 Sequentiality in the random oracle model

Proposition 1 ($\tau\delta$-sequentiality of the trapdoor VDF in the random oracle model). *Let $\mathcal{A}$ be a player winning with probability p_{win} the $(\tau\delta)$-evaluation race game associated to the proposed construction, assuming H_G and $H_{\mathtt{prime}}$ are random oracles and $\mathcal{A}$ is limited to q oracle queries*[2]*. Then, there is a player $\mathcal{C}$ for the (generalised) (δ, τ)-time-lock 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*[3]*).*

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 and μ respectively). When $\mathcal{A}(G)$ outputs an algorithm $\mathcal{B}$, $\mathcal{C}$ generates a random $m \in \mathcal{M}$ (according to the same distribution as the $(\tau\delta)$-evaluation race game). If m 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 $(\tau\delta)$-evaluation race game is at least k. Otherwise, $\mathcal{C}$ outputs the following algorithm $\mathcal{B}'$. When receiving as input the challenge x, $\mathcal{B}'$ adds x to the table of oracle H_G, for the input m (i.e., $H_G(m) = x$). As discussed in Remark 1, we can assume that the algorithm $\mathcal{B}$ does not call the oracle $\mathsf{trapdoor}_{\mathsf{sk}}(-, \mathsf{h}, \Delta)$. Then $\mathcal{B}'$ can invoke $\mathcal{B}$ on input m while simulating the oracles H_G and $H_{\mathtt{prime}}$. Whenever $\mathcal{B}$ outputs h, $\mathcal{B}'$ outputs h, which equals x^{2^τ} whenever h is the correct evaluation of the VDF at m. We assume that simulating the oracle has a negligible cost, so $\mathcal{B}'(x)$ has essentially the same time-cost as $\mathcal{B}(m)$. Then, $\mathcal{C}$ wins the (δ, τ)-time-lock game with probability $p \geq p_{\mathsf{win}}(1 - q/2^k)$. □

6 Analysis of the soundness

In this section, the proposed construction is proven to be sound, meaning that no polynomially bounded player can product 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 B-th roots modulo an RSA modulus N can be extracted

[2] 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.

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

for arbitrary B's following a distribution μ, when the factorisation of N is unknown. More generally, the construction is sound if a generalisation of this assumptions holds in the group of interest.

6.1 The root finding problem

The following game formalises the root finding problem.

Definition 6 (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 wish is given to $\mathcal{A}$ (G is supposedly of unknown order). The player $\mathcal{A}$ then outputs an element u of G. An integer B is generated according to the distribution μ and given to $\mathcal{A}$. The player $\mathcal{A}$ outputs an integer v and wins the game if $v^B = u \neq 1_G$.*

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 $L_N(1/3)$. It is however definitely closely related, and seems as difficult when μ is the uniform distribution over the primes in $(0, 2^{2k})$. Recall that k is the security parameter, which is implicitly passed as a parameter to the procedure keygen.

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.

We now prove that to win this game $\mathcal{G}^{\mathsf{root}}$, it is sufficient to win the following game $\mathcal{G}^{\mathsf{root}}_X$, which is more convenient for our analysis of the soundess.

Definition 7 (The oracle root finding game $\mathcal{G}^{\mathsf{root}}_X$). *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 $\mathcal{O} : \mathcal{A}^* \to \mathbf{Z}_{>0}$ be a random oracle with distribution μ. The player has access to the random oracle $\mathcal{O}$. The* oracle root finding game $\mathcal{G}^{\mathsf{root}}_X(\mathcal{A}, \mathcal{O})$ *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 $\mathcal{G}^{\mathsf{root}}_X(\mathcal{A}, \mathcal{O})$ 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 $i \in \{1, 2, ..., q + 1\}$ is chosen uniformly at random. For the first $i - 1$ (distinct) queries from $\mathcal{A}'$ to $\mathcal{O}$, the oracle value is chosen at random according to distribution μ. 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 B is received, following the distribution μ. This B 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 integers with distribution μ. 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 distribution μ, $\mathcal{A}'$ 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 $v^B = u \neq 1_G$, where $u = X(G, s)$ is the output for the first round of $\mathcal{G}^{\mathsf{root}}$, and $\mathcal{O}(s) = B$ 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 $\mathcal{Q} = \{s_1, s_2, ..., s_{q+1}\}$ 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 $\{1, 2, ..., q+1\}$, and the values given to $\mathcal{A}'$ are independent from i (the oracle values are all independent random variables with distribution μ). So $\Pr\left[i = j | \mathsf{win}_{\mathcal{A}'}\right] = 1/(q+1)$. Therefore $\Pr[\mathcal{B} \text{ wins } \mathcal{G}^{\mathsf{root}}] \geq p_{\mathsf{win}}/(q+1)$. 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. □

