A proof of time or knowledge

Benjamin Wesolowski

A proof of time or knowledge

Benjamin Wesolowski

Univ. Bordeaux, CNRS, Bordeaux INP, IMB, UMR 5251, F-33400, Talence, France
INRIA, IMB, UMR 5251, F-33400, Talence, France

Abstract. *This note was written in 2016. Rejected from PKC 2017, some of the ideas herein later developed into the Eurocrypt 2019 article* Efficient verifiable delay functions*. Other ideas, such as the construction of* fading signatures*, and a discussion on their (in)feasibility, never appeared in public work. In light of the recent development of time-sensitive cryptography, some of this content may have become of interest. The reader may notice that the notion of* proof of time or knowledge *essentially coincides with what is now known as a* (trapdoor) verifiable delay function*.*

This paper introduces *proofs of time or knowledge*, a new primitive in the field of time-sensitive cryptography pioneered by Rivest, Shamir and Wagner in 1996. A party, Alice, has a pair of secret and public keys. Given a piece of data m, a proof of time or knowledge allows to generate a proof p such that anyone can easily verify that either p has been generated by Alice (i.e., she used her secret key), or the party who computed p spent a prescribed amount Δ of wall-clock time to compute p from m. Suppose that a party, Bob, knows that the message m was not known by Alice before a point in time t_0. Then, Bob can infer that Alice computed the proof p if, and only if, the point in time $t_0 + \Delta$ has not been reached yet (in this case, (m, p) has the same value as a signature of Alice on m). After point in time $t_0 + \Delta$ (or if no bound t_0 is known), the pair (m, p) is an indistinguishable proof of time or knowledge, since anyone could have produced it.

1 Introduction

In 1996, Rivest, Shamir and Wagner [12] introduced 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 [2, 3] suggested *time capsules* for key escrowing in order to counter the problem of early recovery. Boneh and Naor [4] introduced *timed commitments*: a hiding and binding commitment scheme, which can be *forced open* by a procedure of determined running time. Applications of timed commitments include *time capsule signatures*: a signature that becomes valid only

a predetermined time after being released, or when the signer provides a trapdoor. Time capsule signatures were later reintroduced by Dodis and Yum [6] and subsequently enjoyed further attention [13, 8, 10]. The notion of *slow-timed hash* (or *sloth*, a hash function which is slow to compute but whose result is easy to verify) was introduced in [9] and was used to provide trust to the generation of public random numbers. Among other applications, it allows the construction of a secure and verifiable *random beacon*: an online service that makes available fresh, unpredictable, random numbers at regular intervals, even if the service provider is not considered a trusted party.

This paper introduces *proofs of time or knowledge*. A party, Alice, has a secret key sk. Given a piece of data m that was generated at time t_0, a proof of time or knowledge allows to generate a message p such that anyone can easily verify that either p has been generated by Alice (using sk), or the party who computed p spent wall-clock time at least Δ to compute p from m. In essence, the security requirements of these proofs encode the following properties:

1. If a lower bound on t_0 is publicly known (for instance, if m contains the value of a secure random beacon at time t_0), and if p is divulged before point in time $t_0 + \Delta$, then p acts as a signature of Alice on m: no one else would have been able to compute p at that point in time.
2. However, if p is divulged after point in time $t_0 + \Delta$ (or if t_0 is not publicly known), then the pair (m, p) constitutes an indistinguishable proof of the knowledge of sk or of wall-clock time Δ worth of computation.

When someone computed p without knowledge of sk, it also implies that the computing party knew the input data m at point in time $t_1 - \Delta$, where t_1 is the time when p is revealed: the scheme can therefore also be seen as a proof that the party computing p either knows sk or knew the input m in advance. The construction we propose achieves perfect indistinguishability in the property 2 above: the proof computed by Alice is the same as the proof anyone can forge in time Δ.

These properties allow for very simple and interesting protocols. Alice can prove her identity to Bob in one round, and in a perfectly zero-knowledge way as follows: Bob chooses a random challenge c, and Alice responds with a proof of time or knowledge on input c, with Δ set to 10 minutes. Since Alice can respond immediately thanks to her secret, Bob is convinced of her identity. Anyone else can compute the response to the challenge in constant time Δ, so it is perfectly zero-knowledge with

respect to Alice's secret. This approach to proving someones identity is very different from the standard approaches. Choosing Δ to be a hundred years, a proof on any piece of data m has, in the real world, the same value as a signature of Alice on m; but this "signature" is simulatable in constant time with respect to the security parameters, whereas signatures in the classical model are inherently non-simulatable.

