Slow-timed hash functions

Benjamin Wesolowski

École Polytechnique Fédérale de Lausanne, EPFL IC LACAL, Switzerland

Abstract. We construct an efficiently verifiable slow-timed hash function. A slow-timed hash function is a hash function with the guarantee that its evaluation requires to run a given number of sequential steps, but the result can be efficiently verified. They have applications in decentralised systems, such as the generation of trustworthy public randomness in a trustless environment. To construct a slow-timed hash function, we actually build a *trapdoor slow-timed hash function*. A trapdoor slow-timed hash function is essentially a slow-timed hash function 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), we obtain a simple slow-timed hash function.

1 Introduction

We describe a hash function that is slow to compute and easy to verify: a *slow-timed hash function* in the sense of [12]. 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 cores), and the result should be easy to verify (i.e., for a cost polylog(Δ)). These special hash functions are used in [12] 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 slow-timed hash proposed in [12], *sloth*, is not asymptotically efficiently verifiable: the verification procedure (given x and y, verify that $\texttt{sloth}(x) = y$) is faster than the evaluation procedure (given x, compute the value $\texttt{sloth}(x)$) only by a linear factor. The new construction provides an exponentially faster verification.

Independently from the present work, the paper [4] was recently published, where the authors describe *verifiable delay functions* (VDF). This

notion essentially coincides with the notion of slow-timed hash. In addition to compiling a variety of interesting applications of such functions in decentralised systems, 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 polylog(Δ) of parallelism to run in parallel time Δ. 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 number of cores to run in optimal time Δ.

Trapdoor slow-timed hash function. To construct a slow-timed hash function, we first construct a *trapdoor* slow-timed hash function. A party, Alice, holds a secret key sk (the trapdoor), and an associated public key pk. Given a piece of data m, a trapdoor slow-timed hash allows to compute a hash h of 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 hash of m should be efficient, for a cost polylog(Δ).

We propose a practical construction based on groups G of unknown order (such at 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 slow-timed hash function. Suppose that a public key pk for a trapdoor slow-timed hash function is given without any known associated private key. This results in a simple slow-timed hash function, 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 provide, 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. 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 [11] and [16].

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 [12] and was used to provide trust to the generation of public random numbers. These slow-timed hash functions were recently revisited and formalised in the paper [4] under the name of verifiable delay functions.

1.2 Notation

Throughout, the integer k denotes a security level (typically $128, 192$, or 256), and $H : \{0,1\}^* \to \{0,1\}^{2k}$ 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 $\{\mathtt{0},\mathtt{1}\} \subset \mathcal{A}$. 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 slow-timed hash 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 $\mathsf{trapdoor}_{\mathsf{sk}}(m)$. We call this output the *timed-hash* of m (with respect to the key pk).

More formally, a trapdoor slow-timed hash function consists of the following components:

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

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

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

$\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{h}, \Delta) \to \mathsf{true}$ **or** false is a procedure to check if h is indeed the timed-hash of m, associated to the public key pk and the evaluation time Δ.

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, we have $\mathsf{trapdoor}_{\mathsf{sk}}(m, \Delta) = \mathsf{eval}_{\mathsf{pk}}(m, \Delta)$, and if $\mathsf{h} = \mathsf{eval}_{\mathsf{pk}}(m, \Delta)$ then $\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{h}, \Delta)$ outputs true.

We also require the protocol to be *sound*. Intuitively, we want that if $\mathsf{h}' \neq \mathsf{eval}_{\mathsf{pk}}(m, \Delta)$ 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 slow-timed hash function is* sound *if any polynomially bounded algorithm solves the following task with negligible probability: given as input the public key* pk*, output a message* m *and a value* h' *such that* $\mathsf{h}' \neq \mathsf{eval}_{\mathsf{pk}}(m, \Delta)$ *and* $\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{h}', \Delta) = true$*.*

The required 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 slow-timed hash function is Δ-*secure* if any polynomially bounded adversary wins the Δ-evaluation race game with negligible probability.

3 Wall-clock time and computational assumptions

Primitives such as slow-timed hash 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. [10]), 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 factorization 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 trapdoor slow-timed hash functions.

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

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

Definition 4 (Δ-security). *A trapdoor slow-timed hash function is* Δ-secure *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 hashes 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 timed-hashed (as is the case in the construction we present in the next section), i.e.

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

[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.

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 timed-hashes of all the elements of this subset.

