Time-Based Cryptography From Weaker Assumptions:
Randomness Beacons, Delay Functions and More

Damiano Abram[1]*, Lawrence Roy[1]**, and Mark Simkin[2]***

[1] Aarhus University
[2] Ethereum Foundation

Abstract. The assumption that certain computations inherently require some sequential time has established itself as a powerful tool for cryptography. It allows for security and liveness guarantees in distributed protocols that are impossible to achieve with classical hardness assumptions. Unfortunately, many constructions from the realm of time-based cryptography are based on new and poorly understood hardness assumptions, which tend not to stand the test of time (cf. Leurent et al. 2023, Peikert & Tang 2023).

In this work, we make progress on several fronts. We formally define the concept of a delay function and present a construction thereof from minimal assumptions. We show that these functions, in combination with classical cryptographic objects that satisfy certain efficiency criteria, would allow for constructing delay encryption, which is otherwise only known to exist based on a new hardness assumption about isogenies. We formally define randomness beacons as they are used in the context of blockchains, and we show that (linearly homomorphic) time-lock puzzles allow for efficiently constructing them.

Our work puts time-based cryptography on a firmer theoretical footing, provides new constructions from simpler assumptions, and opens new avenues for constructing delay encryption.

* `damiano.abram@cs.au.dk`

** `ldr709@gmail.com`. Author partially supported by the Danish Independent Research Council (grant DFF-0165-00107B "C3PO") and the DARPA SIEVE program (contract HR001120C0085 "FROMAGER"). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

*** `mark.simkin@ethereum.org`

Table of Contents

Time-Based Cryptography From Weaker Assumptions:
Randomness Beacons, Delay Functions and More 1
Damiano Abram, Lawrence Roy, and Mark Simkin
1 Introduction 2
1.1 Our Contribution 3
1.2 Related Works 4
2 Technical Overview 6
2.1 Randomness Beacons from Time-Lock Puzzles 6
2.2 Delay Functions from Weaker Assumptions 9
2.3 Delay Encryption 10
3 Preliminaries 11
3.1 Time Lock Puzzles 13
3.2 Delay Encryption 14
3.3 Witness Encryption 16
3.4 Strongly Homomorphic Commitments 16
3.5 Simulation-Extractable NIZKs 18
3.6 Garbled Circuits 20
4 Delay Functions 20
4.1 Building Delay Functions 22
5 Delay Encryption 31
5.1 Delay Encryption from Witness Encryption 31
5.2 Delay Encryption from Attribute-Based Encryption 33
6 Randomness Beacons do not Need VDFs 36
6.1 Defining Randomness Beacons 37
6.2 Constructing Randomness Beacons from Time-Lock Puzzles 39
6.3 A More Efficient Construction from Homomorphic Time-Lock Puzzles 43

1 Introduction

Contrary to the enthusiastic Nike[3] slogan "just do it", in cryptography some things you just cannot do. A set of parties *cannot* flip an unbiased coin without an honest majority among the participants, by a 1986 impossibility result of Cleve [Cle86]. Only two years later Micali famously noted that "cryptographers seldom sleep well" [Kil88]. Given how important coin flipping protocols have become in the context of large real-world distributed system, it is of no surprise that cryptographers may lose sleep over Cleve's impossibility result. Nowadays, many large-scale blockchains, like Algorand [GHM+17], Cardano [DGKR18], and Ethereum [Eth] rely on so-called randomness beacons [Rab83, CMB23], that continuously generate fresh, independent, unbiasable random values. Under the hood, these beacons themselves can be seen as distributed systems that repeatedly flip coins and therefore they are subject to Cleve's impossibility result.

[3] The brand, not the key exchange.

An interesting approach for circumventing Cleve's result was proposed by Boneh and Naor [BN00]. They showed that two parties, in the presence of one corruption, can flip an unbiased coin by relying on time-based cryptography [May93, CLSY93, RSW96]. Cleve's result lives in a world where time is independent of computation, meaning that when Alice evaluates a polynomial-sized circuit, the clock ticks only once. Coin flipping protocols that rely on time-based cryptography instead live in a world where computation time can be observed, meaning that the clock can tick multiple times during Alice's computation and that Bob can observe (or upper bound) how long it took. Time-based cryptography is built upon the computational assumption that certain computations can be efficiently performed by circuits of some sufficiently high depth, but not by any polynomially-sized circuit with small depth, along with the physical assumption that evaluating a circuit must take time proportional to its depth, no matter how much resources you invest.

Since the inception of time-based cryptography, dating back to May [May93] and Cai et al. [CLSY93], there have been numerous other works proposing different time-based cryptographic objects such as time-lock puzzles [RSW96], timed commitments [BN00], delay functions [GS98], verifiable delay functions [BBBF18], homomorphic time-lock puzzles [MT19], and delay encryption [BD21]. All of these primitives allow for security and liveness guarantees in distributed systems that are not possible classically.

Unfortunately, many of these works rely on new, poorly understood hardness assumptions, some of which were proven false in terms of concrete parameters [LMP$^+$23] and some of which were proven false in terms of asymptotic guarantees [PT23]. Contrary to classical hardness assumptions that talk about the hardness of problems w.r.t. to adversaries of polynomial size, much less is known about the hardness of problems w.r.t. to adversaries of bounded depth. It is therefore natural to ask, under what minimal assumptions one can construct time-based cryptography.

1.1 Our Contribution

In this work we make progress on several fronts in the domain of time-based cryptography. In the following, let us highlight each contribution individually.

(Verifiable) Delay Functions. The concept of delay functions was informally introduced by Goldschlag and Stubblebine [GS98]. In their work, they do not construct such functions, but argue that their existence would allow for solving certain distributed computing problems. A delay function guarantees that its evaluation on a random input requires computing a high depth circuit. Boneh et al. [BBBF18] introduce the concept of verifiable delay functions (VDFs) and propose several candidate constructions based on new hardness assumptions. Such delay functions allow for efficiently, i.e. in low depth, verifying that some given value is indeed the output of a delay function for some given input.

In our work, we formally define the concept of delay functions and show that they can be constructed from one-way functions and the *existence* of languages that can be decided by circuits of polynomial size, but not by low depth circuits in the worst-case. We note that we do not assume that we are given such a language, but merely that such a language exists. Using an appropriate succinct proof system, our delay functions can be converted into a VDF.

To put our result into context, we would like to stress that the hardness assumption we use, is very weak. For example, should *any* existing candidate construction for delay functions be secure, then so is our construction. Yet, it is also entirely possible that we live in a world, where all currently existing candidate constructions are broken, but our delay function remains secure.

Delay Encryption. A recent work by Burdges and De Feo [BD21] introduced delay encryption, which allows for encrypting messages towards an arbitrarily chosen identity `id`. Anybody can derive the secret key corresponding to `id`, which allows for decrypting the corresponding encrypted messages, but doing so requires evaluating a circuit of high depth. The authors show that delay encryption can be constructed from a new hardness assumption about isogenies that they introduce in their work.

In our work, we show that delay functions together with timeless classics that satisfy certain efficiency criteria, allow for constructing delay encryption. Concretely, we show that either witness encryption [GGSW13] or attribute-based encryption [SW05] suffice. While we do not currently know how to construct witness or attribute-based encryption schemes that satisfy all of our efficiency criteria, our results demonstrate that minimal time-based assumptions in combination with standard cryptographic primitives are sufficient for constructing advanced objects from the realm of time-based cryptography. We believe that our results provide an interesting new avenue towards constructing delay encryption.

Randomness Beacons. We provide a formal model for randomness beacons which is tailored to the blockchain setting and show that time-lock puzzles – one of the simplest time-based primitives – in combination with standard cryptographic tools are sufficient for constructing provably secure beacons. Time-lock puzzles allow for efficiently encoding a secret value inside of a puzzle, such that retrieving the encoded value can only be done by solving the puzzle, which requires a high-depth computation. In our model, parties can only speak once, and the output of the beacon must be publicly verifiable without needing to repeatedly perform high-depth computations. It is desirable that retrieving the beacon output should require as few high-depth computations as possible, and that the beacon output should be available as soon as possible. The ideas behind our construction here are not all entirely new and several of our insights have appeared in various shapes and forms in previous works, but to the best of our knowledge this is the first fully formal treatment of randomness beacons from time-lock puzzles in the specific context of blockchains.

Our modelling of randomness beacons assumes that parties speak one after another and we assume that honest parties speak frequently enough, i.e. that every subsequence of T consecutive speakers contains at least one honest party. Our construction of a randomness beacon from plain time-lock puzzles requires solving one time-lock puzzle per beacon output on average, but solving $\mathcal{O}(T)$ many puzzles for retrieving a single beacon output. We show that a variant of linearly homomorphic time-lock puzzles, introduced and constructed by Malavolta and Thyagarajan [MT19], allows for reducing the worst-case costs to $\mathcal{O}(1)$ alone. The constructions of Malavolta and Thyagarajan require a trusted setup, which is difficult to realize in practice. We generalize their approach (and their hardness assumption) and show how to construct the linearly homomorphic time-lock puzzles we need in a general algebraic framework, similar to the frameworks used in several recent works of Abram et al. [ADOS22, ADIN24, ARS24]. Instantiating this framework with class groups and an appropriate hardness assumption, provides us with our desired time-lock puzzles without the need for a trusted setup.

1.2 Related Works

Time-based cryptography has been around for a while. In 1993, May [May93] suggested encrypting messages to the future with the help of a social mechanism that would allow decrypting messages

after some time has passed. In the same year, Cai et al. [CLSY93] introduced the concept of uncheatable benchmarks, which allow a verifier to efficiently check an inherently slow computation done by a powerful machine. Cai et al. proposed a candidate construction based on the assumption that the fastest way to perform exponentiations in groups of unknown order, the RSA group specifically, is via repeatedly performing squaring operations sequentially. To this day, this is one of the main hardness assumptions underlying a large part of time-based cryptography [RSW96, BN00, MT19, Pie19, Wes19, EFKP20, FKPS21, BDD+21, BDD+23], including the linearly homomorphic puzzles we will use in our randomness beacon.

Assumptions Needed for Time-Based Cryptography. While currently not much is known about which assumptions can be used for time-based cryptography, we know at least some things. Mahmoody, Moran, and Vadhan [MMV11] showed that time-lock puzzles, where generating the puzzle is much faster than solving it, cannot be constructed from random oracles alone. Their result was later extended by Mahmoody, Smith, and Wu [MSW20] to show that VDFs, satisfying a stronger form of security, cannot be constructed from random oracles alone either. Rotem, Segev, and Shahaf [RSS20] rule out constructing delay functions from known-order cyclic groups.

On the positive side, both the work of Katz, Loss, and Xu [KLX20] and that of Rotem and Segev [RS20] provide evidence suggesting that speeding up the approach sequential squaring for performing exponentiations in certain groups of unknown order is as hard as factoring.

Time-Based Cryptography From Weaker Assumptions. Bitansky et al. [BGJ+16] showed that time-lock puzzles can be constructed from standard cryptographic tools and the existence of languages that in the worst case are decidable by polynomial size circuits, but not by low depth circuits. In combination with one-way functions, their result allowed for constructing a weak form of time-lock puzzles. In combination with indistinguishability obfuscation, their result allowed for constructing standard time-lock puzzles. Jaques, Montgomery, and Roy [JMR20] then extended the result of Bitansky et al. to the setting of VDFs by showing that these can be constructed from an object they called iteratively sequential functions (ISFs), circular-secure FHE, and some appropriate notion of a proof system.

Looking ahead, our construction of delay functions will closely follow the approach of Bitansky et al., but by focusing on delay functions, instead of time-lock puzzles, we get away with just using one-way functions and do not need to rely on indistinguishability obfuscation. In comparison to Jaques, Montgomery, and Roy, our assumptions are weaker, as we do not need to rely on the existence of ISFs or FHE.

Randomness Beacons. Several previous works have considered both coin flipping protocols and randomness beacons from time-based cryptography. For a general overview of all popular approaches to constructing randomness beacons, we refer the reader to the work of Choi, Manoj, and Bonneau [CMB23]. Here we focus on those that are most relevant to our work.

As already mentioned, Boneh and Naor [BN00] showed that time-based cryptography can allow for two-party coin flipping protocols in the presence of one corruption. Freitag et al. [FKPS21] showed how to construct multiparty coin flipping protocols in the presence of a dishonest majority from time-based cryptography. Baum et al. [BDD+23] define a notion of randomness beacons in the universal composability framework of Canetti [Can01] and present a construction thereof. All of these works focus on the setting, where parties interact over multiple rounds with each other,

meaning that parties speak more than once. While these protocols do tolerate malicious parties not participating or speaking only once, they do require the honest parties to speak multiple times.

Boneh et al. presented a randomness beacon as an application of VDFs [BBBF18]. Their beacon is designed to run parallel to a blockchain, using VDF evaluated on the latest block hash. While they gave no security argument, informally security should require frequent enough honest blocks so that the block hash is not predictable too long in advance.

2 Technical Overview

In this section, we will discuss the main ideas underlying the different constructions in this work.

2.1 Randomness Beacons from Time-Lock Puzzles

Recall that time-lock puzzles are a primitive that allows for hiding a message for a limited amount of time. It is always possible to solve a given puzzle and retrieve the secret message, but doing so requires performing a non-parallelizable, high-depth computation.

Unbiased sampling using time-lock puzzles. We build our randomness beacon in the blockchain setting and we make the following assumption: the honest parties publish sufficiently often on the blockchain that there exists at least one honest block in every subsequence of T (or $T(\lambda)$ for security parameter λ) consecutive blocks. A similar assumption is required for the randomness beacon of [BBBF18].

Our idea is to publish a time-lock puzzle hiding a random λ-bit message in each block. Then, at each step in time, we obtain a fresh beacon output by querying the concatenation of the solutions of the last T published time-lock puzzles to a random oracle. We set the parameters of the puzzles so that no PPT adversary is able to solve them before $T-1$ new blocks are published on the blockchain. In this way, we ensure that the adversary cannot bias the outputs of the beacon. Even if the adversary publishes $T-1$ consecutive malicious blocks after the publication of an honest block, it will have no idea of how its choices affect the beacon outputs. To get a better intuition, imagine an adversary that after seeing an honest block, regenerates $T-1$ random blocks in its head many times, hoping to find a combination that sets the first bit of the corresponding beacon output to 0. With high probability, the adversary will find at least one of these combinations, but it will not have time to recognise it, since that would require solving the puzzle in the last honest block. By the time that the adversary solves that honest puzzle, T-1 new blocks will already be posted, so it cannot make any decisions based on the solution. All outputs depending on this puzzle will already be fixed, and future outputs will depend on new honest puzzles. To summarise, for the adversary to bias the beacon output, it must generate a block based on the result of a long computation, so long that it is impossible to complete in time to generate the block.

Fast verification. Currently, retrieving any beacon output requires each party to solve T time-lock puzzles and while this cost can be amortised to one puzzle per output, it remains a high-depth computation.

Suppose that a party solved all the puzzles necessary to compute a beacon output, how can it quickly convince everybody else of the validity of the result it obtained? In particular, how can it succeed in this without requiring anybody else to solve any puzzle?

A possible solution to this problem is to rely on SNARKs, but there is a simpler solution. We augment each block with a commitment to the value that was originally hidden in the time-lock puzzle. We hide the opening information of this commitment inside the time-lock puzzle (along with the secret bit string). Finally, we ensure that commitment and puzzle are consistent by relying on a (simulation-extractable) NIZK.

With these changes, whenever we solve a time-lock puzzle, we obtain an opening of the commitment. In order to quickly convince the other participant of the validity of a beacon output, it is sufficient to broadcast the openings hidden in the relative puzzles. At that point, verifying the output is only a matter of checking T NIZKs and T commitment openings.

A more efficient construction based on homomorphic time-lock puzzles. Although the amortised cost of our randomness beacon is one time-lock puzzle resolution per beacon output, obtaining a single beacon output requires solving T puzzles. We show that, by relying on linearly homomorphic time-lock puzzle, the non-amortised cost can be lowered to that of a single puzzle resolution.

A linearly homomorphic time-lock puzzle [MT19] consists of a time-lock puzzle scheme where the hidden messages belong to a ring $\mathbb{Z}_q$. The puzzles are additively homomorphic in the sense that we can quickly combine (i.e. performing only low-depth computations) two puzzles hiding messages x_1 and x_2 into a third puzzle hiding $x_1 + x_2$. Malavolta and Thyagarajan [MT19] show how to build homomorphic time-lock puzzle over $\mathbb{Z}_N$, where N is an RSA modulus that needs to be generated via a trusted setup. By lifting the techniques of Malavolta and Thyagarajan to class groups, we obtain homomorphic time-lock puzzles over $\mathbb{Z}_q$ for any prime q. These puzzles do not require a trusted setup.

We modify our randomness beacon by using a linearly homomorphic time-lock puzzle. In particular, the secrets will now belong to the ring $\mathbb{Z}_q$. Instead of generating the beacon outputs by querying a random oracle with the concatenation of the puzzle solutions, we will query their sum (ignoring any duplicates among the T considered puzzles). In order to obtain any beacon, it is therefore sufficient to combine T puzzles and solve the resulting object. In other words, we need to solve a single puzzle.

In order to allow the party solving the puzzle combination quickly to convince other participants of the result, we rely on a commitment scheme that is additively homomorphic over $\mathbb{Z}_q$ for both secrets and openings. In other words, it is possible to combine two commitments with openings (s_1, v_1) and (s_2, v_2) in $\mathbb{Z}_q \times \mathbb{Z}_q$ (where s_1 and s_2 are the secrets), into a third commitment with opening $(s_1 + s_2, v_1 + v_2)$. An example of such commitment schemes are Pedersen commitments over multiplicative groups of order q.