This notion is very close to the idea of *fading signatures*, a term originally coined in the opening statement of [7]. A fading signature remains valid only for a limited time: a party verifying the signature after that time limit cannot convince himself of the validity of the signature. Data signed with a fading signature can initially be trusted, but comes with no long-term proof of its authenticity. The construction of a fading signature in [7] did not allow efficient verification: verifying the validity of the signature took essentially as much time as the validity period. The construction of proofs of time or knowledge we introduce in the present paper provides an efficiently verifiable fading signature scheme. Note however that the *fading* properties of fading signatures are to be considered with caution, as discussed in Section 5.

Proofs of time or knowledge are also reminiscent of indistinguishable proofs of work or knowledge [1], with the difference that wall-clock timing is a much more delicate measurement than total workload. Note that measuring the workload rather that the time does not permit to infer anything about the identity of the party performing the computation: it does not give rise to a construction similar to that of fading signatures, which can prove the identity of the signing party under appropriate conditions.

Notation. The integer k denotes a security level (typically, $k \in \{128, 192, 256\}$). With k clear from the context, the cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^{2k}$ denotes a $2k$-bit version of a secure hash algorithm. The function $\mathtt{int} : \{0,1\}^* \to \mathbf{Z}_{\geq 0}$ maps $x \in \{0,1\}^*$ in the canonical manner to the non-negative integer with binary representation x. Given any integer N, and any element $x \in \mathbf{Z}/N\mathbf{Z}$, let $\hat{x}$ be the unique integer in $[0, N)$ congruent to x modulo N. 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 a special separating character in $\mathcal{A} \setminus \mathcal{H}$.

2 Proofs of time or knowledge

Let Δ be a time duration. A party, Alice, has a public key pk and a secret key sk. Let m be a piece of data generated at time t_0. Alice, thanks to

her secret key sk, is able to quickly evaluate a function $\mathsf{shortcut}_{\mathsf{sk}}$ on m. On the other hand, other parties knowing only pk can produce a piece of data $\mathsf{forge}_{\mathsf{pk}}(m)$ at time $t_0 + \Delta$, but not before (and important parallel computing power does not give a substantial advantage in going faster), such that the value $\mathsf{forge}_{\mathsf{pk}}(m)$ is indistinguishable from $\mathsf{shortcut}_{\mathsf{sk}}(m)$.

More formally, a proof of time or knowledge consists of the following components:

$\mathsf{keygen} \to (\mathsf{pk}, \mathsf{sk})$ is a key generation procedure, which outputs a signer'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.

$\mathsf{shortcut}_{\mathsf{sk}}(m, \Delta) \to \mathsf{pro}$ takes as input the data $m \in \mathcal{M}$ (for some input space $\mathcal{M}$), and uses the secret key sk to produce a proof pro on m. The parameter Δ is the time it should take to forge the same proof without knowledge of the secret key.

$\mathsf{forge}_{\mathsf{pk}}(m, \Delta) \to \mathsf{pro}$ is a procedure to forge a valid proof on m using only the public key pk, for a targeted computation time Δ. This procedure is meant to be infeasible in time less than Δ.

$\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{pro}, \Delta) \to \mathsf{true}$ **or** false is a procedure to check if pro is a valid proof on m, associated to the public key pk and the forging time Δ.

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, if pro is either a rapid proof $\mathsf{shortcut}_{\mathsf{sk}}(m, \Delta)$ or a forged proof $\mathsf{forge}_{\mathsf{pk}}(m, \Delta)$, then $\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{pro}, \Delta)$ outputs true.

The security relies on two aspects: on one hand, a valid proof cannot be produced in time less than Δ without knowledge of the secret key sk; on the other hand, given X the output of $\mathsf{shortcut}_{\mathsf{sk}}(m, \Delta)$ and Y the output of $\mathsf{forge}_{\mathsf{pk}}(x, \Delta)$, distinguishing X from Y is infeasible. If the procedures $\mathsf{shortcut}$ and forge are deterministic, this second property simply translates into the fact that they have the same output. If they are not, it can be formalized by a distinguishing game, but we will restrict here to the deterministic case. The first property can be formalized via the Δ-*forgery race* game.