4 Construction of a slow-timed hash function

Let $m \in \mathcal{A}^*$ be the message to be timed-hashed. 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 $H(\text{“}\mathtt{prime}\text{”}||\mathtt{bin}(j)||s)$, for $j \in \mathbf{Z}_{\geq 0}$. Consider a targeted

evaluation time given by $\Delta = 2^\tau \delta$ 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 compute the timed-hash of m, first let $h = H_G(m)$. The basic idea (which finds its origins in [14]) is that for any $t \in \mathbf{Z}_{>0}$, Alice can efficiently compute h^{2^t} with two exponentiations, by first computing $x = 2^t \mod |G|$, followed by h^x. The running time is logarithmic in t. Any other party who does not know $|G|$ can also compute h^{2^t} by performing t sequential squarings, with a running time linear in t. Therefore anyone can compute h^{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 is indeed h^{2^t} is long: there is no shortcut to the obvious strategy consisting in recomputing h^{2^t} and checking if it matches.

A first solution to this issue is discussed in [6], in the RSA setup. Let $t = 2^\tau$, and rather than just publishing $h^{2^{2^\tau}}$, publish the sequence of τ elements

$$(b_1, b_2, b_3, ..., b_\tau) = \left(h^2, h^4, h^{16}, ..., h^{2^{2^\tau}}\right),$$

and prove via a zero-knowledge protocol that each triple (h, b_i, b_{i+1}) is of the form (h, h^x, h^{x^2}) for some integer x. The protocol the authors describe is inspired from the classic zero-knowledge proof that (g, A, B, C) is a Diffie-Hellman tuple [9]. The modified version in [6] is not perfectly zero-knowledge with respect to Alice's secret $G = \phi(N)$. Indeed, to prove that the scheme of [6] is zero-knowledge, the authors refer to the result of [9]. But [9] proves the zero-knowledge property with respect to the exponent, not to the order of the group (the exponents are secret in the Diffie-Hellman setting, and the group order is not), whereas in [6] (as well as in the present situation) the order of the group is secret and the exponent is not. In particular, the protocol of [6] is not perfectly simulatable without knowledge of the group order.

Furthermore, the evaluation of the hash function cannot be interactive; unfortunately there is no obvious way to make the suggested protocol non-interactive while allowing evaluation without the secret key. Therefore the following introduces a new approach. The procedures trapdoor, verify and eval are fully described in Algorithms 1, 2 and 3 respectively, and explained in detail below.

4.1 The basic construction

The timed-hash value on input m is the tuple

$$(h_1, h_2, ..., h_\tau, P_1, ..., P_{\tau-1}),$$

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 $\Delta = 2^\tau \delta$.
Result: the timed-hash h.
$h \leftarrow H_G(m) \in G$;
for $i = 1, ..., \tau$ **do**
  $y_i \leftarrow 2^{2^i - 2} \mod |G|$;
  $h_i \leftarrow h^{y_i}$;
end
for $i = 1, ..., \tau - 1$ **do**
  $B \leftarrow H_{\texttt{prime}}(\texttt{bin}(i)|||\texttt{bin}(h_i)|||\texttt{bin}(i+1))$;
  $x \leftarrow 2y_i$;
  $r \leftarrow$ least residue of $2^{2^i - 1}$ modulo B;
  $\beta \leftarrow (x - r)B^{-1} \mod |G|$; `// since` B `is a large random prime, it is invertible modulo` $|G|$ `with overwhelming probability`
  $P_i \leftarrow (h^\beta, h^{x\beta})$;
end
return $\mathsf{h} = (h_1, h_2, ..., h_\tau, P_1, ..., P_{\tau-1})$;

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

where $h_i = h^{2^{2^i - 2}} \in G$ for $i = 1, ..., \tau$ (where $h = H_G(m)$), and P_i is a (non-interactive) proof that (h, h_i^2, h_{i+1}) is a triple of the form (h, h^x, h^{x^2}). The algorithm is synthesised in Algorithm 1. The construction of the proofs P_i will be discussed in the following Section 4.2.

The verifier can check that $h_\tau = h^{2^{2^\tau - 2}}$ as follows: first check that $h_1 = h = H_G(m)$; then for each $i = 1, ..., \tau - 1$, check the proof P_i. If $h_i = h^{2^{2^i - 2}}$ and P_i is correct, then (h, h_i^2, h_{i+1}) is of the form (h, h^x, h^{x^2}), and

$$h_{i+1} = h^{\left(2 \cdot 2^{2^i - 2}\right)^2} = h^{2^{2^{i+1} - 2}}.$$

Therefore, by induction, if all the proofs are correct, we indeed have that $h_\tau = h^{2^{2^\tau - 2}}$. The verification procedure is synthesised in Algorithm 2.