To summarise, in order to obtain a beacon output, it is sufficient for one party to solve a single time-lock puzzle, obtained by combining T puzzles published on the chain. In this way, the party will obtain the sum of the openings of the corresponding T commitments. With such information, the party can quickly convince all other participants of the validity of its computations. In particular, the verification just requires checking T NIZKs and the opening of the combination of the T commitment published on the chain.

Decreasing the delay. How much time goes by between learning a beacon output and the publication of the last block upon which it depends? We call this quantity *the delay of the beacon*. Currently, it is the time necessary to solve a time-lock puzzle. We recall that the parameters of the puzzles are set so that their resolution takes long enough that at least $T - 1$ new blocks are published in the meantime.

Can we make the delay smaller without sensibly decreasing the efficiency of the construction?

Yes we can. Our idea is to modify our original scheme so that instead of having a single puzzle per block, we have T puzzles of decreasing complexity hiding T independent messages. In particular, the first puzzle preserves the privacy of its message long enough to guarantee that the next $T-1$ blocks on the chain will be computationally independent of its secret. The second puzzle will have similar guarantees but only for the next $T-2$ blocks, and so on. In order to obtain the beacon output derived from a subsequence of T consecutive blocks $B_1, \ldots, B_T$, where B_1 is the oldest block and B_T the youngest, we take the element in the first puzzle in B_1, we add the element in the second puzzle of B_2, and so on, until we add the element in the T-th puzzle of B_T. We query the result to the random oracle.

With these modifications, the delay of our randomness beacon depends only on the security-gap of the time-lock puzzle scheme we adopt. Let $t(\tau)$ be the time required to solve a time-lock puzzle guaranteeing security against adversaries of depth τ, we define the security gap as $t(\tau) - \tau$. In particular, if the gap was close to 0, the delay of the randomness beacon would also be close to 0. As soon as any block is published, we can immediately start solving all its puzzles. By doing so, we will finish solving all puzzles upon which a beacon output rely almost simultaneously, not long after the last of these puzzles is published.

Observe that in order to preserve the quick verifiability of the outputs, each block will now include T homomorphic commitments, one for each value hidden in the puzzles.

Batched homomorphic time-lock puzzles. It would seem that with the changes we just introduced, yes, we managed to reduce the delay, but at the same time, we again increased the amount of necessary computations. It would seem that each beacon output requires the resolution of T puzzles. We show that if we rely on particular type of homomorphic time-lock puzzles, which we called *batched homomorphic time-lock puzzles*, this issue can be avoided. The computations necessary for obtaining each beacon output are essentially as expensive as the resolution of the most complex puzzle. We explain how this is possible by presenting how to construct the primitive we need.

Everything starts once again from the construction of Malavolta and Thyagarajan. They construct their homomorphic time-lock puzzles over a group G that can be decomposed as the direct product of F and H, where F is cyclic, generated by an element f of known order q and discrete logarithms can be efficiently computed over it. On the other hand, the order of H is unknown. In [MT19], the group G corresponds to the Paillier group, however, similarly to [ADOS22], we observe that also class groups satisfy all the desired properties. In order to generate a puzzle hiding a message $m \in \mathbb{Z}_q$, we take an element $g \in G$, we sample a random integer r and we output $(x, y) := (g^r, f^m \cdot g^{r \cdot 2^t})$, where t is a parameter of the puzzle, describing its complexity. It is possible to retrieve m from (x, y) in depth $O(t)$, by computing the discrete logarithm of $y \cdot x^{-2^t}$. However, assuming that the exponentiation over G is "inherently sequential", we can hope for (x, y) to preserve the privacy of m against low-depth adversaries.

Now, suppose that we have T time-lock puzzles $(x_1, y_1), \ldots, (x_T, y_T)$ with decreasing complexity parameters $t_1 \geq t_2 \geq \cdots \geq t_T$, hiding messages $m_1, \ldots, m_T$. In order to retrieve $m_1 + \cdots + m_T$, we need to compute the discrete logarithm of $(y_1 \cdots y_T) \cdot (x_1^{-2^{t_1}} \cdots x_T^{-2^{t_T}})$. Our main observation is that $x_1^{2^{t_1}} \cdots x_T^{2^{t_T}}$ can be computed in depth $O(t_1)$: we start by deriving $a_1 := x_1^{2^{t_1 - t_2}}$ in depth $O(t_1 - t_2)$. Next, when we obtain (x_2, y_2) (i.e. when it gets published on the chain), we compute $a_2 := (a_1 \cdot x_2)^{2^{t_2 - t_3}}$ in depth $O(t_2 - t_3)$. We continue in this way until we obtain (x_T, y_T). At that point, we derive $x_1^{2^{t_1}} \cdots x_T^{2^{t_T}}$ by calculating $(a_{T-1} \cdot x_T)^{2^{t_T}}$ in depth $O(t_T)$.

2.2 Delay Functions from Weaker Assumptions

Delay functions guarantee that evaluation on random inputs is slow, i.e. their computation is highly non-parallelizable, it can only be performed by high-depth circuits. We define their security by asking that the output of their evaluation on random inputs should be unpredictable to any low-depth adversary. Delay functions can be quickly compiled into verifiable delay functions by relying on an appropriate succinct proof system.[4]

A delay function based on garbled circuits. We show how to build delay functions from minimal assumptions: the existence of one-way functions and the existence of a language $L \in \mathsf{P}$ with a decider D of depth d' but no decider of depth $d < d'$. Our delay function is described by an unstructured bit string, so it can, for instance, be generated by a random oracle. Its inputs are unstructured strings as well. In order to evaluate the delay function, we regard the random string in its description as the point-and-permute garbling of a universal circuit outputting a single bit [Yao86, Rog91, BHR12]. We regard the input of the delay function as the list of input labels for such garbled circuit. The output will consist of the output label obtained by evaluating the "garbled circuit" on the given "input labels". We stress that this is not a real garbled circuit evaluation, because in reality we are just dealing with random strings. The crucial reason why this operation still succeeds is that, in point-and-permute garbling, the encrypted circuit is indistinguishable from a random scheme even if we provide the encoding of an input (in order for this property to hold, we need to keep the output labels secret). We call this property *evaluation-obliviousness.*

Why is this secure? We observe that any low-depth adversary with the ability of predicting the output of this delay function can be converted in a low-depth decider for the language L. Indeed, the adversary would still be able to predict the output of the delay function even if we substituted the random strings describing the function and its input with an actual garbling of the universal circuit and the input labels corresponding to the pair (D, x) where x is the candidate member of L. In this new setting, the delay function evaluation would return the output label corresponding to 1 if and only if $x \in L$.

We use this to obtain a low-depth decider for L that relies on auxiliary randomness[5], namely a garbling of the universal circuit and the relative encoding and decoding information. Upon receiving a candidate language member x, the decider provides the low-depth adversary with the garbled circuit and the encoding of (D, x). Then, it outputs the decoding of the string returned by the adversary. If the decoding procedure fails, the decider outputs a random bit.

This randomised decider correctly classifies any input x with probability that is noticeably away from $1/2$. We decrease the error probability to negligible by outputting the most common outcome among multiple parallel executions of the randomised decider. Finally, we derandomise the algorithm by relying on an averaging argument. Notice that the depth of the resulting decider is larger than the adversary's depth only by a $O(\log \lambda)$ amount (due to the depth of the encoding and decoding procedures in garbled circuits and the computation of the most common outcome of

[4] Doing this requires a little bit of care actually. Naively one would like to simply construct a VDF by using a proof system for the statement "y is the output of delay function f on input x.", but this does not necessarily work. The size of description of the delay function, which is an input to the VDF verifier, may be proportional to the required evaluation depth, thereby making the verifier not succinct. To get around this problem, however, one can simply add the hash h of the function's description and modify the statement to "y is the output of some delay function f on input x and $h = \text{hash}(f)$".

[5] The decider receive the auxiliary randomness together with the input.

the parallel executions of the randomised decider). Since L has no decider of depth d, our delay function guarantees security against adversaries of depth $d - \omega(\log \lambda)$.

Why not to directly garble the decider D? In principle, our construction would have been secure even if we regarded the random string describing the delay function as the garbling of a circuit topologically equivalent to D (two circuits are topologically equivalent if their underlying directed graphs coincide). So, why did we instead rely on a universal circuit?

The reason is that, in this way, our construction is secure under a weaker assumption, namely the mere existence of a language L having a decider that fits into the considered universal circuit U, but having no decider of depth d. On the other hand, if we regarded the random string in the delay function description as the garbling of a circuit of any other topology, the construction would have been secure only if there exists a language having a decider of the given topology, but having no decider of depth d. Notice that if a circuit C fits into the universal circuit U, then also any other circuit topologically equivalent to C fits into U. In other words, by relying on universal circuits, we are considering a broader class of languages, increasing the chances that one of them cannot be decided in depth d.

Finally, we recall that Valiant showed how to build a universal circuit of size $O(s \cdot \log s)$ and depth $O(s)$ accepting as input any circuits of size s [Val76]. If we rely on such construction, our delay function can be evaluated in depth $O(s)$.

2.3 Delay Encryption

Delay encryption [BD21] allows for quickly, i.e. via a low-depth computation, encrypting a message under a random public label, guaranteeing its privacy for a limited amount of time. Concretely, each label $\mathtt{id}$ is associated with a secret key that allows for quick decryption of any ciphertext produced under $\mathtt{id}$. Deriving the secret key associated with $\mathtt{id}$ can be done in polynomial time, but requires a high-depth computation.

Let us start by observing that building delay encryption from obfuscation is trivial. The public parameters would consist in an obfuscated program that, on input a label $\mathtt{id}$, outputs a pseudo-random public key $\mathsf{pk}_{\mathtt{id}}$ and a time-lock puzzle hiding the corresponding secret key. In order to encrypt a message m under the label $\mathtt{id}$, it is sufficient to compute $\mathsf{Enc}(\mathsf{pk}_{\mathtt{id}}, m)$. Performing the decryption would require the resolution of the time-lock puzzle containing the secret key. For this reason, the focus of our work lies on building delay encryption outside of obfustopia.

Programmable delay functions. We present two blueprints for building delay encryption. The first is based on witness encryption and the second one on attribute-based encryption. Independently of the path we take, we rely on delay functions satisfying a particular property we call *programmability*.

A programmable delay function is a function for which we can program its evaluation on a random input. Specifically, given any value y, we can generate the parameters of the function along with a random-looking input x, so that x is mapped to y. We require that the programmed function looks the same as a normal function. Additonally, we require that indistinguishability holds against low-depth adversaries, even if we leak x. In particular, this means that the depth of the adversary should be too low to evaluate the function on x.

We show that the delay functions we built using garbled circuits are programmable. Given a value $y \in \{0,1\}^\lambda$, we generate the description of the function by garbling the universal circuit U using y as the output label associated with the outcome 1. The input x that maps to y will be the

encoding of any string v such that $U(v) = 1$. Notice that evaluating the garbling on x, still requires a high-depth computation.

Delay functions from witness encryption. Our idea is fairly simple. The public parameters of our construction will consist of the description of a programmable delay function. In order to encrypt a message m under a label $\mathtt{id}$, we encrypt it using witness encryption. The corresponding statement claims that the evaluation of the delay function on $\mathtt{id}$ is different from 0. Given that the outputs of the delay function are unpredictable, the claim will be true with overwhelming probability. Importantly, a low-depth adversary will not be able to notice if we provide it with a random looking label $\mathtt{id}$ on which the delay function was programmed to output 0. In such setting, plaintext privacy is ensured by the security of witness encryption.

As for efficiency, we would like that encryption and decryption (assuming the knowledge of a witness) are low-depth operations. Luckily, in the witness encryption construction from evasive LWE of Vaikuntanathan, Wee and Wichs [VWW22], both the operations lie in NC_1. On the other hand, however, the ciphertext size scales linearly in the depth of the delay function.

Delay functions from attribute-based encryption. We follow the same blueprint as before. The public parameters of our construction will consist of the description of a programmable delay function and an ABE secret key sk, specifically, the one corresponding to the predicate

the evaluation of the delay function on the attribute is different from 0.

Therefore, in order to generate a ciphertext under a random label $\mathtt{id}$, we just encrypt the message using $\mathtt{id}$ as an attribute.

Correctness is guaranteed once again by the unpredictability of the delay function. Most likely its output will be different from 0, so sk allows us to decrypt. Security follows instead from the fact that a low-degree adversary cannot detect whether we provide it with a label $\mathtt{id}$ on which the delay function was programmed to output 0. In such case, privacy is ensured by the security of ABE.

What is, however, the secret key associated with $\mathtt{id}$ that allows for quick decryption? In order to answer this question, we require an additional property from the ABE scheme. We ask that given sk, the label $\mathtt{id}$ and the master public key of the scheme (but no information about the ciphertext), we can derive a label-specific key $\mathsf{sk}_{\mathtt{id}}$ that allows for quick decryption. We show that the LWE-based ABE scheme of Boneh et al. [BGG+14] satisfies this property. Encryption and decryption (once given $\mathsf{sk}_{\mathtt{id}}$) lie in NC_1. Unfortunately, however, their scheme does not satisfy all the efficiency requirements we impose upon the needed ABE. Their ciphertext scales linearly with the depth of the delay function. A recent work by Hsieh, Lin, and Luo [HLL23] shows how to construct an ABE scheme with a ciphertext size that does not grow with the depth of f, but to the best of our knowledge their construction does not allow for the type of quick decryption we need. We leave constructing a sufficiently succinct ABE with our quick decryption property as an exciting open problem.

3 Preliminaries

Throughout this paper we will focus on algorithms and adversaries that are represented as circuits. This allows us to make precise statements about the depth and parallelism of any given computation. A natural question to ask is whether our results are less general than they could be, as we will be only considering adversaries that are circuits and not, for instance, Turing machines. We note that

this is not the case, since any T-step Turing machine can be simulated by a $\mathcal{O}(T \cdot \lg T)$ circuit, as was shown by Pippenger and Fischer [PF79].

Definition 1 (Circuit Classes). *A circuit class is a set $\mathcal{C}$ of sequences $(C_\lambda)_{\lambda\in\mathbb{N}}$ where C_λ is a circuit for every $\lambda \in \mathbb{N}$.*

Let $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$ be a sequence of sets of circuits. The product circuit class induced by $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$ is the circuit class of all sequences $(C_\lambda)_{\lambda\in\mathbb{N}}$ where $C_\lambda \in \mathcal{C}_\lambda$ for every $\lambda \in \mathbb{N}$.

Definition 2 (Nicely Closed Circuit Class). *We say that a circuit class $\mathcal{C}$ is nicely closed if the following properties are satisfied:*

- ***(Constant Sequentiality.)*** *For every $(C_\lambda)_{\lambda\in\mathbb{N}}, (C'_\lambda)_{\lambda\in\mathbb{N}} \in \mathcal{C}$ such that the input length of C'_λ and the output length of C_λ coincide for every $\lambda \in \mathbb{N}$, the sequence $(C_\lambda \diamond C'_\lambda)_{\lambda\in\mathbb{N}}$, where $C_\lambda \diamond C'_\lambda$ denotes the sequential composition of C_λ and C'_λ (i.e. we feed the outputs of C_λ into C'_λ), still belongs to $\mathcal{C}$.*
- ***(Polynomial Parallelisation.)*** *For every $(C_\lambda)_{\lambda\in\mathbb{N}} \in \mathcal{C}$ and polynomial function $T(\lambda)$, the sequence $(C_\lambda^{\otimes T(\lambda)})_{\lambda\in\mathbb{N}}$, where $C_\lambda^{\otimes T(\lambda)}$ denotes the parallel composition of $T(\lambda)$ copies of C_λ, still belongs to $\mathcal{C}$.*
- ***(Input Fixing).*** *For every $(C_\lambda)_{\lambda\in\mathbb{N}} \in \mathcal{C}$, sets $(B_\lambda)_{\lambda\in\mathbb{N}}$ and functions $(f_\lambda : B_\lambda \to \{0,1\})_{\lambda\in\mathbb{N}}$ where B_λ is a subset of the input wires of C_λ, the sequence $(C_\lambda[B_\lambda, f_\lambda])_{\lambda\in\mathbb{N}}$, where $C_\lambda[B_\lambda, f_\lambda]$ denotes the circuit obtained by assigning the value $f_\lambda(w)$ to every input wire $w \in B_\lambda$ of C_λ and simplifying the result (i.e, we eliminate the gates where all inputs have been fixed), still belongs to $\mathcal{C}$.*
- ***(Asymptoticity).*** *For any sequences $(C_\lambda)_{\lambda\in\mathbb{N}}$ and $(C'_\lambda)_{\lambda\in\mathbb{N}}$ such that there exists a $\overline{\lambda} \in \mathbb{N}$ such that $C_\lambda = C'_\lambda$ for every $\lambda \geq \overline{\lambda}$, if $(C_\lambda)_{\lambda\in\mathbb{N}} \in \mathcal{C}$, then $(C'_\lambda)_{\lambda\in\mathbb{N}} \in \mathcal{C}$.*

Definition 3. *Let $f(\lambda)$ be a function of the security parameter. We define the circuit class $\mathcal{C}_f$ as follows*

$$\mathcal{C}_f := \{(C_\lambda)_{\lambda\in\mathbb{N}} \mid \exists c \in \mathbb{N} \text{ such that } \mathsf{depth}(C_\lambda) \leq c \cdot f(\lambda) \ \ \forall \lambda \in \mathbb{N}\}.$$

Similarly, we define the class $\overline{\mathcal{C}}_f$ as follows

$$\overline{\mathcal{C}}_f := \{(C_\lambda)_{\lambda\in\mathbb{N}} \mid \mathsf{depth}(C_\lambda) \leq f(\lambda) \ \ \forall \lambda \in \mathbb{N}\}.$$

Lemma 1. *Let $f(\lambda)$ be a function of the security parameter. Then, the circuit class $\mathcal{C}_f$ is nicely closed.*

Observe that $\overline{\mathcal{C}}_f$ is a product class but it is not nicely closed.