Definition 1 (Δ-forgery race game). *Let $\mathcal{A}$ be a party playing the game. It goes as follows: at the start of the game,* keygen *is run and the public key* pk *is given to $\mathcal{A}$; it is the beginning of the precomputation phase. Whenever $\mathcal{A}$ outputs "*`ready`*", a message $m \in \mathcal{M}$ is generated uniformly at random and given to $\mathcal{A}$. Party $\mathcal{A}$ wins the Δ-forgery race game if it produces in time less than Δ a value* pro *such that* $\mathsf{verify}_{\mathsf{pk}}(m, \mathsf{pro}, \Delta)$ *returns* true.

Data: a secret key $\mathsf{sk} = (N, \phi(N))$, some data $m \in \mathcal{A}^*$, for a targeted forgery time exponential in ℓ.
Result: the proof pro.
$h \leftarrow H_N(m) \in \mathbf{Z}/N\mathbf{Z}$;
for $i = 1, ..., \ell$ **do**
  $y_i \leftarrow 2^{2^i - 2} \mod \phi(N)$;
  $h_i \leftarrow h^{y_i}$;
end
for $i = 1, ..., \ell - 1$ **do**
  $B \leftarrow H_{\mathtt{prime}}(\mathtt{hex}(i)|||\mathtt{hex}(h_i)|||\mathtt{hex}(i+1))$;
  $x \leftarrow 2y_i$;
  $r \leftarrow$ least residue of $2^{2^i - 2}$ modulo B;
  $\beta \leftarrow (x - r)B^{-1} \mod \phi(N)$;
  $P_i \leftarrow (h^\beta, h^{x\beta})$;
end
return $\mathsf{pro} = (h_1, h_2, ..., h_\ell, P_1, ..., P_{\ell-1})$;

Algorithm 1: $\mathsf{shortcut}_{\mathsf{sk}}(m, \ell) \to \mathsf{pro}$

A proof of time or knowledge is Δ-*secure* if any player whose precomputation phase is polynomially bounded (with respect to the implicit security parameter) wins the Δ-forgery race game with negligible probability. Observe that it is useless to allow $\mathcal{A}$ to adaptively ask for legitimate proofs during the precomputation phase: for any data m', the procedure $\mathsf{forge}_{\mathsf{pk}}(m', \Delta)$ produces the same output as $\mathsf{shortcut}_{\mathsf{sk}}(m', \Delta)$, so any proof the adversary would like to request during the precomputation phase can be replaced by a forgery doable in time $O(\Delta)$.

3 A proof of time or knowledge

Let $m \in \mathcal{A}^*$ be the input of the proof. Alice's secret key sk is a pair of distinct prime numbers p and q, and her public key pk is the RSA modulus $N = pq$. Define $H_N : \mathcal{A}^* \to \mathbf{Z}/N\mathbf{Z}$ by $H_N(m) = \mathtt{int}(H(\text{“}\mathtt{residue}\text{”}||m))$ mod N. Also, given any string s, we denote by $H_{\mathtt{prime}}(s)$ the first prime number in the sequence $H(\text{“}\mathtt{prime}\text{”}||\mathtt{hex}(j)|||s)$, for $j \in \mathbf{Z}_{\geq 0}$. The procedures to sign, verify and forge are fully described in Algorithms 1, 2 and 3 respectively, and explained in detail below.