4.2 The proofs P_i

The following focuses on the problem of proving efficiently that (h, b_1, b_2) is of the form (h, h^x, h^{x^2}), without revealing any information about $|G|$. The exponent x is not meant to be secret: everybody knows it is congruent to some $2^{2^i - 1}$ (yet any other representation of that integer modulo $|G|$ should not be leaked: that would allow to compute a multiple of $|G|$). But checking the form of (h, b_1, b_2) simply by exponentiating by x and x^2 is inefficient. What will actually be produced is a tuple P which asserts that either P was produced by Alice, or (h, b_1, b_2) is of the form (h, h^x, h^{x^2}).

Data: a public key $\mathsf{pk} = (G, H_G)$, some data $m \in \mathcal{A}^*$, a targeted evaluation time $\Delta = 2^\tau \delta$, and a timed-hash value h.
Result: true if h is the correct hash of m, false otherwise.
$(h_1, h_2, ..., h_\tau, P_1, ..., P_{\tau-1}) \leftarrow \mathsf{h}$;
$h \leftarrow H_G(m) \in G$;
if $h_1 \neq h$ **then**
  return false;
end
for $i = 1, ..., \tau - 1$ **do**
  $B \leftarrow H_{\texttt{prime}}(\texttt{bin}(i)|||\texttt{bin}(h_i)|||\texttt{bin}(h_{i+1}))$;
  $r \leftarrow$ least residue of 2^{2^i-1} modulo B;
  $(c_1, c_2) \leftarrow P_i$;
  if $h_i^2 \neq c_1^B h^r$ *or* $h_{i+1} \neq c_2^B {h_i}^{2r}$ **then**
    return false;
  end
end
return true;

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

First, consider an interactive protocol. The verifier first receives h, b_1 and b_2, and then sends a (large) random odd prime number B to the prover. The prover then computes the least residue r of 2^{2^i-1} modulo B, and finally $c_1 = h^\beta$ and $c_2 = h^{x\beta}$ where[2] $\beta \equiv (x - r)B^{-1} \mod |G|$. The proof is the pair $P = (c_1, c_2)$. Such a proof can be forged by a party who does not know Alice's secret; this fact might be surprising at first given that such a party cannot compute $B^{-1} \mod |G|$. The procedure will be described later (in Section 4.3). Then, the verifier simply computes r and checks that $b_1 = c_1^B h^r$ and $b_2 = c_2^B b_1^r$. It is straightforward to check that this holds if the prover is honest.

Now, what can a dishonest prover do? That question is answered formally in Section 5, but the intuitive idea is easy to understand. Given x such that $b_1 = h^x$, only Alice can produce misleading proofs. Indeed, suppose that the proof passes the verification, i.e., $b_1 = c_1^B h^r$ and $b_2 = c_2^B b_1^r$, where r is the least residue of 2^{2^i-1} modulo B. Exponentiating the first equality by x yields $b_1^x = c_1^{xB} h^{xr}$, and the second can be written as $b_2 = c_2^B h^{xr}$. Therefore $b_1^x / b_2 = (c_1^x / c_2)^B$. When publishing b_1 and b_2, the element $\alpha = b_1^x / b_2$ is determined but the prover does not know already about B. Once B is revealed, the prover must be able to produce values c_1 and c_2 that will pass the tests with a good probability, which implies

[2] Note that this step requires B to be invertible modulo $|G|$. A malicious verifier could exploit this, but in the forthcoming construction, B is generated in a non-interactive way via a hash function, so is invertible with overwhelming probability.