Definition 4 ($\mathcal{C}$-Solvable Language). *Let $\mathcal{C}$ be a circuit class. We use $\mathcal{L}^{\mathcal{C}}$ to denote the set of all languages that can be decided by an element in $\mathcal{C}$, i.e. all languages $L = \bigcup_{\lambda\in\mathbb{N}} L_\lambda$ for which there exist a function $\ell : \mathbb{N} \to \mathbb{N}$ and $(C_\lambda)_{\lambda\in\mathbb{N}} \in \mathcal{C}$ such that*

- *For every $\lambda \in \mathbb{N}$ and $x \in L_\lambda$, we have $|x| = \ell(\lambda)$*
- *For every $\lambda \in \mathbb{N}$ and $x \in \{0,1\}^{\ell(\lambda)}$, we have $x \in L$ if and only if $C_\lambda(x) = 1$*

In such case, we say that $(C_\lambda)_{\lambda\in\mathbb{N}}$ is a decider for L.

Definition 5 (Infinitely Often $\mathcal{C}$-Solvable Language). *Let $\mathcal{C}$ be a circuit class. We use $\widetilde{\mathcal{L}}^{\mathcal{C}}$ to denote the set of all languages $L = \bigcup_{\lambda\in\mathbb{N}} L_\lambda$ for which there exist a function $\ell : \mathbb{N} \to \mathbb{N}$, a subset $S \subseteq \mathbb{N}$ of infinite cardinality and $(C_\lambda)_{\lambda\in\mathbb{N}} \in \mathcal{C}$ such that*

- *For every $\lambda \in \mathbb{N}$ and $x \in L_\lambda$, we have $|x| = \ell(\lambda)$*
- *For every $\lambda \in S$ and $x \in \{0,1\}^{\ell(\lambda)}$, we have $x \in L$ if and only if $C_\lambda(x) = 1$*

3.1 Time Lock Puzzles

We now recall the definition of time-lock puzzles, a primitive introduced for the first time in 1996 by Rivest, Shamir and Wagner [RSW96]. A time-lock puzzle consists of an object hiding a message m. Although the message can be retrieved in polynomial time, the operation is slow: it can only be computed by high-depth circuits.

We start by formalising the syntax and the efficiency notions. Notice that in order for a construction to be interesting, we would like the generation of the puzzle to be quick (i.e. low-depth).

Definition 6 (Time-lock puzzle). *A time lock puzzle scheme with domain $\mathcal{X} = (\mathcal{X}_\lambda)_{\lambda\in\mathbb{N}}$ consists of a triple of algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve})$ *with the following syntax:*

$\mathsf{Setup}(1^\lambda, 1^\tau)$**:** *The algorithm takes as input the security parameter and a time parameter 1^τ. The output consists of public parameters* pp.

$\mathsf{Gen}(1^\lambda, \mathsf{pp}, m)$**:** *The algorithm takes as input the security parameter 1^λ, public parameters* pp *and a message $m \in \mathcal{X}_\lambda$. The output is a puzzle z.*

$\mathsf{Solve}(\mathsf{pp}, z)$**:** *The algorithm is deterministic and takes as input the public parameters* pp *and a puzzle z. The output is a message m.*

Definition 7 (Efficiency of time-lock puzzles). *Let $\mathcal{C}_0, \mathcal{C}_1 \subseteq \mathsf{poly}(\lambda)$ be circuit classes. We say that a time-lock puzzle scheme* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve})$ *is $(\mathcal{C}_0, \mathcal{C}_1)$-efficient if $\mathsf{Gen} \in \mathcal{C}_0$, $\mathsf{Solve} \in \mathcal{C}_1$ and $\mathsf{Setup} \in \mathsf{poly}(\lambda)$.*

Next, we recall correctness: the property states that if we solve a puzzle hiding a message m, we indeed retrieve m.

Definition 8 (Correctness of time-lock puzzles). *A time-lock puzzle scheme* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve})$ *with domain $\mathcal{X} = (\mathcal{X}_\lambda)_{\lambda\in\mathbb{N}}$ is correct if, for every $\lambda, \tau \in \mathbb{N}$, message $m \in \mathcal{X}_\lambda$ and randomness $r \in \{0,1\}^*$, it holds that*

$$\Pr\left[\mathsf{pp} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, 1^\tau) \quad z \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda, \mathsf{pp}, m; r) : \mathsf{Solve}(\mathsf{pp}, z) = m\right] = 1.$$

Finally, we formalise security: a time-lock puzzle scheme is secure against the sequence of circuit classes $(\mathcal{C}_\tau)_{\tau\in\mathbb{N}}$, if the privacy of the message hidden in a puzzle generated with time parameter τ is preserved against all adversaries in $\mathcal{C}_\tau$. This is formalised by means of a IND-CPA-like security game.

Definition 9 (Security of time-lock puzzles). *Let $\mathcal{C}_\tau = (\mathcal{C}_{\tau,\lambda})_{\lambda\in\mathbb{N}}$, where $\tau = \tau(\lambda) \in \mathbb{N}$, be a product circuit class. Let $\mathcal{C}$ be the sequence $(\mathcal{C}_\tau)_{\tau\in\mathbb{N}}$. We say that an adversary $\mathcal{A}$ is $(\mathsf{TLP}, \tau, \mathcal{C})$-respecting if, for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\begin{bmatrix}\mathsf{pp} \leftarrow\!\!\$\ \mathsf{TLP.Setup}(1^\lambda, 1^{\tau(\lambda)}) & \\ (m_0, m_1, \mathcal{A}') \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, \mathsf{pp}) & \end{bmatrix} : \mathcal{A}' \in \mathcal{C}_{\tau(\lambda),\lambda} \Bigg] = 1.$$

Let $\overline{\tau}(\lambda)$ be a polynomial function of the security parameter. A time-lock puzzle scheme $\mathsf{TLP} = (\mathsf{Setup}, \mathsf{Gen.Solve})$ is $(\mathcal{C}, \overline{\tau})$-secure if, for every polynomial function $\tau = \Omega(\overline{\tau}(\lambda))$ and $(\mathsf{TLP}, \tau, \mathcal{C})$-respecting adversary $\mathcal{A} \in \mathsf{poly}(\lambda)$, it holds that

$$\mathsf{Adv}^{\mathsf{TLP\text{-}SEC},\tau}_{\mathsf{TLP}}(\mathcal{A}) := \left|\Pr[\mathsf{Exp}^{\mathsf{TLP\text{-}SEC},\tau}_{\mathcal{A},\mathsf{TLP}}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$$

where experiment $\mathsf{Exp}^{\mathsf{TLP\text{-}SEC},\tau}_{\mathcal{A},\mathsf{TLP}}$ is specified in Figure 1.

$\mathsf{Exp}^{\mathsf{TLP\text{-}SEC},\tau}_{\mathcal{A},\mathsf{TLP}}(1^\lambda)$

1 : $\mathsf{pp} \leftarrow\$ \mathsf{TLP.Setup}(1^\lambda, 1^{\tau(\lambda)})$

2 : $(m_0, m_1, \mathcal{A}') \leftarrow\$ \mathcal{A}(1^\lambda, \mathsf{pp})$

3 : $b \leftarrow\$ \{0,1\}$

4 : $z \leftarrow\$ \mathsf{TLP.Gen}(1^\lambda, \mathsf{pp}, m_b)$

5 : $b' \leftarrow\$ \mathcal{A}'(z)$

6 : **return** $b = b'$

Fig. 1. Time-lock puzzle security game TLP-SEC.

Linearly homomorphic time-lock puzzles. In [MT19], Malavolta and Thyaragajan showed how to construct linearly homomorphic time-lock puzzles schemes: given a linear function $f : \mathbb{Z}_q^M \to \mathbb{Z}_q$, it is possible to quickly combine M time-lock puzzles hiding messages $x_1, \ldots, x_M$, into a single time-lock puzzle hiding $f(x_1, \ldots, x_M)$. In other words, in order to obtain $f(x_1, \ldots, x_M)$ it is sufficient to solve a single puzzle instead of M.

Definition 10 (Linearly homomorphic time-lock puzzle). *Let $\mathcal{C}$ be a circuit class. A time-lock puzzle* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve})$ *is $\mathcal{C}$-efficient $\mathbb{Z}_q$-linearly homomorphic if there exists a PPT algorithm* $\mathsf{Eval} \in \mathcal{C}$ *such that, for every $\lambda, \tau \in \mathbb{N}$, public parameters* pp *in the domain of* $\mathsf{Setup}(1^\lambda, 1^\tau)$*, linear function $f : \mathbb{Z}_q^M \to \mathbb{Z}_q$, elements $x_1, \ldots, x_M \in \mathbb{Z}_q$ and random strings $r_1, \ldots, r_M \in \{0,1\}^*$, there exists a string $r \in \{0,1\}^*$ such that*

$$\mathsf{Eval}(\mathsf{pp}, f, z_1, \ldots, z_M) = \mathsf{Gen}(1^\lambda, \mathsf{pp}, f(x_1, \ldots, x_M); r)$$

where, for every $i \in [M]$, we define z_i as the output of $\mathsf{Gen}(1^\lambda, \mathsf{pp}, x_i; r_i)$.

Notice that since we are relying on a perfectly correct time-lock puzzle, by solving $z := \mathsf{Eval}(\mathsf{pp}, f, z_1, \ldots, z_M)$, we obtain $f(x_1, \ldots, x_M)$ with probability 1. Observe also that if we defined the homomorphic properties of time-lock puzzles by just requiring this property, i.e. that the resolution of the output of Eval returns $f(x_1, \ldots, x_M)$, we would have obtained a weaker definition: there would be no guarantee that the output of the evaluation can be the input for another evaluation!

3.2 Delay Encryption

Next, we recall the definition of delay encryption, formalised for the first time by Burdges and De Feo [BD21]. In their work, they indirectly define delay encryption by defining an appropriate key encapsulation mechanism. In our work, we prefer directly focusing on delay encryption as is own primitive.

Let us start by defining the syntax and efficiency.

Definition 11. *A delay encryption scheme for message space $\mathcal{M} := (\mathcal{M}_\lambda)_{\lambda \in \mathbb{N}}$ and identity space $\mathcal{ID} := (\mathcal{ID}_\lambda)_{\lambda \in \mathbb{N}}$ is a tuple of algorithms* $\mathsf{E} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ext})$ *defined as follows:*

$\mathsf{pp} \leftarrow\$ \mathsf{Setup}(1^\lambda)$**:** *The setup algorithm takes security parameter λ as input and returns public parameters* pp.

$\mathtt{idk} \leftarrow \mathsf{Ext}(\mathtt{pp}, \mathtt{id})$: *The extraction algorithm takes public parameters* $\mathtt{pp}$ *and an identity* $\mathtt{id} \in \mathcal{ID}_\lambda$ *as input and returns an identity decryption key* $\mathtt{idk}$.

$c \leftarrow\$ \mathsf{Enc}(\mathtt{pp}, \mathtt{id}, m)$: *The encryption algorithm takes public parameters* $\mathtt{pp}$*, an identity* $\mathtt{id} \in \mathcal{ID}_\lambda$*, and a message* $m \in \mathcal{M}_\lambda$ *as input and returns a ciphertext* c.

$m \leftarrow \mathsf{Dec}(\mathtt{pp}, \mathtt{idk}, c)$: *The decryption algorithm takes public parameters* $\mathtt{pp}$*, an identity decryption key* $\mathtt{idk}$ *and a ciphertext* c *as input and returns a message* m.

Definition 12 (Efficiency). *Let* $\lambda \in \mathbb{N}$*. We say a delay encryption scheme* $\mathsf{E} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ext})$ *is* $\mathcal{C}$*-efficient, if* $\mathsf{Enc}, \mathsf{Dec} \in \mathcal{C}$ *and* $\mathsf{Setup}, \mathsf{Ext} \in \mathsf{poly}(\lambda)$.

Correctness states that if we decrypt a ciphertext generated under a label $\mathtt{id}$ using the identity key $\mathtt{idk}$ produced by the extraction algorithm Ext, we always recover the plaintext.

Definition 13 (Correctness). *We say a delay encryption scheme* $\mathsf{E} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ext})$ *for message space* $\mathcal{M} := (\mathcal{M}_\lambda)_{\lambda \in \mathbb{N}}$ *and identity space* $\mathcal{ID} := (\mathcal{ID}_\lambda)_{\lambda \in \mathbb{N}}$ *is correct if, for every sequence* $(m_\lambda)_{\lambda \in \mathbb{N}}$ *such that* $m_\lambda \in \mathcal{M}_\lambda$ *for every* $\lambda \in \mathbb{N}$*, it holds that*

$$\Pr\left[\begin{array}{l} \mathtt{pp} \leftarrow\$ \mathsf{Setup}(1^\lambda) \\ \mathtt{id} \leftarrow\$ \mathcal{ID}_\lambda \\ c \leftarrow\$ \mathsf{Enc}(\mathtt{pp}, \mathtt{id}, m_\lambda) \\ \mathtt{idk} \leftarrow \mathsf{Ext}(\mathtt{pp}, \mathtt{id}) \end{array} : \mathsf{Dec}(\mathtt{pp}, \mathtt{idk}, c) = m_\lambda \right] \leq \mathsf{negl}(\lambda).$$

Finally, we recall the definition of security. The latter is defined by means of a IND-CPA-like game against a low-depth adversary. The label under which we generate the ciphertexts is sampled at random.

Definition 14 (Security). *Let* $\lambda \in \mathbb{N}$*. We say a delay encryption scheme* $\mathsf{E} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ext})$ *for message space* $\mathcal{M} := (\mathcal{M}_\lambda)_{\lambda \in \mathbb{N}}$ *and identity space* $\mathcal{ID} := (\mathcal{ID}_\lambda)_{\lambda \in \mathbb{N}}$ *is* D-IND-CPA *secure against adversary class* $\mathcal{C}$*, if for all adversaries* $\mathcal{A} := (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathcal{C}$ *it holds that*

$$\mathsf{Adv}_{\mathsf{E}}^{\mathsf{D\text{-}IND\text{-}CPA}}(\mathcal{A}) := \left| \Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{E}}^{\mathsf{D\text{-}IND\text{-}CPA}}(1^\lambda) = 1] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda),$$

where experiment $\mathsf{Exp}_{\mathcal{A},\mathsf{E}}^{\mathsf{D\text{-}IND\text{-}CPA}}$ *is specified in Figure 2.*

$\mathsf{Exp}_{\mathcal{A},\mathsf{E}}^{\mathsf{D\text{-}IND\text{-}CPA}}(1^\lambda)$

1 : $\mathtt{pp} \leftarrow\$ \mathsf{Setup}(1^\lambda)$
2 : $(\mathtt{aux}, m_0, m_1) \leftarrow\$ \mathcal{A}_0(\mathtt{pp})$
3 : $\mathtt{id} \leftarrow\$ \mathcal{ID}_\lambda$
4 : $b \leftarrow\$ \{0, 1\}$
5 : $c \leftarrow\$ \mathsf{Enc}(\mathtt{pp}, \mathtt{id}, m_b)$
6 : $b' \leftarrow\$ \mathcal{A}_1(\mathtt{aux}, \mathtt{id}, c)$
7 : **return** $b = b'$

Fig. 2. Security game D-IND-CPA.

3.3 Witness Encryption

We recall the definition of witness encryption, introduced for the first time by Garg, Gentry, Sahai and Waters [GGSW13]. Informally, this primitive consists of an encryption scheme where there exist no keys: the message is hidden under a statement for a relation $\mathcal{R}$ and can be retrieved using a corresponding witness. If no such witness exists, the privacy if the plaintexts is guaranteed. We start by recalling the syntax.

Definition 15 (Witness Encryption). *Let $\mathcal{R}$ be a relation in NP. A witness encryption scheme for the relation $\mathcal{R}$ consists of a pair of PPT algorithms $(\mathsf{Enc}, \mathsf{Dec})$ with the following syntax:*

$\mathsf{Enc}(1^\lambda, x, m)$**:** *The algorithm takes as input the security parameter 1^λ, a statement x and a message m. The output is a ciphertext c.*

$\mathsf{Dec}(c, w)$**:** *The algorithm is deterministic and takes as input a ciphertext c and a witness w. The output is a message m.*

Correctness states that if we decrypt a ciphertext under a statement x using a witness for x (i.e. an element w such that $(x, w) \in \mathcal{R}$), we always recover the plaintext.

Definition 16 (Correctness). *Let $\mathcal{R}$ be a relation in NP. A witness encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ for the relation $\mathcal{R}$ satisfies correctness if, for every $\lambda \in \mathbb{N}$ and $(x, w) \in \mathcal{R}$ and message m, it holds*

$$\Pr\left[c \leftarrow\$ \mathsf{Enc}(1^\lambda, x, m) : \mathsf{Dec}(c, w) = m\right] = 1.$$

Finally, we recall the definition of security for witness encryption: a scheme is secure if the privacy of the plaintext is preserved whenever the statement x under which we perform the encryption is false (i.e. there exist no witness w such that $(x, w) \in \mathcal{R}$).