6.2 Soundness in the random oracle model

Proposition 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 to the proposed scheme, assuming H_G and $H_{\mathtt{prime}}$ are random oracles and $\mathcal{A}$ is limited to q oracle queries*[4]*. 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 $\mathcal{D}$, 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 $\mathtt{bin}(g)$. For any such group G, any elements $x, y \in G$, let

$$X(G, \mathtt{bin}(x)|||\mathtt{bin}(y)) = y/x^{2^\tau},$$

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_G is simulated by generating random values in the straightforward way, and $H_{\mathtt{prime}}$ is set to be exactly the oracle $\mathcal{O}$. The algorithm $\mathcal{A}$ outputs a message m, and pair $\mathsf{h} = (a, b) \in G \times G$ (if it is not of this form, abort). Output $s = \mathtt{bin}(h)|||\mathtt{bin}(a)$ and $v = b/h^{\lfloor \frac{2^\tau}{\mathcal{O}(s)} \rfloor}$. 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. 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 $p \geq p_{\mathsf{win}}(1-\varepsilon)/(q+1)$. □

[4] 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.

Remark 6. The construction remains sound if instead of $\mathsf{h} = a$ and $\mathsf{p} = b$, we consider $\mathsf{h} = (a, b)$ and p is the empty proof. The winning probability of $\mathcal{D}$ in Proposition 2 becomes $p \geq p_{\mathsf{win}}(1-\varepsilon)/(q+1)$, where $\varepsilon = \mathrm{negl}\left(\frac{k}{\log\log(|G|)\log(q)}\right)$, by accounting for the unlikely event that $\mathcal{O}(s)$ is a divisor of $|G|$.

Acknowledgements

The author wishes to thank Serge Vaudenay, Alexandre Gélin, Arjen K. Lenstra and Novak Kaluderovic for interesting discussions about the present work.

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. *Advances in Cryptology — CRYPTO' 18*, 2018.
5. D. Boneh and M. Franklin. Efficient generation of shared rsa keys. In *Annual International Cryptology Conference*, pages 425–439. Springer, 1997.
6. 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.
7. 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.
8. J. Buchmann and H. C. Williams. A key-exchange system based on imaginary quadratic fields. *Journal of Cryptology*, 1(2):107–118, 1988.
9. CPU-Z OC world records. http://valid.canardpc.com/records.php, 2015.
10. 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.
11. A. K. Lenstra and B. Wesolowski. Trustworthy public randomness with sloth, unicorn and trx. *International Journal of Applied Cryptology*, 2016.
12. K. Pietrzak. Simple verifiable delay functions. Cryptology ePrint Archive, Report 2018/627, Version 20180626:145529, 2018. https://eprint.iacr.org/2018/627.
13. M. O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256 – 267, 1983.
14. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.
15. T. Sander. Efficient accumulators without trapdoor extended abstract. In *International Conference on Information and Communications Security*, pages 252–262. Springer, 1999.
16. U. Vollmer. Asymptotically fast discrete logarithms in quadratic number fields. In *International Algorithmic Number Theory Symposium (ANTS)*, pages 581–594. Springer, 2000.

A Proof of Remark 1

Model H as a random oracle. Suppose that

$$\begin{aligned}\mathsf{trapdoor}^{H}_{\mathsf{sk}}(m, \varDelta) &= t_{\mathsf{sk}}(H(m), \varDelta),\\ \mathsf{eval}^{H}_{\mathsf{pk}}(m, \varDelta) &= e_{\mathsf{pk}}(H(m), \varDelta), and\\ \mathsf{verify}_{\mathsf{pk}}(m, \mathsf{h}, \varDelta) &= v_{\mathsf{pk}}(H(m), \mathsf{h}, \varDelta),\end{aligned}$$

for procedures t, e and v that do not have access to H.

Let $\mathcal{A}$ be a player of the $\mathit{\Delta}$-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 $\mathit{\Delta}$-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 $t_{\mathsf{sk}}(H'(m), \mathit{\Delta})$.

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 $(H(x), \mathsf{eval}_{\mathsf{pk}}(x, \mathit{\Delta}))$ in a list L. In addition, it computes and stores $(Y_i, e_{\mathsf{pk}}(Y_i, \mathit{\Delta}))$ for each $i = 1, \dots, q$, 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 $\mathit{\Delta}$-evaluation race game with same probability as $\mathcal{A}$ even when its output $\mathcal{B}'$ is not given access to a evaluation oracle.