Data: a public key $\mathsf{pk} = (G, H_G)$, some data $m \in \mathcal{A}^*$, a targeted evaluation time $\Delta = 2^\tau \delta$.
Result: the timed-hash value h.
$h \leftarrow H_G(m) \in G$;
$h_1 \leftarrow h$;
for $i = 1, ..., \tau - 1$ **do**
  $B \leftarrow H_{\texttt{prime}}(\texttt{bin}(i)|||\texttt{bin}(h_i)|||\texttt{bin}(h_{i+1}))$;
  $c_1 \leftarrow \mathsf{eval\_rec}(h, B, i)$; `// this function is described in Algorithm 4`
  $c_2 \leftarrow c_1^{2^{2^i-1}}$;
  $P_i \leftarrow (c_1, c_2)$;
  $h_{i+1} \leftarrow h_i^{2^{2^i}} = h^{2^{2^{i+1}-2}} \in G$;
end
return $\mathsf{h} = (h_1, h_2, ..., h_\tau, P_1, ..., P_{\tau-1})$;

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

Data: an element h in a group G (with identity 1_G), a prime number B and a positive integer i.
Result: h^β, where β is the quotient of the euclidean division of 2^{2^i-1} by B.
if $2^{2^i-1} < B$ **then**
  return 1_G;
else
  $x \leftarrow \mathsf{eval\_rec}(h, B, i-1)$;
  $y \leftarrow \mathsf{eval\_rec}(x, B, i-1)$;
  $\alpha \leftarrow 2^{i-1} - 1 \mod B - 1$;
  $r \leftarrow$ least residue of 2^α modulo B;
  $q \leftarrow$ quotient of the euclidean division of $2r^2$ by B;
  return $y^{2B} x^{4r} h^q$;
end

Algorithm 4: $\mathsf{eval\_rec}(h, B, i) \rightarrow h^\beta$

that c_1^x / c_2 is a B-th root of α. For a prover to cheat and succeed with good probability, he must be able to extract B-th roots of α for arbitrary values of B.

Remark 5. Observe that in the RSA setup, this task is easy if $\alpha = \pm 1$, i.e. $b_2 = \pm b_1^x = \pm h^{x^2}$. 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.

Recall that given a string s, $H_{\texttt{prime}}(s)$ denotes the first prime number in the sequence $H(\text{“}\texttt{prime}\text{”}||\texttt{bin}(j)|||s)$. The above protocol can be made

non-interactive by letting B be, for instance, the prime number

$$H_{\mathtt{prime}}(\mathtt{bin}(i)|||\mathtt{bin}(b_1)|||\mathtt{bin}(b_2)),$$

where i is the index of P_i. This simulates a choice of B uniform among the primes in $[0, 2^{2k})$. If H is considered to be a random function with uniform distribution, let μ denote the probability distribution of the output of $H_{\mathtt{prime}}(s)$.

4.3 Evalutation in total time $O(2^\tau)$

This section shows that the procedure $\mathsf{eval}_{\mathsf{pk}}(m, \tau)$ described in Algorithm 3 is correct and runs in total time $O(2^\tau)$. The asymptotic estimates in this section are given in terms of the time parameter τ, for a fixed security parameter k. Proving that there is no way to produce a valid output with less than $\Delta = 2^\tau \delta$ sequential operations is the matter of the security analysis in Section 5.

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^β where β is the quotient of the euclidean division of 2^{2^i-1} by B, it is easy to check that the function $\mathsf{eval}_{\mathsf{pk}}(m, \tau)$ is correct. The function $\mathsf{eval\_rec}(h, B, i)$ deserves more explanations. It operates by recursion on i. The base case is easy: if 2^{2^i-1} is smaller than B, then β is zero, so h^β is the identity element of the group. In particular, if $i = 1$, then $2^{2^i-1} = 1 < B$.

For the general step of the recursion, let β' be the quotient of the euclidean division of $2^{2^{i-1}-1}$ by B, and r the remainder. Observe that

$$2^{2^i-1} = 2\left(2^{2^{i-1}-1}\right)^2 = 2(\beta' B + r)^2 = (2\beta'^2 B + 4\beta' r)B + 2r^2.$$

Therefore, if q is the quotient of the euclidean division of $2r^2$ by B, then $\beta = 2\beta'^2 B + 4\beta' r + q$. We can recursively compute $h^{\beta'} = \mathsf{eval\_rec}(h, B, i-1)$ and $h^{\beta'^2} = \mathsf{eval\_rec}(h^{\beta'}, B, i-1)$. Then h^β can be computed as

$$h^\beta = \left(h^{\beta'^2}\right)^{2B} \left(h^{\beta'}\right)^{4r} h^q.$$

Besides the recursive calls, the most expensive step is the computation of the remainder of the euclidean division of $2^{2^{i-1}-2}$ by B. But all these remainders of 2^{2^j-2}, $j < i$, that will be used in the recursion, can be