To built the proof on m, first let $h = H_N(m)$. The basic idea is that for any $t \in \mathbf{Z}_{>0}$, Alice can efficiently compute h^{2^t} with 2 modular exponentiations, by first computing $x = 2^t \mod \phi(N)$ where $\phi(N) = (p-1)(q-1)$ is the order of the group $(\mathbf{Z}/N\mathbf{Z})^\times$, followed by $h^{2^t} = h^x$ mod N. The running time is logarithmic in t. Any other party who does not know $\phi(N)$ can also compute it by performing t sequential squarings,

Data: a public key $\mathsf{pk} = N$, some data $m \in \mathcal{A}^*$ for a targeted forgery time exponential in ℓ, and a proof pro.
Result: true if the proof pro on m is valid, false otherwise.
$(h_1, h_2, ..., h_\ell, P_1, ..., P_{\ell-1}) \leftarrow \mathsf{pro}$;
$h \leftarrow H_N(m) \in \mathbf{Z}/N\mathbf{Z}$;
if $h_1 \neq h$ **then**
  return false;
end
for $i = 1, ..., \ell - 1$ **do**
  $B \leftarrow H_{\mathtt{prime}}(\mathtt{hex}(i)|||\mathtt{hex}(h_i)|||\mathtt{hex}(i+1))$;
  $r \leftarrow$ least residue of 2^{2^i-2} modulo B;
  $(c_1, c_2) \leftarrow P_i$;
  if $h_i^2 \neq c_1^B h^r$ *or* $h_{i+1} \neq \pm c_2^B {h_i}^{2r}$ **then**
    return false;
  end
end
return true;

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

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} takes as long as the forgery: 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 [4]. Let $t = 2^\ell$, and rather than just publishing $h^{2^{2^\ell}}$, publish the sequence of ℓ elements

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

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 [5]. It should be stressed that this protocol was designed to be zero-knowledge with respect to the exponents and not with respect to the group order (the exponents are secret in the Diffie-Hellman setting, and the group order is not; in the present setting, the opposite is true). The modified version in [4], contrary to what is claimed, is not perfectly zero-knowledge with respect to Alice's secret $\phi(N)$. Furthermore, the construction of the proof should be non-interactive; unfortunately there is no obvious way to make the suggested protocol non-interactive while still allowing timed forgeries. Therefore the following introduces a new approach, which not only is perfectly zero-knowledge with respect

Data: a public key $\mathsf{pk} = N$, some data $m \in \mathcal{A}^*$, for a targeted forgery time exponential in ℓ.
Result: the (forged) proof pro.
$h \leftarrow H_N(m) \in \mathbf{Z}/N\mathbf{Z}$;
$h_1 \leftarrow h$;
for $i = 1, ..., \ell - 1$ **do**
  $B \leftarrow H_{\texttt{prime}}(\texttt{hex}(i)|||\texttt{hex}(h_i)|||\texttt{hex}(i+1))$;
  $c_1 \leftarrow \mathsf{forge\_rec}(h, B, i)$; `// this function is described in Algorithm 4`
  $c_2 \leftarrow c_1^{2^{2^i-1}}$;
  $P_i \leftarrow (c_1, c_2)$;
  $h_{i+1} \leftarrow h^{2^{2^{i+1}-2}} \in \mathbf{Z}/N\mathbf{Z}$;
end
return $\mathsf{pro} = (h_1, h_2, ..., h_\ell, P_1, ..., P_{\ell-1})$;

Algorithm 3: $\mathsf{forge}_{\mathsf{pk}}(m, \ell) \to \mathsf{pro}$

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-2} by B.
if $2^{2^i-2} < B$ **then**
  return 1_G;
else
  $x \leftarrow \mathsf{forge\_rec}(h, B, i-1)$;
  $y \leftarrow \mathsf{forge\_rec}(x, B, i-1)$;
  $\alpha \leftarrow 2^{i-1} - 2 \mod B - 1$;
  $r \leftarrow$ least residue of 2^α modulo B;
  $q \leftarrow$ quotient of the euclidean division of $4r^2$ by B;
  return $y^{4B} x^{8r} h^q$;
end

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

to $\phi(N)$, but can also be made non-interactive while remaining perfectly zero-knowledge and forgeable.

3.1 The basic construction.

The proof on input m is the tuple

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

where $h_i = h^{2^{2^i-2}} \in \mathbf{Z}/N\mathbf{Z}$ for $i = 1, ..., \ell$, and P_i is a (non-interactive) proof that (h, h_i^2, h_{i+1}) is a triple of the form $(h, h^x, \pm h^{x^2})$. The algorithm is synthesised in Algorithm 1. The construction of the proofs P_i will be discussed in the following paragraph.

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

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

Therefore, by induction, if all the proofs are correct, then $h_\ell = \pm h^{2^{2^\ell-2}}$. The verification procedure is synthesised in Algorithm 2.