Definition 17 (Security). *Let $\mathcal{R}$ be a relation in NP. We say that an adversary $\mathcal{A} := (\mathcal{A}_0, \mathcal{A}_1)$ is $\mathcal{R}$-consistent if $\mathcal{A}_0$ always outputs a pair (x, m_0, m_1, ψ) where $x \notin L_\mathcal{R}$, m_0 and m_1 are messages, and ψ is an internal state. A witness encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ for $\mathcal{R}$ is secure if, for every $\mathcal{R}$-consistent adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathsf{poly}(\lambda)$, it holds that*

$$\left| \Pr\begin{bmatrix} (x, m_0, m_1, \psi) \leftarrow\$ \mathcal{A}_0(1^\lambda) & \\ b \leftarrow\$ \{0, 1\} & : \mathcal{A}_1(\psi, c) = b \\ c \leftarrow\$ \mathsf{Enc}(1^\lambda, x, m_b) & \end{bmatrix} - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda).$$

3.4 Strongly Homomorphic Commitments

Finally, we recall the definition of strongly homomorphic, non-interactive commitment. Such a commitment scheme allows a party to commit to some chosen value and to reveal it at some later point in time. This is obtained by broadcasting an object, called the *commitment*. The object guarantees the privacy of the chosen value. Furthermore, it bounds the committer to reveal exactly the value it has chosen at the time the commitment was broadcast. If the committer changes its mind, revealing another message, all bystanders will realise that the broadcast value is inconsistent with the previously sent commitment. On the other hand, in order to convince the bystanders of the correct value, the committer need to provide auxiliary information called *opening*, which was generated in conjunction with the commitment.

Now, in a strongly homomorphic commitment scheme both messages and openings lie in a ring $\mathbb{Z}_q$ and it is possible to combine two commitments c_1 and c_2, with openings (s_1, v_1) and (s_2, v_2), where s_1 and s_2 are the hidden messages, into a commitment c_3 with opening $(s_1 + s_2, v_1 + v_2)$.

Definition 18. *A strongly homomorphic commitment consists of a tuple of PPT algorithms* $(\mathsf{D}, \mathsf{Setup}, \mathsf{Commit}, \mathsf{Add}, \mathsf{Check})$ *having the following syntax:*

$\mathsf{D}(1^\lambda)$**:** *The algorithm takes as input the security parameter. The output is a positive integer* $q \in \mathbb{N}$.

$\mathsf{Setup}(1^\lambda, q)$**:** *The algorithm takes as input the security parameter and an integer* q. *The output is a CRS* ω.

$\mathsf{Commit}(\omega, s)$**:** *The algorithm takes as input the CRS* ω *and a message* s. *The output is a commitment* c *and an opening* v.

$\mathsf{Add}(\omega, c_1, c_2)$**:** *The algorithm is deterministic and takes as input the CRS* ω *and two commitments* c_1 *and* c_2. *The output is another commitment* c_3.

$\mathsf{Check}(\omega, c, s, v)$**:** *The algorithm is deterministic and takes as input the CRS* ω, *a commitment* c, *a message* s *and an opening* v. *The output is a bit* b.

For correctness, we require that if a commitment c hides a message s and is generates together with an opening v, the pair (s, v) is always accepted by the verification algorithm. Furthermore, we requite that the combination of L commitments can be opened using the sum of the L openings.

Definition 19 (Correctness). *The strongly homomorphic commitment scheme is correct if the following properties are satisfied*

- *For every* $\lambda \in \mathbb{N}$ *and* $s \in \mathbb{N}$, *we have*

$$\Pr\begin{bmatrix} q \leftarrow\$ \mathsf{D}(1^\lambda) & \\ \omega \leftarrow\$ \mathsf{Setup}(1^\lambda, q) & : \mathsf{Check}(\omega, c, s, v) = 1 \\ (c, v) \leftarrow\$ \mathsf{Commit}(\omega, s) & \end{bmatrix} = 1.$$

- *For every* $\lambda, L \in \mathbb{N}$ *and* $s_1, \ldots, s_L \in \mathbb{N}$, *we have*

$$\Pr\begin{bmatrix} q \leftarrow\$ \mathsf{D}(1^\lambda) & \\ \omega \leftarrow\$ \mathsf{Setup}(1^\lambda, q) & \\ \forall i \in [L] : (c_i, v_i) \leftarrow\$ \mathsf{Commit}(\omega, s_i) & \\ s \leftarrow (s_1 + \cdots + s_L) \bmod q & : \mathsf{Check}(\omega, c, s, v) = 1 \\ v \leftarrow (v_1 + \cdots + v_L) \bmod q & \\ c \leftarrow c_1 & \\ \forall i \in [2..L] : c \leftarrow \mathsf{Add}(\omega, c, c_i) & \end{bmatrix} = 1.$$

Finally, we recall the standard definitions of binding and hiding: if no opening is provided, the commitment hides the message it contains. Furthermore, it is infeasible for polynomial time adversaries to generate commitments that can be opened to different values.

Definition 20 (Binding). *The strongly homomorphic commitment scheme is computationally binding if, for every PPT adversary* $\mathcal{A}$

$$\Pr\begin{bmatrix} q \leftarrow\$ \mathsf{D}(1^\lambda) & \mathsf{Check}(\omega, c, s_1, v_1) = 1 \\ \omega \leftarrow\$ \mathsf{Setup}(1^\lambda, q) & : \mathsf{Check}(\omega, c, s_2, v_2) = 1 \\ (c, s_1, v_1, s_2, v_2) \leftarrow\$ \mathcal{A}(1^\lambda, \omega, q) & s_1 \neq s_2 \end{bmatrix} \leq \mathsf{negl}(\lambda).$$

Definition 21 (Hiding). *The strongly homomorphic commitment scheme is computationally hiding if, for every adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathsf{poly}(\lambda)$

$$\left| \Pr\left[\begin{array}{l} b \leftarrow\$ \{0,1\} \\ q \leftarrow\$ \mathsf{D}(1^\lambda) \\ \omega \leftarrow\$ \mathsf{Setup}(1^\lambda, q) \\ (s_0, s_1, \psi) \leftarrow\$ \mathcal{A}_0(1^\lambda, \omega, q) \\ (c, v) \leftarrow\$ \mathsf{Commit}(\omega, s_b) \end{array} : \mathcal{A}_1(\psi, c) = b \right] - \frac{1}{2} \right| \le \mathsf{negl}(\lambda).$$

3.5 Simulation-Extractable NIZKs

In this subsection, we recall the definition of simulation-extractable NIZKs [GO07]. A NIZK consists of a primitive that allows proving knowledge of a witness w for a statement x in an NP-language L without revealing any additional information. Specifically, let $\mathcal{R}$ be a relation for the language L and imagine that a prover holds a pair $(x, w) \in \mathcal{R}$. The scheme allows generating a proof π with which the prover can convince any external verifier that it knows a witness for x. The proofs reveal no information about the witnesses, not even if we provide many of them for different statements. Furthermore, the prover cannot generate proofs for any statement x without knowing at least one witness: by hiding a trapdoor in the public parameters of the scheme, we can extract a witness from the verifying proofs any malicious prover generates. In other words, if the adversary is able to generate proofs for some statement with non-negligible probability over the randomness of the public parameters, then, there also exists a way for that adversary to derive a witness in polynomial time. Below, we recall the syntax of simulation-extractable NIZKs.

Definition 22 (Simulation-extractable NIZK). *Let* $\mathcal{R}$ *be a relation in* NP*. A simulation-extractable NIZK for* $\mathcal{R}$ *consists of a tuple of algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Ext})$ *with the following syntax:*

$\mathsf{Setup}(1^\lambda)$**:** *The algorithm takes as input the security parameter and outputs a CRS* σ*.*

$\mathsf{Prove}(\sigma, x, w)$**:** *The algorithm takes as input a CRS* σ*, a statement* x *and a corresponding witness* w*. The output is a proof* π*.*

$\mathsf{Verify}(\sigma, x, \pi)$**:** *The algorithm is deterministic and takes as input a CRS* σ*, a statement* x *and a proof* π*, the output is a bit* $b \in \{0,1\}$ *representing whether the proof is accepted or not.*

$\mathsf{Sim}_1(1^\lambda)$**:** *The algorithms takes as input the security parameter* 1^λ *and outputs a CRS* σ *and a trapdoor* ζ*.*

$\mathsf{Sim}_2(\zeta, x)$**:** *The algorithm takes as input a trapdoor* ζ *and a statement* x*, the output is a proof* π*.*

$\mathsf{Ext}(\zeta, x, \pi)$**:** *The algorithm is deterministic and takes as input a trapdoor* ζ*, a statement* x *and a proof* π*. The output is a witness* w *or* $\perp$*.*

Next, we recall their definition of correctness. We require two properties: the first one states that if we generate a proof using a pair $(x, w) \in \mathcal{R}$, then such proof is always accepted. The second property states that if we run the extraction algorithm on a statement x and a proof π, the output is either a witness for w or $\perp$.

Definition 23 (Correctness). *Let* $\mathcal{R}$ *be a relation in* NP*. A simulation-extractable NIZK* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Ext})$ *is correct if:*

– *For every* $\lambda \in \mathbb{N}$ *and* $(x, w) \in \mathcal{R}$*, it holds that*

$$\Pr\left[\begin{matrix}\sigma \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda) \\ \pi \leftarrow\!\!\$\ \mathsf{Prove}(\sigma, x, w)\end{matrix} : \mathsf{Verify}(\sigma, x, \pi) = 1\right] = 1.$$

– *For every* $\lambda \in \mathbb{N}$*, statement* x *and proof* π*, it holds that*

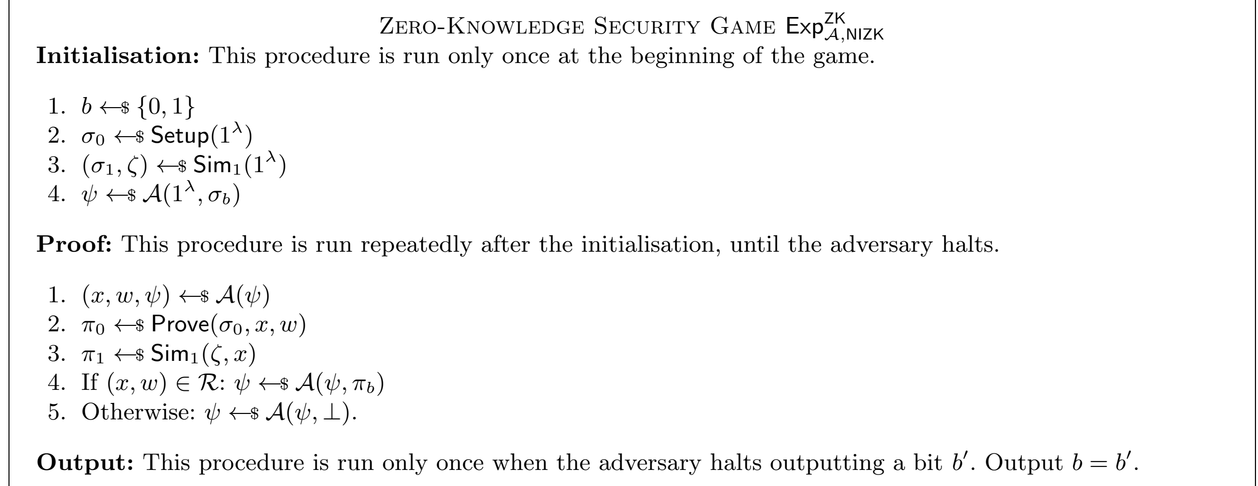
$$\Pr\left[\begin{matrix}(\sigma, \zeta) \leftarrow\!\!\$\ \mathsf{Sim}_1(1^\lambda) \\ w \leftarrow \mathsf{Ext}(\zeta, x, \pi)\end{matrix} : \begin{matrix} w \neq \bot \\ (x, w) \notin \mathcal{R}\end{matrix}\right] = 0.$$

Next, we recall the definition of zero-knowledge: the proofs reveal no information that cannot be already recovered from the statements. This is formalised by means of a indistinguishability-based game definition: in one world, the adversary is provided with a honest CRS and with unbounded oracle access to a proving oracle (the adversary gets to choose statements and witness), in the other world, the CRS given to the adversary hides a trapdoor. The latter is used to generate fake proofs without needing any information about the relative witnesses.

Definition 24 (Zero-knowledge). *Let* $\mathcal{R}$ *be a relation in* NP*. A simulation-extractable NIZK* $\mathsf{NIZK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Ext})$ *is zero-knowledge if, for every PPT adversary* $\mathcal{A}$[6]*, it holds that*

$$\mathsf{Adv}^{\mathsf{ZK}}_{\mathsf{NIZK}}(\mathcal{A}) := \left|\Pr[\mathsf{Exp}^{\mathsf{ZK}}_{\mathcal{A},\mathsf{NIZK}}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda),$$

where experiment $\mathsf{Exp}^{\mathsf{ZK}}_{\mathcal{A},\mathsf{NIZK}}$ *is specified in Figure 3.*



Zero-Knowledge Security Game $\mathsf{Exp}^{\mathsf{ZK}}_{\mathcal{A},\mathsf{NIZK}}$

Initialisation: This procedure is run only once at the beginning of the game.

1. $b \leftarrow\!\!\$\ \{0, 1\}$
2. $\sigma_0 \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)$
3. $(\sigma_1, \zeta) \leftarrow\!\!\$\ \mathsf{Sim}_1(1^\lambda)$
4. $\psi \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, \sigma_b)$

Proof: This procedure is run repeatedly after the initialisation, until the adversary halts.

1. $(x, w, \psi) \leftarrow\!\!\$\ \mathcal{A}(\psi)$
2. $\pi_0 \leftarrow\!\!\$\ \mathsf{Prove}(\sigma_0, x, w)$
3. $\pi_1 \leftarrow\!\!\$\ \mathsf{Sim}_1(\zeta, x)$
4. If $(x, w) \in \mathcal{R}$: $\psi \leftarrow\!\!\$\ \mathcal{A}(\psi, \pi_b)$
5. Otherwise: $\psi \leftarrow\!\!\$\ \mathcal{A}(\psi, \bot)$.

Output: This procedure is run only once when the adversary halts outputting a bit b'. Output $b = b'$.

Fig. 3. Zero-knowledge security game ZK.

Finally, we recall the definition of simulation-extractability: even if we give unbounded oracle access to simulated proofs, no adversary is able to generate any fresh valid proof for which the extraction of the witness fails.

[6] We require that $\mathcal{A}$ halts after a polynomial number of executions of the procedure Proof.

Definition 25 (Simulation-extractability). *Let $\mathcal{R}$ be a relation in* NP. *A NIZK* $\mathsf{NIZK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Ext})$ *is simulation-extractable if, for every PPT adversary $\mathcal{A}$, it holds that*

$$\Pr\left[\begin{matrix}(\sigma,\zeta) \leftarrow\!\!\$\ \mathsf{Sim}_1(1^\lambda) \\ (\pi, x) \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Sim}_2(\zeta,\cdot)}(1^\lambda,\sigma)\end{matrix} : \begin{matrix}\mathsf{Verify}(\sigma,x,\pi)=1 \\ \mathsf{Ext}(\zeta,x,\pi)=\bot \\ x \notin Q\end{matrix}\right] \leq \mathsf{negl}(\lambda),$$

where $\mathcal{A}^{\mathsf{Sim}_2(\zeta,\cdot)}$ indicates that the adversary has unbounded oracle access to $\mathsf{Sim}_2(\zeta,\cdot)$ and Q denotes the set of all queries issued by the adversary to such oracle.

3.6 Garbled Circuits

A garbling scheme specifies how to encrypt a circuit while, at the same time, allowing us to evaluate it on encrypted inputs. Furthermore, the scheme guarantees privacy: no information is revealed beyond the output. Yao informally showed how to construct the primitive at FOCS'86 during the presentation of [Yao86]: garbling just requires the existence of one-way functions. In order to garble a circuit C, it is sufficient to associate each wire of C with a pair of keys (often called labels), one for the value 0 and one for the value 1. Then, we encrypt each gate using the keys associated with its input wires: if the gate computes the function $f : \{0,1\} \times \{0,1\} \to \{0,1\}$, the keys for the input wires are (k_a^0, k_a^1) and (k_b^0, k_b^1), and the keys for the output wire are (k_c^0, k_c^1), we encrypt $k_c^{f(x,y)}$ under the concatenation of k_a^x and k_b^y for every $(x,y) \in \{0,1\} \times \{0,1\}$. The four ciphertexts are then permuted. Now, if we want to evaluate such encrypted circuit on an input $x \in \{0,1\}^n$, it is sufficient to just reveal one key for each input wire of the circuit: we reveal the key associate with 0 if the corresponding bit in x is 0, we reveal the other key otherwise. Given these keys, we can start a cascade of decryptions, retrieving one key for every internal wire of the circuit (in particular, the key associated with the value of the wire in $C(x)$). In this way, we obtain the key associated with the outputs of the circuit, which can be easily decoded, assuming that the keys of the output wires are given as part of the garbling.

All known garbling schemes follow this blueprint. What often changes is the way in which to point to the ciphertexts that we need to decrypt during the evaluation: each encrypted gate consists of four ciphertexts and we are able to decrypt only one of them! Which one is the right one? In the original construction by Yao, the encrypted keys were padded with a sufficiently large number of zeros. The right ciphertext was the one that decrypted into a string ending with a lot of zeros. A more efficient solution is known as the *point-and-permute* technique, introduced by Rogaway [Rog91], which allows for decrypting only the correct ciphertext directly.