computed at once at the beginning for a total time in $O(i)$: simply compute all the values $2^{2^j} \mod B$, for $j < i$ by successive squarings, multiply all of them by $2^{-2} \mod B$ (recall that B is an odd prime number), and store the results for later use. Therefore, leaving this operation aside as a preprocessing step, the processing time of a single step of the recursion is bounded above by a constant c (it actually depends on the bit-length of B, (almost) bounded by the bit-length of the output of the hash function H; it does not depend on the time parameter τ). If $R(i)$ denotes the running time of $\mathsf{eval\_rec}(h, B, i)$, then

$$R(i) \leq 2R(i-1) + c \leq 2^{i-1}R(1) + (2^{i-1} - 1)c \leq 2^i c.$$

This function is called in the evaluation algorithm for any $i < \tau$ (with different values of B). The total time of these calls is therefore in $O(2^\tau)$. It is then easy to see that the total time of the evaluation is also in $O(2^\tau)$.

5 Security analysis

In this section, the proposed construction is proven to be sound and $2^\tau\delta$-secure, meaning that no polynomially bounded player can win the associated $2^\tau\delta$-evaluation race game with non-negligible probability (in other words, the timed-hash cannot be computed in time less than $2^\tau\delta$). For the RSA setup, it is proved under the classic time-lock assumption of Rivest, Shamir and Wagner [14] (formalised in Assumption 1), and a new 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 for arbitrary B's following a distribution μ, when the factorisation of N is unknown. More generally, it is secure for groups where a generalisation of these assumptions hold.

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 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 $\mathcal{G}_X^{\mathsf{root}}$.

Definition 7 (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 $\mathcal{O} : \mathcal{A}^* \to \mathbf{Z}_{>0}$ be a random oracle with distribution μ. The player has access to the random oracle $\mathcal{O}$. The* oracle root finding game $\mathcal{G}_X^{\mathsf{root}}(\mathcal{A}, \mathcal{O})$ *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 $i \in \{1, 2, ..., q+1\}$ 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. □

5.3 Security in the random oracle model

Proposition 1 (Security of the trapdoor slow-timed hash function in the random oracle model). *Let $\mathcal{A}$ be a player winning with probability p_{win} the $(2^\tau\delta)$-evaluation race 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[3]. Then, there is a player $\mathcal{A}_1$ for the (generalised) $(\delta, 2^\tau)$-time-lock game, and $\mathcal{A}_2$ for the root finding game $\mathcal{G}^{\mathsf{root}}$, with respective winning probabilities p_1 and p_2 with $p_1/(1 - q/2^k) + (q+1)p_2 \geq p_{\mathsf{win}}$, and with same running time as $\mathcal{A}$ (up to a constant factor[4]).*

Proof. Let $\mathcal{A}$ be a player winning with probability p_{win} the Δ-evaluation race game. Let p'_{win} be the probability that $\mathcal{A}$ wins with an output $\mathsf{h} = (h_1, ..., h_\tau, P_1, ..., P_{\tau-1})$ where $h_1 = H_G(m)$ and $h_\tau = h_1^{2^{2^\tau}-2}$.

[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 $\mathcal{A}_1$ to win the $(\delta, 2^\tau)$-time-lock game, since it concerns only the running time of $\mathcal{A}_1$ itself and not of the algorithm output by $\mathcal{A}_1(G)$

Constructing $\mathcal{A}_1$. Build $\mathcal{A}_1$ as follows. Upon receiving $\mathsf{pk} = (G, H_G)$, $\mathcal{A}_1$ 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 $\mathcal{B}$, $\mathcal{A}_1$ generates a random $m \in \mathcal{M}$ according to distribution ν. If m has been queried by the oracle already, $\mathcal{A}_1$ aborts; this happens with probability at most $q/2^h$, where h is the min-entropy of the distribution of messages in the $(2^\tau\delta)$-evaluation race game (it is at least k). Otherwise, $\mathcal{A}_1$ outputs the following algorithm $\mathcal{B}_1$. When receiving as input the challenge x, $\mathcal{B}_1$ 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 $\mathsf{trapdoor}_{\mathsf{pk}}(-, \mathsf{h}, \Delta)$. Then $\mathcal{B}_1$ can invoke $\mathcal{B}$ on input m while simulating the oracles H_G and $H_{\mathtt{prime}}$. Whenever $\mathcal{B}$ outputs $\mathsf{h} = (h_1, ..., h_\tau, P_1, ..., P_{\tau-1})$, $\mathcal{B}_1$ outputs h_τ^4, which equals $x^{2^{2^\tau}}$ whenever $h_\tau = x^{2^{2^\tau-2}}$. We assume that simulating the oracle has a negligible cost, so $\mathcal{B}_1(x)$ has essentially the same time-cost as $\mathcal{B}(m)$. Then, $\mathcal{A}_1$ wins the $(\delta, 2^\tau)$-time-lock game with probability $p_1 \geq p'_{\mathsf{win}}(1 - q/2^k)$.