3.2 The proofs P_i.

The following focuses on the problem of proving efficiently that (h, b_1, b_2) is of the form $(h, h^x, \pm h^{x^2})$, without revealing any information about $\phi(N)$. 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 $\phi(N)$ should not be leaked: that would allow to compute a multiple of $\phi(N)$, and to factor N). 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 $(h, h^x, \pm h^{x^2})$. First, consider an interactive protocol. The verifier first receives h, b_1 and b_2, and then sends a (large) random prime number B to the prover. The prover then computes the least residue r of x modulo B. The prover then computes $c_1 = h^\beta$ and $c_2 = h^{x\beta}$ such that $\beta \equiv (x - r)B^{-1} \mod \phi(N)$, and the proof sent to the verifier 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 $B^{-1} \mod \phi(N)$. The procedure will be described later (in Subsection 3.3), but assuming that fact, the protocol is perfectly zero-knowledge since the secret is not necessary to produce the proof. The verifier simply computes r and checks that $b_1 = c_1^B h^r$ and $b_2 = \pm c_2^B b_1^r$. It is straightforward to check that this holds if the prover is honest.

Now, what can a dishonest prover do? In fact, 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 $b_2 = \pm c_2^B b_1^r$, where r is the least residue r of x modulo B. Exponentiating the first equality by x yields $b_1^x = c_1^{xB} h^{xr}$, and the second can be written as $b_2 = \pm c_2^B h^{xr}$. Therefore $b_1^x / b_2 = (c_1^x / c_2)^B$. When publishing b_1 and b_2, $\alpha = b_1^x / b_2$ 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 that c_1^x/c_2 is a B-th root of α. For a prover to cheat and succeed with good probability, it is necessary that B-th roots of α to be easily extracted for arbitrary B. This 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 can be made non-interactive by letting B be, for instance, the first prime in the sequence of integers represented by

$$H(\text{“}\texttt{prime}\text{”}||\texttt{hex}(j)|||\texttt{hex}(i)|||\texttt{hex}(b_1)|||\texttt{hex}(b_2)),$$

for $j \in \mathbf{Z}_{>0}$, where i is the index of P_i. It is still forgeable, so still zero-knowledge. This simulates a choice of B uniform among the primes in $[0, 2^{2k})$. Recall that given a string s, $H_{\texttt{prime}}(s)$ denotes the first prime number in the sequence $H(\text{“}\texttt{prime}\text{”}||\texttt{hex}(j)|||s)$. If H is considered to be a random function with uniform distribution, let μ denote the probability distribution of the output of $H_{\texttt{prime}}(s)$.

Definition 2 ((Δ, ℓ)-exponentiation race game). *Let $\mathcal{A}$ be a party playing the game. The* (Δ, ℓ)-exponentiation race game *goes as follows: first, the* keygen *procedure is run and the public key* $\mathsf{pk} = N$ *is given to $\mathcal{A}$; it is the beginning of the precomputation phase. Whenever $\mathcal{A}$ outputs “*`ready`*”, an element $x \in \mathbf{Z}/N\mathbf{Z}$ is generated uniformly at random and given to $\mathcal{A}$. Party $\mathcal{A}$ wins the (Δ, ℓ)-exponentiation race game if it produces in time less than Δ a value $y = x^{2^{2^\ell}-2}$.*

The (Δ, ℓ)-exponentiation race game is assumed to be difficult for $\ell = O(\log \Delta)$. I.e., the simple strategy consisting of sequential squarings is optimal (up to some optimisations by a constant factor).

3.3 Forging the proofs in time $O(\Delta)$.

As $B^{-1} \mod \phi(N)$ cannot be computed without knowledge of the private key, an alternative strategy is needed for the forgery. Assuming the correctness of the function $\mathsf{forge\_rec}(h, B, i)$, which computes h^β where β is the quotient of the euclidean division of 2^{2^i-2} by B, it is easy to check that the function $\mathsf{forge}_{\mathsf{pk}}(m, \ell)$ described in Algorithm 3 is correct. The function $\mathsf{forge\_rec}(h, B, i)$, described in Algorithm 4 deserves more explanations. It operates by recursion on i. The base case is easy: if 2^{2^i-2} is smaller than B, then β is zero, so h^β is the identity element of the group. In particular, if $i = 1$, then $2^{2^i-2} = 1 < B$.