4 Delay Functions

In this section, we discuss delay functions: we provide security definitions and we show how to construct them from minimal assumptions.

Defining Delay Functions. A delay function is a function whose evaluation on random inputs is slow. This primitive was introduced for the first time in 1998 by Goldschlag and Stubblebine [GS98]. Their work only provides an informal study of the primitive and lacks precise definitions. In this section, we present a more rigorous description of this cryptographic object and its security properties. We start by presenting the syntax and the efficiency of delay functions.

Definition 26 (Delay function). *A delay function with domain* $\mathcal{X} := (\mathcal{X}_\lambda)_{\lambda\in\mathbb{N}}$ *and range* $\mathcal{Y} := (\mathcal{Y}_\lambda)_{\lambda\in\mathbb{N}}$ *consists of a pair of algorithms* $(\mathsf{Setup}, \mathsf{Eval})$ *with the following syntax:*

$\mathsf{Setup}(\lambda)$**:** *This algorithm takes as input the security parameter* 1^λ *and output public parameters* pp*.*
$\mathsf{Eval}(\mathsf{pp}, x)$**:** *This algorithm is deterministic and takes as input public parameters* pp *and an input* $x \in \mathcal{X}_\lambda$*. The output is a value* $y \in \mathcal{Y}_\lambda$*.*

Definition 27 (Efficiency of delay functions). *Let* $\mathcal{C}$ *be a circuit class. A delay function* $(\mathsf{Setup}, \mathsf{Eval})$ *is* $\mathcal{C}$*-efficient if* $\mathsf{Eval} \in \mathcal{C}$ *and* $\mathsf{Setup} \in \mathsf{poly}(\lambda)$*.*

Next, we present the definition of sequentiality. A low-depth adversary is unable to predict the evaluation of the function on a random input, even if it is allowed to perform polynomial-time preprocessing on the public parameters.

Definition 28 (Sequentiality of delay functions). *Let* $\mathcal{C}$ *be a circuit class. We say a delay function* $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Eval})$ *for domain* $\mathcal{X} := (\mathcal{X}_\lambda)_{\lambda\in\mathbb{N}}$ *and range* $\mathcal{Y} := (\mathcal{Y}_\lambda)_{\lambda\in\mathbb{N}}$ *is* $\mathcal{C}$*-sequential if, for all adversaries* $\mathcal{A} := (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathcal{C}$ *it holds that*

$$\mathsf{Adv}^{\mathsf{SEQ}}_{\mathsf{DF}}(\mathcal{A}) := \Pr[\mathsf{Exp}^{\mathsf{SEQ}}_{\mathcal{A},\mathsf{DF}}(1^\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where experiment $\mathsf{Exp}^{\mathsf{SEQ}}_{\mathcal{A},\mathsf{DF}}$ *was specified in Figure 4 (notice that, compared to VDFs, the syntax of* Eval *is slightly different in delay functions, as no proof of evaluation is produced).*

$\mathsf{Exp}^{\mathsf{SEQ}}_{\mathcal{A},\mathsf{DF}}(1^\lambda)$

1 : $\mathsf{pp} \leftarrow\$ \mathsf{Setup}(1^\lambda)$
2 : $\mathsf{aux} \leftarrow\$ \mathcal{A}_0(\mathsf{pp})$
3 : $x \leftarrow\$ \mathcal{X}_\lambda$
4 : $(y, \pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x)$
5 : $y' \leftarrow\$ \mathcal{A}_1(\mathsf{aux}, x)$
6 : **return** $y = y'$

Fig. 4. Sequentiality security game SEQ.

Finally, we introduce a new security notion, called programmability, for delay functions that will be satisfied by the construction described later in this section. Programmability states that it is possible to generate the public parameters of the delay function so that there exists a random looking input that evaluates to a programmed value. More specifically, given a value $\widehat{y}$, we are able to generate public parameters pp and an input $\widehat{x}$ such that: (1) pp is indistinguishable from honestly generated delay function parameters; (2) the evaluation of the function on $\widehat{x}$ gives $\widehat{y}$; (3) the input $\widehat{x}$ looks random to every low-depth adversary (even if the latter knows pp and $\widehat{y}$). Notice that this last property can be ensured only if the delay function evaluation is sufficiently deep to prevent the adversary from computing it.

Definition 29 (Programmability). *A delay function* $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Eval})$ *with domain* $\mathcal{X} := (\mathcal{X}_\lambda)_{\lambda\in\mathbb{N}}$ *and range* $\mathcal{Y} := (\mathcal{Y}_\lambda)_{\lambda\in\mathbb{N}}$ *is* $\mathcal{C}$*-programmable if there exists a PPT algorithm* $\mathsf{Program}$ *satisfying the following properties:*

(Correctness). *For every sequence* $(\widehat{y}_\lambda)_{\lambda\in\mathbb{N}}$ *such that* $y_\lambda \in \mathcal{Y}_\lambda$ *for every* $\lambda \in \mathbb{N}$,

$$\Pr\left[(\mathsf{pp}, x) \leftarrow\!\!\$\ \mathsf{Program}(1^\lambda, \widehat{y}_\lambda) : \mathsf{Eval}(\mathsf{pp}, x) \neq \widehat{y}_\lambda\right] \leq \mathsf{negl}(\lambda).$$

($\mathcal{C}$-Indistinguishability). *For all adversaries* $\mathcal{A} := (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2) \in \mathsf{poly}(\lambda) \times \mathsf{poly}(\lambda) \times \mathcal{C}$ *it holds that*

$$\mathsf{Adv}^{\mathsf{PROG}}_{\mathsf{DF}}(\mathcal{A}) := \left|\Pr[\mathsf{Exp}^{\mathsf{PROG}}_{\mathcal{A},\mathsf{DF}}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda),$$

where experiment $\mathsf{Exp}^{\mathsf{PROG}}_{\mathcal{A},\mathsf{DF}}$ *is specified in Figure 5.*

$\mathsf{Exp}^{\mathsf{PROG}}_{\mathcal{A},\mathsf{DF}}(1^\lambda)$

1 : $b \leftarrow\!\!\$\ \{0,1\}$
2 : $(\widehat{y}, \mathtt{aux}_0) \leftarrow\!\!\$\ \mathcal{A}_0(1^\lambda)$
3 : $\mathsf{pp}_0 \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)$
4 : $(\mathsf{pp}_1, x_1) \leftarrow\!\!\$\ \mathsf{Program}(1^\lambda, \widehat{y})$
5 : $x_0 \leftarrow\!\!\$\ \mathcal{X}_\lambda$
6 : $\mathtt{aux}_1 \leftarrow\!\!\$\ \mathcal{A}_1(\mathsf{pp}_b, \mathtt{aux}_0)$
7 : $b' \leftarrow\!\!\$\ \mathcal{A}_2(\mathtt{aux}_1, x_b)$
8 : **return** $b = b'$

Fig. 5. Programmability indistinguishability game PROG.

Although it seems counterintuitive, programmability does not seem to imply sequentiality. We can for instance imagine a delay function that consisting of an obfuscated program that on input any $x \in \{0,1\}^\lambda$, produces an pseudorandom time-lock puzzle hiding 0. The message hidden in the puzzle is the output corresponding to the input x. It is not hard to see that the construction does not satisfy sequentiality: the output are highly predictable independently of the adversarial class. However, thanks to indistinguishability obfuscation [BGI+01, JLS21] and the trick by Boyle, Chung and Pass [BCP14], it is easy to modify the program so that, on input a value $\widehat{x}$ chosen at random, it outputs a puzzle hiding the programmed value $\widehat{y}$.

4.1 Building Delay Functions

We are now ready to show how to build programmable delay functions. We construct them based on one-way functions and minimal delay assumptions. Our idea is to rely on point-and-permute garbled circuits [Rog91]. The public parameters of the delay function will consist of a random string that we regard as the garbling of a circuit of a particular topology. The delay function's random inputs are regarded as input labels for the garbled circuit. The evaluation of the function consist of the output labels obtained by evaluating the "garbled circuit" (encoded in the public parameters) on the "input labels".

Now suppose that there exist two circuit classes $\mathcal{C}' \subseteq \mathcal{C}$ and a language L with a decider in $\mathcal{C}$ but no decider in $\mathcal{C}'$. We choose the length of the random string consisting in the public parameters to match the length of the garbling of the universal circuit U for the evaluation of circuits in $\mathcal{C}$.

In this way, it is possible for us to replace the public parameters with a garbling of U without the adversary noticing it. Furthermore, we can substitute the random input string with the labels corresponding to the input (D, x), where $D \in \mathcal{C}$ is the decider for the language L and x is a candidate element in the language. Notice that in this way, the evaluation of the delay function returns the label corresponding with the output 1 or 0 depending on whether $x \in L$ or not. Since we rely on point-and-permute, the adversary cannot tell whether the input corresponds to real input labels or purely random strings.

At this point, it is easy to imagine that if an adversary $\mathcal{A} \in \mathcal{C}'$ manages to predict the delay function output with non-negligible probability, we can leverage it to construct a decider for L in the circuit class $\mathcal{C}'$. We do this by running multiple concurrent tests with $\mathcal{A}$ and using an averaging argument (we need to assume that $\mathcal{C}'$ is closed under polynomially-many parallel compositions). Having reached a contradiction, we conclude that the delay function is $\mathcal{C}'$-sequential. As for programmability, we notice that many garbled circuit schemes (including point-and-permute) allow us to program the output labels. Furthermore, the knowledge of these labels gives no advantage in speeding up the evaluation of the garbled circuit. This is sufficient to ensure the programmability of the delay function.

Evaluation-Oblivious and Evaluation-Programmable Garbled Circuits. Before formalising our intuition, we need to define garbled circuit schemes and all the properties we require from them. As already mentioned in the above paragraph, the classical point-and-permute construction from one-way functions [Rog91] satisfies all the necessary conditions to build programmable and sequential delay functions.

We start by recapping the syntax, correctness and authenticity of garbled circuits following the blueprint of Bellare, Hoang and Rogaway [BHR12]. We recall that a garbling scheme consists of a primitive describing how to encrypt a circuit so that we can still evaluate it on encrypted inputs while preserving their privacy. Correctness states that if we evaluate the garbling of a circuit C on the encoding of an input x, we obtain an encoding of $y := C(x)$. Authenticity states instead that, if we provide the garbling of a circuit C and the encoding of an input x, the only output encodings the adversary will be able to derive are relative to $y := C(x)$.

Definition 30 (Garbled Circuits [BHR12]). *A garble circuit scheme for the circuit class $(\mathcal{C}_\lambda)_{\lambda \in \mathbb{N}}$ consists of a tuple of PPT algorithms $(\mathsf{Garble}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ with the following syntax:*

$(G, e, d) \leftarrow\!\$\ \mathsf{Garble}(1^\lambda, C)$**:** *The garbling algorithm takes as input the security parameter 1^λ and a circuit $C \in \mathcal{C}_\lambda$. It returns a garbling G and encoding and decoding information e and d.*

$X \leftarrow \mathsf{Enc}(e, x)$**:** *The encoding algorithm takes as input encoding information e and a value x. The output is an encoding of the input X.*

$Y \leftarrow \mathsf{Eval}(G, X)$**:** *The evaluation algorithm takes as input a garbling G and an encoding of the input X. The output is an encoding of the output Y.*

$y \leftarrow \mathsf{Dec}(d, Y)$**:** *The verification algorithm takes as input decoding information d and an encoding of the output Y. The output is a value y or $\bot$.*

Definition 31 (Correctness of Garbled Circuits [BHR12]). *We say that a garbled circuit scheme for the circuit class $(\mathcal{C}_\lambda)_{\lambda \in \mathbb{N}}$ if there exists a negligible function $\mathsf{negl}(\lambda)$ such that, for every*

$\lambda \in \mathbb{N}$, $C \in \mathcal{C}_\lambda$ *and* x *in the input space of* C,

$$\Pr\begin{bmatrix}(G,e,d) \leftarrow\!\!\$\ \mathsf{Garble}(1^\lambda, C) & \\ X \leftarrow \mathsf{Enc}(e,x) & \\ Y \leftarrow \mathsf{Eval}(G,X) & : y \neq C(x) \\ y \leftarrow \mathsf{Dec}(d,T) & \end{bmatrix} \leq \mathsf{negl}(\lambda).$$

Definition 32 (Authenticity [BHR12]). *A garbling scheme for the circuit class* $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$ *satisfies authenticity if there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that, for every PPT adversary* $\mathcal{A}$, $\lambda \in \mathbb{N}$, $C \in \mathcal{C}_\lambda$ *and value* x *in the input space of* C,

$$\Pr\begin{bmatrix}(G,e,d) \leftarrow\!\!\$\ \mathsf{Garble}(1^\lambda, C) & \mathsf{Dec}(d,Y) \neq \bot \\ X \leftarrow \mathsf{Enc}(e,x) & : \\ Y \leftarrow \mathcal{A}(1^\lambda, G, X) & Y \neq \mathsf{Eval}(G,X)\end{bmatrix} \leq \mathsf{negl}(\lambda).$$

Next, we introduce the first property required to build sequential delay functions: *evaluation-obliviousness*. The latter states that the garbled circuit G can be simulated without any knowledge of the underlying circuit. Furthermore, the input labels can be simulated given just G, no other information is required. If the simulators both outputs random strings, we say that the garbling scheme satisfies *strong evaluation obliviousness*. Notice that this property implies that an adversary cannot tell whether it is evaluating a garbled circuit on the actual input labels or on random strings. We also highlight that the original garbling scheme of Yao [Yao86] does not satisfy evaluation obliviousness due to the padding of zeros added to the encrypted labels (the padding indicates which labels to use for the decryption of the next layer of gates). Point-and-permute garbling [Rog91], on the other hand, is strongly evaluation-oblivious.

Definition 33 (Evaluation-Oblivious Garbled Circuits). *We say that a garbled circuit scheme for the circuit class* $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$ *satisfies evaluation obliviousness if there exists PPT algorithms* Sim_1, Sim_2 *and a negligible function* $\mathsf{negl}(\lambda)$ *such that, for every PPT adversary* $\mathcal{A}$, $\lambda \in \mathbb{N}$, $C \in \mathcal{C}_\lambda$ *and* x *in the input space of* C,

$$\left|\Pr\begin{bmatrix}(G,e,d) \leftarrow\!\!\$\ \mathsf{Garble}(1^\lambda, C) \\ X \leftarrow \mathsf{Enc}(e,x)\end{bmatrix} : \mathcal{A}(1^\lambda, G, X) = 1\right] - \Pr\begin{bmatrix}G \leftarrow\!\!\$\ \mathsf{Sim}_1(1^\lambda) \\ X \leftarrow\!\!\$\ \mathsf{Sim}_2(G)\end{bmatrix} : \mathcal{A}(1^\lambda, G, X) = 1\right]\right|$$

is upper bounded by $\mathsf{negl}(\lambda)$. *We say that the garbling scheme is strongly evaluation-oblivious if* Sim_1 *and* Sim_2 *output uniformly random strings of bits of fixed polynomial length.*

We highlight that evaluation-obliviousness is stronger than the simulation-based obliviousness definition of [BHR12]. In their definition, the garbling and the input encodings were generated by a single simulator, whereas, in the above definition, we rely on two simulators sharing no trapdoor.

The next garbled circuit property we define is necessary for delay function programmability; we call it *evaluation-programmability*. It states that it is possible to program the encoding of one of the outputs of the garbled circuit, namely given a pair $(y, \widehat{Y})$, we can garble a circuit C so that the evaluation of the encrypted circuit on any input x, such that $C(x) = y$, returns the value $\widehat{Y}$. We also require two additional properties: we ask that the programmed garbled circuit looks like a normal garbling of C, even if we provide the encoding of an input x such that $C(x) \neq y$ (if $C(x) = y$, the

adversary can trivially verify if the encoding of the output y has been programmed to $\widehat{Y}$ by just evaluating the encrypted circuit). Furthermore, we require that even if we program the encoding of the output y, the garbling scheme still guarantees the privacy of the inputs: if $C(x_0) = C(x_1) = y$, the encoding of x_0 and x_1 are still indistinguishable.