Constructing $\mathcal{A}'_2$. Instead of directly building $\mathcal{A}_2$, we build an algorithm $\mathcal{A}'_2$ playing the game $\mathcal{G}_X^{\mathsf{root}}(\mathcal{A}, \mathcal{O})$, 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 $b_1, b_2 \in G$, and any integer $i \in \mathbf{Z}_{>0}$, let

$$X(G, \mathtt{bin}(i)|||\mathtt{bin}(b_1)|||\mathtt{bin}(b_2)) = b_1^{2^{2^i-1}}/b_2,$$

and let $X(G, s) = 1_G$ for any other string s. When receiving pk, $\mathcal{A}'_2$ 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 $\mathcal{O}$. When $\mathcal{A}$ outputs an algorithm $\mathcal{B}$, $\mathcal{A}'_2$ generates a random message m and runs $\mathcal{B}$ on input m. Again, we can assume based on Remark 1 that $\mathcal{B}$ does not call the timed-hashing oracle. Then, $\mathcal{B}(m)$ outputs $(h_1, ..., h_\tau, P_1, ..., P_{\tau-1})$. Let $i > 1$ be the smallest index such that $h_i \neq h_1^{2^{2^i-2}}$. If there is no such index, abort. Otherwise, output $s = \mathtt{bin}(i-1)|||\mathtt{bin}(h_{i-1})|||\mathtt{bin}(h_i)$ and $v = c_1^{2^{2^i-1}}/c_2$, where $(c_1, c_2) = P_{i-1}$. If such an index was found, and $\mathcal{A}$ won the simulated evaluation game, then $v^{\mathcal{O}(s)} = X(G, s) \neq 1_G$, so $\mathcal{A}'_2$ wins the game. This happens with

probability

$$\begin{aligned} p'_2 &\geq \Pr\left[\mathcal{A} \text{ wins and } h_\tau \neq h_1^{2^{2^\tau-2}}\right] \\ &= p_{\mathsf{win}} - \Pr\left[\mathcal{A} \text{ wins and } h_\tau = h_1^{2^{2^\tau-2}}\right] \\ &= p_{\mathsf{win}} - p'_{\mathsf{win}}. \end{aligned}$$

Since $\mathcal{A}$ was limited to q oracle queries, $\mathcal{A}'_2$ also does not do more than q queries. Applying Lemma 1, there is an algorithm $\mathcal{A}_2$ winning the game $\mathcal{G}^{\mathsf{root}}(\mathcal{B})$ with probability $p_2 \geq p'_2/(q+1)$. To conclude the proof, we have

$$p_{\mathsf{win}} \leq p'_{\mathsf{win}} + p'_2 \leq \frac{p_1}{1 - q/2^k} + (q+1)p_2.$$

□

Remark 6. The soundness of the construction can be proven in exactly the same way, by relating it to the game $\mathcal{G}^{\mathsf{root}}$. Indeed, an algorithm $\mathcal{A}$ breaking the soundness leads to an algorithm playing $\mathcal{G}^{\mathsf{root}}$ (or $\mathcal{G}_X^{\mathsf{root}}$) via essentially the same construction as $\mathcal{A}'_2$, except that instead of generating a random message m, $\mathcal{A}'_2$ uses the output m of $\mathcal{A}$.

Acknowledgements

The author wishes to thank Serge Vaudenay, Alexandre Gélin, and Arjen K. Lenstra 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. D. Chaum and T. Pedersen. Wallet databases with observers. In E. Brickell, editor, *Advances in Cryptology CRYPTO 92*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer Berlin Heidelberg, 1993.
10. CPU-Z OC world records. http://valid.canardpc.com/records.php, 2015.
11. 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.
12. A. K. Lenstra and B. Wesolowski. Trustworthy public randomness with sloth, unicorn and trx. *International Journal of Applied Cryptology*, 2016.
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

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

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 $t_{\mathsf{sk}}(H'(m),\Delta)$.

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 timed-hashing oracle.