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

$$2^{2^i-2} = 4\left(2^{2^{i-1}-2}\right)^2 = 4(\beta' B + r)^2 = (4\beta'^2 B + 8\beta' r)B + 4r^2.$$

Therefore, if q is the quotient of the euclidean division of $4r^2$ by B, then $\beta = 4\beta'^2 B + 8\beta' r + q$. We can recursively compute $h^{\beta'} = \mathsf{forge\_rec}(h, B, i-1)$ and $h^{\beta'^2} = \mathsf{forge\_rec}(h^{\beta'}, B, i-1)$. Then h^β can be computed as

$$h^\beta = \left(h^{\beta'^2}\right)^{4B} \left(h^{\beta'}\right)^{8r} h^q.$$

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 a prime number), and store the results for later use. Therefore leaving this operation aside, as a preprocessing step, the processing time 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 Δ). If $T(i)$ denotes the running time of $\mathsf{forge\_rec}(h, B, i)$, then

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

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

4 Security analysis

The following game formalises the problem of finding 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 factorization of N is unknown. This problem is supposedly difficult when N is large enough, and 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 keygen.

Definition 3 (The root finding game G^{root}). *Let $\mathcal{A}$ be a party playing the game. The* root finding game $G^{\mathsf{root}}(\mathcal{A})$ *goes as follows: first, the* keygen *procedure is run and the public key $\mathsf{pk} = N$ is given to $\mathcal{A}$ (N is an RSA modulus). The player $\mathcal{A}$ then outputs an integer u modulo N. 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 $u \not\equiv 0, \pm 1 \mod N$ and $v^B \equiv u \mod N$.*

Definition 4 (The oracle root finding game G_X^{root}). *Let $\mathcal{A}$ be a party playing the game, let $X : \mathcal{A}^* \to \mathbf{Z}/N\mathbf{Z}$ be a map, and $\mathcal{O} : \mathcal{A}^* \to \mathbf{Z}_{>0}$ a random oracle with distribution μ. The player has access to the random oracle $\mathcal{O}$. The* oracle root finding game $G_X^{\mathsf{root}}(\mathcal{A}, \mathcal{O})$ *goes as follows: first, the* keygen *procedure is run and the public key $\mathsf{pk} = N$ is given to $\mathcal{A}$. The player $\mathcal{A}$ then outputs a string $s \in \mathcal{A}^*$, and an integer v modulo N. The game is won if $X(s) \not\equiv 0, \pm 1$ and $v^{\mathcal{O}(s)} = X(s)$.*

Lemma 1. *If an algorithm $\mathcal{A}$ limited to q queries to the oracle $\mathcal{O}$ wins the game $G_X^{\mathsf{root}}(\mathcal{A}, \mathcal{O})$ with probability p_{win}, there is an algorithm $\mathcal{B}$ winning the game $G^{\mathsf{root}}(\mathcal{B})$ 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 G^{root}, using $\mathcal{A}'$ as follows. Upon receiving pk, $\mathcal{B}$ 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 $u = X(s)$, concluding the first round of the game G^{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 integer v modulo N. To conclude the game $G^{\mathsf{root}}(\mathcal{B})$, $\mathcal{B}$ 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 $X(s') \not\equiv 0, \pm 1$ and $v^{\mathcal{O}(s')} =$

$X(s')$; denote this event $\mathsf{win}_{\mathcal{A}'}$. If $s = s'$, these conditions are exactly $u \not\equiv 0, \pm 1$ and $v^B = u$, where $u = X(s)$ is the output for the first round of G^{root}, and $\mathcal{O}(s) = B$ is the input for the second round. If these conditions are met, the game $G^{\mathsf{root}}(\mathcal{B})$ is won. Therefore

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

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 $\Pr[\mathcal{B} \text{ wins } G^{\mathsf{root}}] \geq p_{\mathsf{win}}/(q+1)$. 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. □