Definition 34 (Evaluation-programmability). *Let* $\mathsf{GC} := (\mathsf{Garble}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *be a garbled circuit scheme for the circuit class* $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$. *For every* $\lambda \in \mathbb{N}$ *and* $C \in \mathcal{C}_\lambda$, *we define the set*

$$\mathcal{Y}_\lambda^C := \left\{\mathsf{Eval}(G, X) \middle| (G, e, d) = \mathsf{Garble}(1^\lambda, C; r), r \in \{0,1\}^*, e = \mathsf{Enc}(e, x), x \in \{0,1\}^m\right\}$$

where m *denotes the input size of* C. *We say that the garbled circuit scheme is evaluation-programmable if there exists a PPT algorithm* $\mathsf{ProgramGarble}$ *satisfying the following properties:*

(Programmability). *For every* $\lambda \in \mathbb{N}$, $C \in \mathcal{C}_\lambda$, $\widehat{Y} \in \mathcal{Y}_\lambda^C$, $y \in \{0,1\}^*$ *and* x *such that* $C(x) = y$,

$$\Pr\left[\begin{matrix}(G, e, d) \leftarrow\!\!\$\ \mathsf{ProgramGarble}(1^\lambda, C, y, \widehat{Y}) \\ X \leftarrow \mathsf{Enc}(e, x)\end{matrix} : \mathsf{Eval}(G, X) = \widehat{Y}\right] = 1$$

(Indistinguishability). *We say that an adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ *is valid if, on input* 1^λ, $\mathcal{A}_0$ *outputs an internal state* aux, *a circuit* $C \in \mathcal{C}_\lambda$, *an element* $\widehat{Y} \in \mathcal{Y}_\lambda^C$ *and a pair* (x, y) *such that* x *is in the input space of* C *and* $C(x) \neq y$. *We require that for any valid adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathsf{poly}(\lambda)$,

$$\mathsf{Adv}_{\mathsf{GC}}^{\mathsf{EvPROG}}(\mathcal{A}) := \left|\Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{GC}}^{\mathsf{EvPROG}}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda),$$

where experiment $\mathsf{Exp}_{\mathcal{A},\mathsf{GC}}^{\mathsf{EvPROG}}$ *is specified in Figure 6.*

$\mathsf{Exp}_{\mathcal{A},\mathsf{GC}}^{\mathsf{EvPROG}}(1^\lambda)$

1 : $b \leftarrow\!\!\$\ \{0,1\}$
2 : $(\mathsf{aux}, C, \widehat{Y}, x, y) \leftarrow\!\!\$\ \mathcal{A}_0(1^\lambda)$
3 : $(G_0, e_0, d_0) \leftarrow\!\!\$\ \mathsf{Garble}(1^\lambda, C)$
4 : $(G_1, e_1, d_1) \leftarrow\!\!\$\ \mathsf{ProgramGarble}(1^\lambda, C, y, \widehat{Y})$
5 : $X_0 \leftarrow \mathsf{Enc}(e_0, x)$
6 : $X_1 \leftarrow \mathsf{Enc}(e_1, x)$
7 : $b' \leftarrow\!\!\$\ \mathcal{A}_1(\mathsf{aux}, G_b, X_b)$
8 : **return** $b = b'$

Fig. 6. Evaluation-programmability game EvPROG.

(Input privacy). *We say that an adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ *is valid if, on input* 1^λ, $\mathcal{A}_0$ *outputs an internal state* aux, *a circuit* $C \in \mathcal{C}_\lambda$, *an element* $\widehat{Y} \in \mathcal{Y}_\lambda^C$ *and a triple* (x_0, x_1, y) *such that* $C(x_0) = C(x_1) = y$. *We require that for any valid adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathsf{poly}(\lambda)$,

$$\mathsf{Adv}_{\mathsf{GC}}^{\mathsf{PRIV}}(\mathcal{A}) := \left|\Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{GC}}^{\mathsf{PRIV}}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda),$$

where experiment $\mathsf{Exp}_{\mathcal{A},\mathsf{GC}}^{\mathsf{PRIV}}$ *is specified in Figure 7.*

$\mathsf{Exp}^{\mathsf{PRIV}}_{\mathcal{A},\mathsf{GC}}(1^\lambda)$

1 : $b \leftarrow\!\!\$ \{0,1\}$
2 : $(\mathtt{aux}, C, \widehat{Y}, x_0, x_1, y) \leftarrow\!\!\$ \mathcal{A}_0(1^\lambda)$
3 : $(G, e, d) \leftarrow\!\!\$ \mathsf{ProgramGarble}(1^\lambda, C, y, \widehat{Y})$
4 : $X_0 \leftarrow \mathsf{Enc}(e, x_0)$
5 : $X_1 \leftarrow \mathsf{Enc}(e, x_1)$
6 : $b' \leftarrow\!\!\$ \mathcal{A}_1(\mathtt{aux}, G, X_b)$
7 : **return** $b = b'$

Fig. 7. Input privacy game PRIV.

We finally provide an efficiency definition with respect to a circuit class $\mathcal{C}$. We are particularly interested in the efficiency of the encoding and decoding procedures.

Definition 35 (Efficiency). *Let $\mathcal{C}$ be a circuit class. We say that a garbling scheme is $\mathcal{C}$-efficient if* $\mathsf{Enc}, \mathsf{Dec} \in \mathcal{C}$.

The following theorem summarises the state-of-the-art on evaluation-oblivious, evaluation-programmable garbled circuits. Thanks to the point-and-permute construction of Rogaway [Rog91], we can build them from one-way functions.

Theorem 2 ([Yao86, Rog91]). *Let $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$ be a class of circuits such that for every $\lambda \in \mathbb{N}$ and $C_0, C_1 \in \mathcal{C}_\lambda$, the circuits C_0 and C_1 have the same underlying graph structure.*

Assuming the existence of one-way functions, there exists a strongly evaluation-oblivious, evaluation-programmable garbling scheme for $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$ satisfying both correctness and authenticity. Moreover, the scheme is NC_1-efficient. The size of the garbled circuit is $4\lambda \cdot s$ where s is the size of the garbled circuit.

We observe that, depending on the circuit class $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$, the size of the strongly evaluation-oblivious garbled circuit can be improved to $4\lambda \cdot s'$ where s' denotes the number of non-linear gates in the garbled circuit, thanks to the free-XOR technique of [KS08]. With this improvement, however, the garbling scheme leaks information about the gates of the garbled circuit (in particular, the number and the position of all linear gates).

Our Programmable and Sequential Delay Function. We can finally formalise our $\mathcal{C}'$-programmable, $\mathcal{C}'$-sequential delay function based on garbled circuits and the existence of a circuit class $\mathcal{C} \subseteq \mathsf{poly}(\lambda)$ such that $\mathcal{L}^{\mathcal{C}} \not\subseteq \widetilde{\mathcal{L}}^{\mathcal{C}'}$. The scheme is described in Fig. 8. We state and prove the security properties in the theorem below. Notice that we need to rely on $\mathcal{L}^{\mathcal{C}} \not\subseteq \widetilde{\mathcal{L}}^{\mathcal{C}'}$ instead of the weaker assumption $\mathcal{L}^{\mathcal{C}} \not\subseteq \mathcal{L}^{\mathcal{C}'}$ due to the usual gap between security and infinitely-often security: the assumption $\mathcal{L}^{\mathcal{C}} \not\subseteq \mathcal{L}^{\mathcal{C}'}$ would only allow us to be a delay function that is $\mathcal{C}'$-sequential only for a strictly increasing subsequence of infinitely-many $\lambda \in \mathbb{N}$.

Theorem 3. *Le $\mathcal{C} \subseteq \mathsf{poly}(\lambda)$ be the product circuit class induced by $(\mathcal{C}_\lambda)_{\lambda\in\mathbb{N}}$. Let U_λ be the universal circuit for the circuit class $\mathcal{C}_\lambda$. Let $\mathsf{GC} = (\mathsf{Garble}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec}, \mathsf{Sim}_1, \mathsf{Sim}_2)$ be an evaluation-oblivious garbling scheme for a circuit class containing U_λ for every $\lambda \in \mathbb{N}$. Suppose that the scheme satisfies correctness, authenticity and $\mathcal{C}'$-efficiency where $\mathcal{C}'$ is nicely closed and contains NC_1.*

A DELAY FUNCTION BASED ON EVALUATION-OBLIVIOUS GARBLED CIRCUITS

Let U_λ be the universal circuit for the circuit class $\mathcal{C}_\lambda$. Let $\mathsf{GC} = (\mathsf{Garble}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec}, \mathsf{Sim}_1, \mathsf{Sim}_2)$ be an evaluation-oblivious garbling scheme for a circuit class containing U_λ for every $\lambda \in \mathbb{N}$. Suppose also that Sim_2 outputs a uniformly random value in the domain $\mathcal{X}$

$\mathsf{Setup}(1^\lambda)$**:** Output $\mathsf{pp} := G \leftarrow\!\!\$\ \mathsf{GC.Sim}_1(1^\lambda)$
$\mathsf{Eval}(\mathsf{pp} = G, X)$**:** Output $Y \leftarrow \mathsf{GC.Eval}(G, X)$

Fig. 8. A delay function based on evaluation-oblivious garbled circuits

Then, if $\mathcal{L}^{\mathcal{C}} \not\subseteq \widetilde{\mathcal{L}}^{\mathcal{C}'}$, the construction in Fig.8 is a $\mathcal{C}'$-sequential delay function. Furthermore, if GC *is evaluation-programmable, the delay function is $\mathcal{C}'$-programmable.*

Proof. Suppose that our claim is wrong: there exists an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathcal{C}'$ such that the advantage $\mathsf{Adv}^{\mathsf{SEQ}}_{\mathsf{DF}}(\mathcal{A})$ is a non-negligible function. Let $\epsilon(\lambda)$ be an inverse polynomial function lower-bounding this advantage for infinitely many $\lambda \in \mathbb{N}$. Let S be the subset of all such values of λ.

We consider a language $L = \bigcup_{\lambda \in \mathbb{N}} L_\lambda \in \mathcal{L}^{\mathcal{C}} \setminus \widetilde{\mathcal{L}}^{\mathcal{C}'}$. Let $(D_\lambda)_{\lambda \in \mathbb{N}} \in \mathcal{C}$ be a decider for such language. We construct a sequence of circuits $(R_\lambda)_{\lambda \in \mathbb{N}}$ as follows:

1. R_λ will receive a challenge x as input along with advice (e, d, aux).
2. R_λ will compute $X \leftarrow \mathsf{GC.Enc}(e, (D_\lambda, x))$ and run $Y \leftarrow\!\!\$\ \mathcal{A}_1(\mathsf{aux}, X)$.
3. R_λ will output $\mathsf{GC.Dec}(d, Y)$ if the latter is different from $\perp$, otherwise, it output a random bit.

We observe that $(R_\lambda)_{\lambda \in \mathbb{N}} \in \mathcal{C}'$, given that such class is nicely closed.

Claim 4. *There exists a negligible function* $\mathsf{negl}(\lambda)$ *such that, for every $\lambda \in S$,*

$$\Pr\left[\begin{array}{l}(G, e, d) \leftarrow\!\!\$\ \mathsf{GC.Garble}(1^\lambda, U_\lambda) \\ \mathsf{aux} \leftarrow\!\!\$\ \mathcal{A}_0(1^\lambda, G) \\ y \leftarrow\!\!\$\ R_\lambda(x, e, d, \mathsf{aux})\end{array} : y = D_\lambda(x)\right] \geq \frac{1}{2} + \frac{\epsilon(\lambda)}{2} + \mathsf{negl}(\lambda).$$

Proof. We prove such claim by observing that, by the authenticity of the garbling scheme, there exists another negligible function $\eta_1(\lambda)$ such that

$$\Pr\left[\begin{array}{l}(G, e, d) \leftarrow\!\!\$\ \mathsf{GC.Garble}(1^\lambda, U_\lambda) \\ \mathsf{aux} \leftarrow\!\!\$\ \mathcal{A}_0(1^\lambda, G) \\ X \leftarrow \mathsf{GC.Enc}(e, (D_\lambda, x)) \\ Y \leftarrow\!\!\$\ \mathcal{A}_1(\mathsf{aux}, X)\end{array} : \begin{array}{l}\mathsf{GC.Dec}(d, Y) \neq \perp \\ Y \neq \mathsf{GC.Eval}(G, X)\end{array}\right] \leq \eta_1(\lambda).$$

Furthermore, due to the correctness of the garbling scheme, there exists a negligible function $\eta_2(\lambda)$ such that

$$\Pr\left[\begin{array}{l}(G, e, d) \leftarrow\!\!\$\ \mathsf{GC.Garble}(1^\lambda, U_\lambda) \\ \mathsf{aux} \leftarrow\!\!\$\ \mathcal{A}_0(1^\lambda, G) \\ y \leftarrow\!\!\$\ R_\lambda(x, e, d, \mathsf{aux})\end{array} : y = D_\lambda(x)\right]$$

$$= \Pr\left[\begin{array}{l}(G,e,d) \leftarrow\$ \ \mathsf{GC.Garble}(1^\lambda, U_\lambda)\\ \mathsf{aux} \leftarrow\$ \ \mathcal{A}_0(1^\lambda, G)\\ X \leftarrow \mathsf{GC.Enc}(e,(D_\lambda,x))\\ Y \leftarrow\$ \ \mathcal{A}_1(\mathsf{aux},X)\end{array} : Y = \mathsf{GC.Eval}(G,X)\right]$$

$$+ \frac{1}{2}\cdot\Pr\left[\begin{array}{l}(G,e,d) \leftarrow\$ \ \mathsf{GC.Garble}(1^\lambda, U_\lambda)\\ \mathsf{aux} \leftarrow\$ \ \mathcal{A}_0(1^\lambda, G)\\ X \leftarrow \mathsf{GC.Enc}(e,(D_\lambda,x))\\ Y \leftarrow\$ \ \mathcal{A}_1(\mathsf{aux},X)\end{array} : \mathsf{GC.Eval}(G,X) = \bot\right] + \eta_1(\lambda) + \eta_2(\lambda) =$$

$$= \frac{1}{2} + \frac{1}{2}\cdot\Pr\left[\begin{array}{l}(G,e,d) \leftarrow\$ \ \mathsf{GC.Garble}(1^\lambda, U_\lambda)\\ \mathsf{aux} \leftarrow\$ \ \mathcal{A}_0(1^\lambda, G)\\ X \leftarrow \mathsf{GC.Enc}(e,(D_\lambda,x))\\ Y \leftarrow\$ \ \mathcal{A}_1(\mathsf{aux},X)\end{array} : Y = \mathsf{GC.Eval}(G,X)\right] + \frac{\eta_1(\lambda)}{2} + \eta_2(\lambda).$$

Finally, by the evaluation-obliviousness of the garbling scheme, there exists a negligible function $\eta_3(\lambda)$ such that

$$\Pr\left[\begin{array}{l}(G,e,d) \leftarrow\$ \ \mathsf{GC.Garble}(1^\lambda, U_\lambda)\\ \mathsf{aux} \leftarrow\$ \ \mathcal{A}_0(1^\lambda, G)\\ X \leftarrow \mathsf{GC.Enc}(e,(D_\lambda,x))\\ Y \leftarrow\$ \ \mathcal{A}_1(\mathsf{aux},X)\end{array} : Y = \mathsf{GC.Eval}(G,X)\right]$$

$$= \Pr\left[\begin{array}{l}G \leftarrow\$ \ \mathsf{GC.Sim}_1(1^\lambda)\\ \mathsf{aux} \leftarrow\$ \ \mathcal{A}_0(1^\lambda, G)\\ X \leftarrow\$ \ \mathsf{GC.Sim}_2(G)\\ Y \leftarrow\$ \ \mathcal{A}_1(\mathsf{aux},X)\end{array} : Y = \mathsf{GC.Eval}(G,X)\right] + \eta_3(\lambda).$$

Since $\mathsf{GC.Sim}_2$ samples X uniformly in $\mathcal{X}$, we conclude that, for every $\lambda \in S$,

$$\Pr\left[\begin{array}{l}(G,e,d) \leftarrow\$ \ \mathsf{GC.Garble}(1^\lambda, U_\lambda)\\ \mathsf{aux} \leftarrow\$ \ \mathcal{A}_0(1^\lambda, G)\\ y \leftarrow\$ \ R_\lambda(x,e,d,\mathsf{aux})\end{array} : y = D_\lambda(x)\right] = \frac{1}{2} + \frac{\epsilon(\lambda)}{2} + \frac{\eta_1(\lambda)}{2} + \eta_2(\lambda) + \frac{\eta_3(\lambda)}{2}$$

This ends the proof of the claim. ■

Now, consider the algorithm that, on input x, runs $R_\lambda(x,e,d,\mathsf{aux})$ T times generating

$$(G,e,d) \leftarrow\$ \ \mathsf{GC.Garble}(1^\lambda, U_\lambda) \quad \text{and} \quad \mathsf{aux} \leftarrow\$ \ \mathcal{A}_0(1^\lambda, G)$$

freshly each time. Then, it outputs the most frequent result among the T executions. We observe that, for every $\lambda \in S$, the number of correct outputs among the T executions is distributed according to a binomial distribution with parameter $p(\lambda) \geq \frac{1}{2} + \frac{\epsilon(\lambda)}{4}$. Using the Chernoff bound, we obtain that the majority of outputs is correct with probability at least $1 - e^{-\Omega(\epsilon^2)\cdot T}$ for every $\lambda \in S$.

We conclude by an averaging argument: for every $\lambda \in S$ and x, our algorithm correctly classifies x with respect to L_λ with probability at least $1 - e^{-\Omega(\epsilon^2)\cdot T}$ over its randomness and the randomness

of e, d, aux. Therefore, restricted to $\lambda \in S$, there exists a random tape for which our algorithm correctly classifies at least a $1 - e^{-\Omega(\epsilon^2) \cdot T}$ fraction of all values x. We say an x is bad if it is not classified correctly by this random tape.

Let $\ell(\lambda)$ be the length of all elements in L_λ. We chose T such that $T = \omega(\ell(\lambda) \cdot \epsilon^{-2})$ and we observe that the number of bad x is less than $2^{\ell(\lambda)} \cdot e^{-\omega(\ell(\lambda))}$ such quantity is asymptotically strictly smaller than 1. In other words, there exists a finite set $E \subseteq \mathbb{N}$ such that, for every $\lambda \in S \setminus E$, there is a random tape r_λ for our algorithm that correctly classifies all instances $x \in \{0,1\}^{\ell(\lambda)}$. For every $\lambda \in \mathbb{N} \setminus S$, we set r_λ to the all zero string.