Proposition 1 (Security of the proof of time or knowledge in the random oracle model). *Let $\mathcal{A}$ be a player winning with probability p_{win} the Δ-forgery race game associated to the proposed scheme, assuming H_N and $H_{\mathtt{prime}}$ are random oracles and $\mathcal{A}$ is limited to q oracle queries. Then, there is a player $\mathcal{B}_1$ for the (Δ, ℓ)-exponentiation race game, and $\mathcal{B}_2$ for the root finding game G^{root}, with respective winning probabilities p_1 and p_2 with $p_1 + (q+1)p_2 \geq p_{\mathsf{win}}$, and with same running time as $\mathcal{A}$ (up to a constant factor).*

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

Constructing $\mathcal{B}_1$. Build $\mathcal{B}_1$ as follows. Upon receiving pk, $\mathcal{B}_1$ starts running $\mathcal{A}$ on input pk. The random oracles H_N 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 $\mathcal{A}$ outputs "`ready`", $\mathcal{B}_1$ looks for a string m that has not been queried to the oracle yet, and outputs "`ready`". When receiving the challenge h, $\mathcal{B}_1$ adds x to the table of oracle H_N, for the input m (i.e., $H_N(m) = x$). Then $\mathcal{B}_1$ sends m to $\mathcal{A}$ and continues to run it while simulating the oracle. Whenever $\mathcal{A}$ outputs $\mathsf{pro} = (h_1, ..., h_\ell, P_1, ..., P_{\ell-1})$,

$\mathcal{B}_2$ outputs h_ℓ. The game is won with probability $p_1 \geq p'_{\mathsf{win}}$.

Constructing $\mathcal{B}'_2$. Instead of directly building $\mathcal{B}_2$, we build an algorithm $\mathcal{B}'_2$ playing the game $G_X^{\mathsf{root}}(\mathcal{A}, \mathcal{O})$, and apply Lemma 1. Define $X : \mathcal{A}^* \to \mathbf{Z}/N\mathbf{Z}$ as follows: for any integers $i, b_1, b_2 \in \mathbf{Z}_{>0}$, let

$$X(\mathtt{hex}(i)|||\mathtt{hex}(b_1)|||\mathtt{hex}(b_2)) = b_1^{2^{2^i-1}}/b_2 \mod N,$$

and let $X(s) = 0$ for any other string s. When receiving pk, $\mathcal{B}'_2$ starts running $\mathcal{A}$ with input pk. The oracle H_N 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 "`ready`", $\mathcal{B}'_2$ generates a random message m and continues running $\mathcal{A}$. Then, $\mathcal{A}$ outputs $(h_1, ..., h_\ell, P_1, ..., P_{\ell-1})$. Let $i > 1$ be the smallest index such that $h_i \neq \pm h_1^{2^{2^i-2}}$. If there is no such index, abort. Otherwise, output $s = \mathtt{hex}(i-1)|||\mathtt{hex}(h_{i-1})|||\mathtt{hex}(h_i)$ 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 forgery game, then $X(s) \not\equiv 0, \pm 1$, and $v^{\mathcal{O}(s)} = X(s)$, so $\mathcal{B}'_2$ wins the game. This happens with probability

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

Since $\mathcal{A}$ was limited to q oracle queries, $\mathcal{B}'_2$ also does not do more than q queries. Applying Lemma 1, there is an algorithm $\mathcal{B}_2$ winning the game $G^{\mathsf{root}}(\mathcal{B})$ with probability $p_2 \geq p'_2/(q+1)$. To conclude the proof, we have $p_{\mathsf{win}} \leq p_1 + p'_2 \leq p_1 + (q+1)p_2$. □

5 Fading signatures and their (in)feasibility

The idea of fading signatures relies on a fairly simple idea: a signature is trustworthy only as long as a forgery is believed to be infeasible. Just like in a conventional signature scheme, the signing party, Alice, has a public key pk and a secret key sk. Her secret key allows Alice to efficiently produce signatures, while the public key allows anyone to efficiently check their validity. In the conventional setting, forging valid signatures without knowledge of sk is meant to be infeasible. Fading signatures however can be forged by anyone. The only distinction between legitimate and forged

signatures comes from time constraints. Given a fading-time length Δ, a fading signature scheme provides the following guarantee: given a message m generated at point in time t_0, Alice, thanks to sk, can quickly produce a signature $\mathsf{shortcut}_{\mathsf{sk}}(m)$ on m at time $t_0+\epsilon$ (for a very short ϵ), whereas other parties, knowing only pk, can produce a valid signature on m at time $t_0 + \Delta$, but not before (and important parallel computing power does not give a substantial advantage in going faster). If Bob wants to check the authenticity of a valid signature $\mathsf{shortcut}_{\mathsf{sk}}(m)$, he should be somehow aware of the time t_0 when m was generated. Guarantees on t_0 can be provided in different ways in different contexts, the simplest of which would be to rely on a secure random beacon. At this point is should be clear that this notion of fading signature is simply another way of thinking about the previously introduces proofs of time or knowledge.