We finally reach a contradiction: for every $\lambda \in \mathbb{N}$, we consider the circuit obtained by fixing the randomness of T parallel copies of R_λ and their advice (e, d, aux) according to r_λ, or, when $\lambda \in E$, the circuit D_λ. Such sequence of circuits constitutes an infinitely often decider for L belonging to $\mathcal{C}'$. So, $L \in \widetilde{\mathcal{L}}^{\mathcal{C}'}$.

Programmability. We not show that the delay function is $\mathcal{C}'$-programmable. Let $\widehat{x}_\lambda$ be an element of L_λ. We consider the algorithm $\mathsf{GC.Program}$ that, on input $1\!\!1^\lambda$ and $\widehat{Y} \in \mathcal{Y}_\lambda$, computes the following operations:

1. $(G, e, d) \leftarrow\$ \mathsf{GC.ProgramGarble}(1^\lambda, U_\lambda, 1, \widehat{Y})$
2. $X \leftarrow \mathsf{GC.Enc}(e, (D_\lambda, \widehat{x}_\lambda))$
3. Output $(\mathsf{pp} := G, X)$

We observe that such algorithm satisfies the correctness of the delay function programmability (see Def.29). Indeed, we have that $U_\lambda(D_\lambda, \widehat{x}_\lambda) = 1$, therefore, by the correctness of evaluation-programmability, we obtain that $\mathsf{Eval}(\mathsf{pp} := G, X) = \mathsf{GC.Eval}(G, X) = \widehat{Y}$.

Next, we focus on $\mathcal{C}'$-indistinguishability. Suppose that there exists an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2) \in \mathsf{poly}(\lambda) \times \mathsf{poly}(\lambda) \times \mathcal{C}'$ that wins the programmability game PROG (see Fig.5) with non negligible advantage. Let $\epsilon(\lambda)$ be an inverse polynomial function lower-bounding this advantage for infinitely many $\lambda \in \mathbb{N}$. Let S be the subset of all such values of λ. We show how to build a decider for the language $L \in \mathcal{L}^{\mathcal{C}} \setminus \widetilde{\mathcal{L}}^{\mathcal{C}'}$.

Consider any value x in the input space of D_λ. We consider the sequence of circuits $(R_\lambda)_{\lambda \in \mathbb{N}}$ performing the following operations:

1. R_λ will receive as input a challenge x and auxiliary advice (e, d, aux)
2. R_λ will compute $X \leftarrow \mathsf{GC.Enc}(e, (D_\lambda, x))$
3. R_λ will output $\mathcal{A}_2(\mathsf{aux}, X)$.

Now, suppose that $x \notin L$. We observe that

$$\left| \Pr\left[\begin{matrix} (\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\ (G, e, d) \leftarrow\$ \mathsf{GC.ProgramGarble}(1^\lambda, U_\lambda, 1, \widehat{y}) : R_\lambda(x, e, d, \mathsf{aux}) = 1 \\ \mathsf{aux} \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0) \end{matrix}\right] - \Pr\left[\begin{matrix} (\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\ (G, e, d) \leftarrow\$ \mathsf{GC.Garble}(1^\lambda, U_\lambda) : R_\lambda(x, e, d, \mathsf{aux}) = 1 \\ \mathsf{aux} \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0) \end{matrix}\right] \right| \leq \mathsf{negl}(\lambda).$$

Indeed, if that was not the case, we would contradict the second property of the evaluation programmability of GC. Specifically, in the reduction, we would consider the adversary $\mathcal{A}' =$

$(\mathcal{A}'_0, \mathcal{A}'_1)$ against EvPROG (see Fig.6) where $\mathcal{A}'_0$ runs $(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0$ and outputs the tuple $(\mathsf{aux}_0, U_\lambda, \widehat{y}, (D_\lambda, x), 1)$. The adversary $\mathcal{A}'_1$, after receiving aux_0, G and X, would instead compute $\mathsf{aux}_1 \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0)$ and output $b' \leftarrow\$ \mathcal{A}_2(\mathsf{aux}_1, X)$.

We observe that

$$\left| \Pr\left[\begin{array}{l}(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\ G \leftarrow\$ \mathsf{GC.Sim}_1(1^\lambda) \\ \mathsf{aux}_1 \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0) \\ X \leftarrow\$ \mathsf{GC.Sim}_2(1^\lambda)\end{array} : \mathcal{A}_2(\mathsf{aux}_1, X) = 1\right] - \Pr\left[\begin{array}{l}(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\ (G, e, d) \leftarrow\$ \mathsf{GC.Garble}(1^\lambda, U_\lambda) \\ \mathsf{aux} \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0)\end{array} : R_\lambda(x, e, d, \mathsf{aux}) = 1\right] \right| \leq \mathsf{negl}(\lambda).$$

Indeed, if that was not the case, we would contradict the evaluation obliviousness of GC. Specifically, in the reduction, we would consider the adversary $\mathcal{A}'$ that runs $(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0$ and computes $\mathsf{aux}_1 \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0)$ and output $b' \leftarrow\$ \mathcal{A}_2(\mathsf{aux}_1, X)$.

Similarly to before, we now consider the case in which $x \in L$. We observe that

$$\left| \Pr\left[\begin{array}{l}(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\ (G, e, d) \leftarrow\$ \mathsf{GC.ProgramGarble}(1^\lambda, U_\lambda, 1, \widehat{y}) \\ \mathsf{aux} \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0)\end{array} : R_\lambda(x, e, d, \mathsf{aux}) = 1\right] - \Pr\left[\begin{array}{l}(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\ (G, e, d) \leftarrow\$ \mathsf{GC.ProgramGarble}(1^\lambda, U_\lambda, 1, \widehat{y}) \\ \mathsf{aux} \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0) \\ X \leftarrow \mathsf{GC.Enc}(e, (D_\lambda, \widehat{x}_\lambda))\end{array} : \mathcal{A}_2(\mathsf{aux}, X) = 1\right] \right| \leq \mathsf{negl}(\lambda).$$

Indeed, if that was not the case, we would contradict the third property of the evaluation programmability of GC. Specifically, in the reduction, we would consider the adversary $\mathcal{A}' = (\mathcal{A}'_0, \mathcal{A}'_1)$ against PRIV (see Fig.7) where $\mathcal{A}'_0$ runs $(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0$ and outputs the tuple $(\mathsf{aux}_0, U_\lambda, \widehat{y}, (D_\lambda, \widehat{x}_\lambda), (D_\lambda, x), 1)$. The adversary $\mathcal{A}'_1$, after receiving aux_0, G and X, would instead compute $\mathsf{aux}_1 \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0)$ and output $b' \leftarrow\$ \mathcal{A}_2(\mathsf{aux}_1, X)$.

Now, let $\ell(\lambda)$ be the length of all elements in L_λ. Let $T := \lambda \cdot \epsilon(\lambda)^{-2} \cdot \ell(\lambda)$ and define

$$p_0(\lambda) := \Pr\left[\begin{array}{l}(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\ G \leftarrow\$ \mathsf{GC.Sim}_1(1^\lambda) \\ \mathsf{aux}_1 \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0) \\ X \leftarrow\$ \mathsf{GC.Sim}_2(1^\lambda)\end{array} : \mathcal{A}_2(\mathsf{aux}_1, X) = 1\right]$$

$$p_1(\lambda) := \Pr\left[\begin{array}{l}(\widehat{y}, \mathsf{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\ (G, e, d) \leftarrow\$ \mathsf{GC.ProgramGarble}(1^\lambda, U_\lambda, 1, \widehat{y}) \\ \mathsf{aux} \leftarrow\$ \mathcal{A}_1(G, \mathsf{aux}_0) \\ X \leftarrow \mathsf{GC.Enc}(e, (D_\lambda, \widehat{x}_\lambda))\end{array} : \mathcal{A}_2(\mathsf{aux}, X) = 1\right]$$

We know that for every $\lambda \in S$, $|p_0(\lambda) - p_1(\lambda)| \geq \epsilon(\lambda)$.

We consider the algorithm that, on input the candidate value x for the language L, runs $R_\lambda(x, e, d, \mathtt{aux})$ for T times generating

$$(\widehat{y}, \mathtt{aux}_0) \leftarrow\$ \mathcal{A}_0(1^\lambda), \quad (G, e, d) \leftarrow\$ \mathsf{GC.ProgramGarble}(1^\lambda, U_\lambda, 1, \widehat{y}), \quad \mathtt{aux} \leftarrow\$ \mathcal{A}_1(G, \mathtt{aux}_0)$$

freshly each time. The algorithm outputs 0 if the average result is closer to p_0 than to p_1, otherwise, it outputs 1. Notice that by the Chernoff bound, when $x \in L$, the algorithm outputs 1 with overwhelming probability. Otherwise, the algorithm outputs 0 with overwhelming probability.

We conclude again by an averaging argument: for every $\lambda \in S$ and x, our algorithm correctly classifies x with respect to L_λ with probability at least $1 - e^{-\epsilon^2 \cdot T/8}$ over its randomness and the randomness of $e, d, \mathtt{aux}$. Therefore, restricted to $\lambda \in S$, there exists a random tape for which our algorithm correctly classifies at least a $1 - e^{-\Omega(\epsilon^2) \cdot T}$ fraction of all values x. We say an x is bad if it is not classified correctly by this random tape. We observe that the number of bad x is less than $2^{\ell(\lambda)} \cdot e^{-\omega(\ell(\lambda))}$ such quantity is asymptotically strictly smaller than 1. In other words, there exists a finite set $E \subseteq \mathbb{N}$ such that, for every $\lambda \in S \setminus E$, there is a random tape r_λ for our algorithm that correctly classifies all instances $x \in \{0,1\}^{\ell(\lambda)}$. For every $\lambda \in \mathbb{N} \setminus S$, we set r_λ to the all zero string.

We finally reach a contradiction: for every $\lambda \in \mathbb{N}$, we consider the circuit obtained by fixing the randomness of T parallel copies of R_λ and their advice $(e, d, \mathtt{aux})$ according to r_λ, or, when $\lambda \in E$, the circuit D_λ. Such sequence of circuits constitutes an infinitely often decider for L belonging to $\mathcal{C}'$. So, $L \in \widetilde{\mathcal{L}}^{\mathcal{C}'}$. □

5 Delay Encryption

In this section, we study delay encryption outside of obfustopia. Recall that within obfustopia one can trivially achieve delay encryption as outlined in the technical overview. We propose two ways to build delay encryption, both involving the use of programmable delay functions. The first one is based on witness encryption, whereas the second one is based on attribute-based encryption.

5.1 Delay Encryption from Witness Encryption

We start by presenting the construction based on witness encryption. The idea is rather simple. The public parameters of the primitive we want to build include the description of a delay function and an element $\widehat{y}$ in its range. In order to encrypt a message m under a random label $\mathtt{id}$, we just rely on witness encryption. More precisely, the statement under which we generate the ciphertext claims that the evaluation of the delay function on $\mathtt{id}$ does not return $\widehat{y}$.

It is easy to see that the scheme satisfies correctness with overwhelming probability. Since the delay function outputs are highly unpredictable, the evaluation on $\mathtt{id}$ will return $\widehat{y}$ with negligible probability. On the other hand, the construction guarantees security against low-depth attackers. We can consider a hybrid world in which we provide the adversary $\mathcal{A}$ with a random looking label $\mathtt{id}$ on which the delay function is programmed to output $\widehat{y}$. If $\mathcal{A}$ lies in a class $\mathcal{C}$ and the delay function is $\mathcal{C}$-programmable, the adversary is not able to detect the change. We conclude by observing that, in this hybrid world, since the statement under which we generate the witness encryption ciphertext is false, the privacy of the plaintext is preserved.

There are of course efficiency requirements that our scheme needs to satisfy. We would like the witness encryption scheme to have quick decryption once a witness for the statement is known.

Notice that this property does not contradict the security of our delay encryption construction as long as any such witness is slow to derive. Furthermore, we would like the witness encryption scheme to have also a fast encryption algorithm. Finally, we would like the ciphertext to be compact in the sense that the ciphertext size is quasi independent of the time required for their decryption.

Assuming evasive LWE [VWW22], it is possible to build a witness encryption scheme with some of the desired properties. Both encryption and decryption can be computed in low depth, but unfortunately the ciphertexts are not compact. In the construction of Vaikuntanathan, Wee and Wichs [VWW22], the ciphertext size grows polynomially in the witness size. It would therefore be desirable that the size of the witness grows (poly)logarithmically in the time required to evaluate the delay function. In particular this means that our witness cannot be the list of gate values in the delay function evaluation. One could try to solve this problem by relying on succinct proofs, but this would quickly run into another problem: in general, succinct proofs are not perfectly sound. That would compromise the security of the whole construction as witness encryption guarantees the privacy of the plaintext as long as there is no witness. For succinct proofs that are not perfectly sound, however, there will always be false statements for which there exist proofs, even if the latter are hard to find.

One potential solution out of this is to perhaps build "somewhere perfectly sound succinct proofs". Imagine a succinct proof scheme where the CRS hides a statement over which we achieve perfect soundness. In order to prove that an encryption under a false statement x guarantees privacy, we first make the succinct proof system perfectly sound on x by hiding it in the CRS, and then we rely on the security of witness encryption. In other words, this type of proof system would allow us to obtain delay encryption with ciphertext compactness from evasive LWE. We leave building such a proof system as an exciting open problem for future work.

Theorem 5. *Consider two circuit classes $\mathcal{C}_0, \mathcal{C}_1$, where $\mathcal{C}_0$ is nicely closed and $\mathcal{C}_1 \subseteq \mathcal{C}_0 \subseteq \mathsf{poly}(\lambda)$. Let $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Eval})$ be a $\mathcal{C}_0$-programmable, $\mathcal{C}_0$-sequential delay function with domain $\mathcal{X} := (\mathcal{X}_\lambda)_{\lambda \in \mathbb{N}}$ and range $\mathcal{Y} := (\mathcal{Y}_\lambda)_{\lambda \in \mathbb{N}}$. Let $(\widehat{y}_\lambda)_{\lambda \in \mathbb{N}}$ be a sequence of elements where $\widehat{y}_\lambda \in \mathcal{Y}_\lambda$ for every $\lambda \in \mathbb{N}$. Consider the language*

$$L := \{(\lambda, x, \mathtt{pp}) | x \in \mathcal{X}_\lambda, \mathsf{Eval}(\mathtt{pp}, x) \neq \widehat{y}_\lambda\} .$$

Let $\mathcal{R}$ be a relation for L. Let $\mathsf{Witness}$ be a polynomial time algorithm that, on input $(\lambda, x, \mathtt{pp}) \in L$, outputs w such that $((\lambda, x, \mathtt{pp}), w) \in \mathcal{R}$.

Let $\mathsf{WE} = (\mathsf{Enc}, \mathsf{Dec})$ be a witness encryption scheme for the relation $\mathcal{R}$ such that $\mathsf{Enc}, \mathsf{Dec} \in \mathcal{C}_1$. Then, the construction in Fig.9 is a $\mathcal{C}_0$-secure, $\mathcal{C}_1$-efficient, correct delay encryption scheme.

DELAY ENCRYPTION FROM WITNESS ENCRYPTION

$\mathsf{Setup}(1^\lambda)$: Output $\mathtt{pp} \leftarrow\!\!\$\ \mathsf{DF.Setup}(1^\lambda)$
$\mathsf{Enc}(\mathtt{pp}, \mathtt{id}, m)$: Output $c \leftarrow\!\!\$\ \mathsf{WE.Enc}(1^\lambda, (\lambda, \mathtt{id}, \mathtt{pp}), m)$
$\mathsf{Ext}(\mathtt{pp}, \mathtt{id})$: Output $\mathtt{idk} := w \leftarrow \mathsf{Witness}(\lambda, \mathtt{id}, \mathtt{pp})$
$\mathsf{Dec}(\mathtt{pp}, \mathtt{idk} := w, c)$: Output $m \leftarrow \mathsf{WE.Dec}(c, w)$

Fig. 9. Delay encryption from witness encryption

Proof. It is trivial to see that the scheme is $\mathcal{C}_1$-efficient. Correctness is also straightforward: thanks to $\mathcal{C}_0$-sequentiality, we know that, for a random $\mathtt{id}$, $\mathsf{Eval}(\mathtt{pp}, \mathtt{id}) \neq \widehat{y}_\lambda$ with overwhelming probability. Therefore, the property follows immediately from the correctness of the witness encryption scheme and the fact that $\mathsf{Witness}$ outputs the witness for the statement provided as input.

We therefore focus on security. Suppose that there exists an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathcal{C}_0$ that wins the delay encryption security game with non-negligible advantage. We show that this leads to a contradiction by a series of hybrids.

Hybrid 0. This hybrid corresponds to the original delay encryption security game D-IND-CPA (see Fig. 14).

Hybrid 1. In this hybrid, we change the distribution of the public parameters $\mathtt{pp}$ and of the identity $\mathtt{id}$ given as a challenge. Specifically, we generate the pair $(\mathtt{pp}, \mathtt{id})$ using $\mathsf{Program}(1^\lambda, \widehat{y}_\lambda)$. Since $\mathcal{A}_1 \in \mathcal{C}_0$, $\mathsf{WE.Enc}$ is in $\mathcal{C}_1 \subseteq \mathcal{C}_0$ and $\mathcal{C}_0$ is nicely closed, we conclude that the advantage of $\mathcal{A}$ in this hybrid is still non-negligible.