However, the intuitive idea of fading signatures which seems to have motivated the construction of [7] is more powerful: it should be a signature scheme S that would allow Alice to produce a signature σ on a message m at time t_0, such that anyone seeing σ before time $t_0 + \Delta$ could be sure Alice signed it, but after this delay, nobody else could be sure Alice signed it. This is *not* something that proofs of time or knowledge allow, and we show that it is essentially impossible to achieve.

The following approach, combining a proof of time or knowledge with a secure random beacon, would be tempting. Let $r = \mathsf{beacon}(t_0)$ be the value generated by the beacon at time t_0. Then, Alice signs $\mathsf{shortcut}_{\mathsf{sk}}(r||m)$, and anyone can check the validity of the signature, with the guarantee that r was not known by any party before time t_0 (even by the beacon itself, if it is based on a secure and verifiable protocol as described in [9]; the beacon therefore needs not be considered a trusted third party). Then indeed, after time $t_0+\Delta$, Bob could not check *by himself* that Alice indeed produced that signature.

But — and this applies to any scheme S — if a third party, Charlie, checked that signature σ on time, and if Bob trusts Charlie, the latter can certify to Bob that Alice produced the signature. Even though Bob did not check the signature on time, he can still be convinced that Alice signed it, because he trusts Charlie. This means that assuming Charlie cannot lie, the signature cannot fade away because at any point in the future Charlie can testify.

Of course, Charlie could be a liar, and this scheme might be secure in a world where nobody can be trusted. Ironically, the existence of trust between different parties implies the impossibility of signature schemes with such properties. In the real world, Charlie could be a trusted time-

stamping service. More slyly, Charlie could publish Alice's signature on a distributed public ledger, *à la* Bitcoin [11], providing a convincing argument to anyone checking the ledger that the signature was indeed produced within some tight time interval.

References

1. F. Baldimtsi, A. Kiayias, T. Zacharias, and B. Zhang. Indistinguishable proofs of work or knowledge. In *Advances in Cryptology – ASIACRYPT 2016*, pages 902–933. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
2. M. Bellare and S. Goldwasser. Encapsulated key escrow. Technical report, 1996.
3. 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.
4. 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.
5. 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.
6. Y. Dodis and D. H. Yum. Time capsule signature. In *In Financial Cryptography and Data Security 2005*, pages 57–71. Springer-Verlag, 2005.
7. H. Ferradi, R. Géraud, and D. Naccache. Slow motion zero knowledge identifying with colliding commitments. Cryptology ePrint Archive, Report 2016/399, 2016. http://eprint.iacr.org/.
8. B. C. Hu, D. S. Wong, Q. Huang, G. Yang, and X. Deng. Time capsule signature: Efficient and provably secure constructions. In J. Lopez, P. Samarati, and J. Ferrer, editors, *Public Key Infrastructure*, volume 4582 of *Lecture Notes in Computer Science*, pages 126–142. Springer Berlin Heidelberg, 2007.
9. A. K. Lenstra and B. Wesolowski. Trustworthy public randomness with sloth, unicorn and trx. *International Journal of Applied Cryptology*, 2016.
10. B. Libert and J.-J. Quisquater. Practical time capsule signatures in the standard model from bilinear maps. In T. Takagi, T. Okamoto, E. Okamoto, and T. Okamoto, editors, *Pairing-Based Cryptography Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 23–38. Springer Berlin Heidelberg, 2007.
11. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system,http://bitcoin.org/bitcoin.pdf, 2009.
12. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.
13. M. Zhang, G. Chen, J. Li, L. Wang, and H. Qian. A new construction of time capsule signature, 2006.