Hybrid 2. In this hybrid, we change the distribution of the ciphertext given to the adversary $\mathcal{A}_1$. Specifically, we compute it using $\mathsf{WE.Enc}((\lambda, \mathtt{id}, \mathtt{pp}), m_0)$, independently of the value of b. Notice that $(\lambda, \mathtt{id}, \mathtt{pp})$ is not in the language L, so the advantage of $\mathcal{A}$ is still non-negligible thanks to the security of witness encryption. We have reached a contradiction. Indeed, in Hybrid 2, all the information received by $\mathcal{A}$ is independent of b. □

5.2 Delay Encryption from Attribute-Based Encryption

Following a similar blueprint, we show how to build delay encryption from delay functions and attribute-based encryption schemes satisfying a particular condition we called it *decryption amortisability*. Specifically, we require that, given a decryption key associated with a function f, the master public key mpk and an attribute x, we are able to preprocess a fast decryption key fsk, so that, whenever we are given an ABE ciphertext under the attribute x, we are able to quickly decrypt it using fsk. We formalise the idea below. As we will explain later, the LWE-based ABE scheme of Boneh et al. [BGG+14] satisfies the desired property.

Definition 36 (Decryption-amortisable ABE). *Let $\mathcal{C}$ be a circuit class. A $\mathcal{C}$-decryption-amortisable ABE scheme is a tuple of PPT algorithms* $(\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Ext}, \mathsf{FastDec})$ *satisfying the following properties:*

(Correctness). *For every function $f : \{0,1\}^n \to \{0,1\}$ and $x \in \{0,1\}^n$ such that $f(x) = 1$ and message m,*

$$\Pr\left[\begin{array}{l}(\mathsf{mpk}, \mathsf{msk}) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)\\ \mathsf{sk}_f \leftarrow\!\!\$\ \mathsf{KeyGen}(\mathsf{msk}, f)\\ c \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{mpk}, x, m)\\ \mathsf{fsk} \leftarrow \mathsf{Ext}(\mathsf{mpk}, \mathsf{sk}_f, x)\end{array} : \mathsf{FastDec}(c, \mathsf{fsk}) = m\right] = 1$$

(Selective security). *We say that an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ is valid if, $\mathcal{A}_0$ always outputs a tuple $(f, x, m_0, m_1, \mathsf{aux})$ where $f : \{0,1\}^n \to \{0,1\}$ is a function such that $f(x) = 0$. The ABE scheme is selectively secure if for every valid adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathsf{poly}(\lambda)$, it holds that*

$$\mathsf{Adv}^{\mathsf{SEL}}_{\mathsf{ABE}}(\mathcal{A}) := \left|\Pr[\mathsf{Exp}^{\mathsf{SEL}}_{\mathcal{A},\mathsf{ABE}}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda),$$

where experiment $\mathsf{Exp}^{\mathsf{SEL}}_{\mathcal{A},\mathsf{ABE}}$ is specified in Figure 10.

($\mathcal{C}$-Efficiency). *The algorithms* Enc *and* FastDec *belong to* $\mathcal{C}$.

$\mathsf{Exp}^{\mathsf{SEL}}_{\mathcal{A},\mathsf{ABE}}(1^\lambda)$

1 : $b \leftarrow\$ \{0,1\}$
2 : $(\mathsf{mpk}, \mathsf{msk}) \leftarrow\$ \mathsf{Setup}(1^\lambda)$
3 : $(f, x, m_0, m_1, \mathsf{aux}) \leftarrow\$ \mathcal{A}_0(1^\lambda, \mathsf{mpk})$
4 : $\mathsf{sk}_f \leftarrow\$ \mathsf{KeyGen}(\mathsf{msk}, f)$
5 : $c \leftarrow\$ \mathsf{Enc}(\mathsf{mpk}, x, m_b)$
6 : $b' \leftarrow\$ \mathcal{A}_1(c, \mathsf{sk}_f, \mathsf{aux}_0)$
7 : **return** $b = b'$

Fig. 10. Selective security game SEL.

As we already mentioned, the ABE scheme for generic circuits built by Boneh et al. [BGG+14] is decryption-amortisable. We informally recap the construction and we explain why it satisfies our desired property. The master public key of the ABE scheme consists of $n+1$ "fat" LWE matrices $A_0, A_1, \ldots, A_n$ over a ring $\mathbb{Z}_q$. In order to encrypt a bit $m \in \{0,1\}$ under a attribute $x \in \{0,1\}^n$, we sample a random vector s and low-norm vectors $e_0, \ldots, e_n$. The ciphertext will consist of the list of $n+1$ vectors $(c_0, \ldots, c_n)$ where

$$c_i = (A_i + x_i \cdot G)^\intercal \cdot s + e_i \qquad \text{for } i = 1, 2, \ldots, n$$
$$c_0 = A_0^\intercal \cdot s + e_0 + \left\lceil \frac{q}{2} \right\rfloor \cdot m$$

Above, G denotes the gadget matrix.

In [BGG+14], the authors showed that, for any function $f : \{0,1\}^n \to \{0,1\}$, by applying linear operations depending only on $A_1, \ldots, A_n, f$ and x on $(c_1, \ldots, c_n)$, it is possible to derive a vector c_f of the form $(A_f + f(x) \cdot G)^\intercal \cdot s + e_f$, where e_f is a small-norm vector and A_f is a "fat" matrix that can be easily computed from $A_1, \ldots, A_n$ and f. The decryption key associated with f is a small-norm matrix such that $A_f \cdot \mathsf{sk}_f = A_0$. In other words, if $f(x) = 0$, it is possible to retrieve m by rounding $c_0 - \mathsf{sk}_f^\intercal \cdot c_f$ to the closest multiple of $\lceil q/2 \rfloor$.

Notice that linear operations and rounding can be performed in logarithmic depth. So, we can split the decryption of ciphertexts under the attribute x with respect to a secret key sk_f into two phases: first, given the matrices $A_1, \ldots, A_n$, the function f and the attribute x, we derive the matrix M describing the linear operations we need to apply on the vectors $(c_1, \ldots, c_n)$. Then, given the ciphertexts $c_0, \ldots, c_n$, we compute the plaintext $m = \mathsf{Round}\Big(c_0 - \mathsf{sk}_f^\intercal \cdot M \cdot (c_1, \ldots, c_n)\Big)$. While the first operation can be "slow" (depending on the function f), the second phase can always be computed "quickly" by a circuit in NC_1. To summarise, the fast decryption key relative to the function f and the attribute x will be $\mathsf{sk}_f^\intercal \cdot M$.

Lemma 6 ([BGG+14]). *Let* $\mathsf{ABE} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ *be the attribute-based encryption scheme of [BGG+14]. Let* $\mathsf{Eval}_{\mathsf{ct}}$ *be the algorithm used during the decryption procedure (see [BGG+14, page 16]). Then, there exists a polynomial-time algorithm* ABE.PreComp *that, on input the public*

parameters mpk*, a function* $f : \{0,1\}^n \to \{0,1\}$ *and an attribute* x*, outputs a matrix* M *such that, for every ciphertext* c*,*

$$\Pr\begin{bmatrix}(\mathsf{mpk}, \mathsf{msk}) \leftarrow\!\!\$\ \mathsf{ABE.Setup}(1^\lambda) & \\ \mathsf{sk}_f \leftarrow\!\!\$\ \mathsf{ABE.KeyGen}(\mathsf{msk}, f) & : M \cdot c = \mathsf{Eval}_{\mathsf{ct}}(\mathsf{mpk}, f, x, c) \\ M \leftarrow \mathsf{ABE.PreComp}(\mathsf{mpk}, f, x) & \end{bmatrix} = 1$$

Corollary 7. *Consider the ABE scheme obtained by augmenting the construction of [BGG+14] with the following procedures:*

- $\mathsf{ABE.Ext}(\mathsf{mpk}, \mathsf{sk}_f, x)$
  1. $M \leftarrow \mathsf{ABE.PreComp}(\mathsf{mpk}, f, x)$
  2. *Output* $\mathsf{fsk} := \mathsf{sk}_f^\mathsf{T} \cdot M$
- $\mathsf{ABE.FastDec}(c, \mathsf{fsk})$*:*
  1. $z \leftarrow \mathsf{fsk} \cdot c$
  2. *Output* $\mathsf{Round}(v)$

Under the security of LWE with subexponential modulus-to-noise ratio, the scheme described above is NC_1*-decryption-amortisable.*

Building Delay Encryption from Decryption-Amortisable ABE. It is now time to show how to build delay encryption from ABE. The idea is the same as for witness encryption. The public parameters of the scheme will include the description of a programmable delay function along with a value $\widehat{y}$ in its range. Furthermore, the public parameters will also include an ABE secret-key sk_f for the function that, on input a attribute x, evaluates the delay function on it and outputs 1 if and only if the result is different from $\widehat{y}$. The rest is fairly straightforward: when we want to encrypt a message m under a random label `id`, we just perform an ABE encryption of m using `id` as attribute.

With overwhelming probability, the evaluation of the delay function on `id` will return a value different from $\widehat{y}$, so the parties can retrieve the plaintext using sk_f. On the other hand, if the delay function is $\mathcal{C}$-programmable, no adversary in $\mathcal{C}$ would be able to distinguish the real world from an hybrid in which we provide a label `id` for which the delay function is programmed to output $\widehat{y}$. In such setting, the attribute under which the ciphertext is generated does not satisfy the policy f, so the security of ABE guarantees the privacy of the plaintext.

As for the efficiency properties, we notice that once the parties have derived the fast decryption key relative to f and the label `id`, they can decrypt all encryptions under `id` quickly by evaluating low-depth circuits. Moreover, by the efficiency properties of the decryption-amortisable ABE scheme, the encryption algorithm has low circuit depth.

Unfortunately, the scheme does not satisfy ciphertext compactness. We would like the size of the ABE ciphertexts to be independent of the depth of the policy f, but their construction has a ciphertext size that grows linearly in depth of f. A recent work by Hsieh, Lin, and Luo [HLL23] shows how to construct an ABE scheme with a ciphertext size that does not grow with the depth of f, but to the best of our knowledge their construction is not decryption-amortisable. If, one day, we manage to build a decryption-amortisable ABE scheme where the ciphertext size is independent of the depth of the policies, we would immediately obtain delay encryption schemes with ciphertext compactness.

Delay Encryption from Attribute-Based Encryption

Let $f_\lambda^{\mathsf{pp},\widehat{y}}$ be the function that on input x, outputs 1 if and only if $\mathsf{DF.Eval}(\mathsf{pp}, x) \neq \widehat{y}$.

$\mathsf{Setup}(1^\lambda)$:

1. $\mathsf{pp}' \leftarrow\$ \mathsf{DF.Setup}(1^\lambda)$
2. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow\$ \mathsf{ABE.Setup}(1^\lambda)$
3. $\mathsf{sk} \leftarrow \mathsf{ABE.KeyGen}(\mathsf{msk}, f_\lambda^{\mathsf{pp}',\widehat{y}_\lambda})$
4. Output $\mathsf{pp} := (\mathsf{mpk}, \mathsf{sk})$

$\mathsf{Enc}(\mathsf{pp} = (\mathsf{mpk}, \mathsf{sk}), \mathtt{id}, m)$: Output $c \leftarrow\$ \mathsf{ABE.Enc}(\mathsf{mpk}, \mathtt{id}, m)$

$\mathsf{Ext}(\mathsf{pp} = (\mathsf{mpk}, \mathsf{sk}), \mathtt{id})$: Output $\mathtt{idk} := \mathsf{fsk} \leftarrow \mathsf{ABE.Ext}(\mathsf{mpk}, \mathsf{sk}, \mathtt{id})$

$\mathsf{Dec}(\mathsf{pp}, \mathtt{idk}, c)$: Output $m \leftarrow \mathsf{ABE.FastDec}(c, \mathtt{idk})$

Fig. 11. Delay encryption from attribute-based encryption

Theorem 8. *Let $\mathcal{C}_0$ and $\mathcal{C}_1$ be circuit classes where $\mathcal{C}_1 \subseteq \mathcal{C}_0 \subseteq \mathsf{poly}(\lambda)$ and $\mathcal{C}_0$ is nicely closed.*

Let $\mathsf{ABE} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Ext}, \mathsf{FastDec})$ be a $\mathcal{C}_1$-description-amortisable ABE scheme. Let $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Eval})$ be a $\mathcal{C}_0$-programmable, $\mathcal{C}_0$-sequential delay function with domain $\mathcal{X} := (\mathcal{X}_\lambda)_{\lambda\in\mathbb{N}}$ and range $\mathcal{Y} := (\mathcal{Y}_\lambda)_{\lambda\in\mathbb{N}}$. Let $(\widehat{y}_\lambda)_{\lambda\in\mathbb{N}}$ be a sequence of elements where $\widehat{y}_\lambda \in \mathcal{Y}_\lambda$ for every $\lambda \in \mathbb{N}$. Then the scheme in Fig.11 is a $\mathcal{C}_0$-secure, $\mathcal{C}_1$-efficient, correct delay encryption scheme.

Proof. It is trivial to see that the scheme is $\mathcal{C}_1$-efficient. Correctness is also straightforward: thanks to $\mathcal{C}_0$-sequentiality, we know that, for a random $\mathtt{id}$, $\mathsf{Eval}(\mathsf{pp}', \mathtt{id}) \neq \widehat{y}_\lambda$ with overwhelming probability. Therefore, the property follows immediately from the correctness of the ABE scheme.

We therefore focus on security. Suppose that there exists an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathcal{C}_0$ that wins the delay encryption security game with non-negligible advantage. We show that this leads to a contradiction by a series of hybrids.

Hybrid 0. This hybrid corresponds to the original delay encryption security game D-IND-CPA (see Fig. 14).

Hybrid 1. In this hybrid, we change the distribution of the public parameters pp' and of the identity $\mathtt{id}$ given as a challenge. Specifically, we generate the pair $(\mathsf{pp}', \mathtt{id})$ using $\mathsf{Program}(1^\lambda, \widehat{y}_\lambda)$. Since $\mathcal{A}_1 \in \mathcal{C}_0$, $\mathsf{ABE.Enc}$ is in $\mathcal{C}_1 \subseteq \mathcal{C}_0$ and $\mathcal{C}_0$ is nicely closed, we conclude that the advantage of $\mathcal{A}$ in this hybrid is still non-negligible.

Hybrid 2. In this hybrid, we change the distribution of the ciphertext given to the adversary $\mathcal{A}_1$. Specifically, we compute it using $\mathsf{ABE.Enc}(\mathsf{mpk}, \mathtt{id}, m_0)$, independently of the value of b. Notice that $f_\lambda^{\mathsf{pp}',\widehat{y}_\lambda}(\mathtt{id}) = 0$, so the advantage of $\mathcal{A}$ is still non-negligible thanks to the selective security of the ABE scheme. We have reached a contradiction. Indeed, in Hybrid 2, all the information received by $\mathcal{A}$ is independent of b. □

6 Randomness Beacons do not Need VDFs

In this section, we show how to build randomness beacons based on timing assumptions without making use of VDFs. We just need to rely on time-lock puzzles. Furthermore, our constructions achieve better asymptotic complexity compared to the solution based on VDFs, both in terms of rounds and computation. In particular, the schemes we present are non-interactive: after publishing their block, the parties do not need to speak again.

Given that randomness beacons were never precisely defined [BBBF18], we start by formalising the primitive and its security property. Then, we present two ways to instantiate the primitive.

The first one is less efficient and is based on generic time-lock puzzles [RSW96], the second one, achieving better computational complexity, is based on additively homomorphic time-lock puzzles [MT19].

6.1 Defining Randomness Beacons

We now formalise randomness beacons and their security properties. Our definition, in particular the syntax, reflects the characteristics of the constructions we are going to present later: non-interactiveness, efficient verifiability and security in the programmable random oracle model.

Randomness beacons are protocols for the generations of randomness on a blockchain. In particular, the primitive specifies how to generate blocks to be published on the chain. From any sequence of T blocks it is possible to derive a random string of bits. Although the computational costs of this operation is relatively *low*, the procedure is *slow*: the algorithm producing the output is highly non-parallelisable. This ensure that any adversary is not able to learn how to bias the randomness produced by the beacon before an honest party publishes a new block. For intuition, imagine an adversary that would like to bias the first bit of the next string produced by the beacon towards 0: after learning the first $T-1$ blocks, the adversary generates in its head the last block many times with the intention of publishing one that makes the first bit of the output equal to 0. However, checking which blocks satisfy the desired property takes time, in particular long enough that, with extremely high probability, an honest party would end up publishing the last block first. The adversary is therefore forced to publish a block without knowing how this will bias the output.

We finally present our definition of non-interactive, verifiable randomness beacon. The primitive is non-interactive in the sense that it does not require any further communication from the parties after they publish their block. Furthermore, it is verifiable meaning that, when T blocks are processed to generate a random string, the procedure provides also an efficiently and quickly verifiable proof of correct evaluation. In other words, the slow operations necessary to determine the next random string produced by the beacon can be performed by a single party, which can then quickly convince all other participants of the correct result.

Definition 37 (Non-interactive Verifiable Randomness Beacon). *A non-interactive verifiable randomness beacon consists of a tuple of PPT algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{FastSolve}, \mathsf{Verify})$ *with the following syntax:*

$\mathsf{Setup}(1^\lambda)$**:** *The setup algorithm takes as input the security parameter* 1^λ *and outputs public parameters* pp

$\mathsf{Gen}(\mathsf{pp})$**:** *The generation algorithm takes as input the public parameters* pp*. The output is a block* B*.*

$\mathsf{Solve}(\mathsf{pp}, k, B_1, \dots, B_T)$**:** *The solving algorithm is deterministic and takes as input the public parameters* pp*, a block index* k *and blocks* $B_1, \dots, B_T$ *for some* $T \in \mathbb{N}$*. The output is a solution* r *and a proof* w*.*

$\mathsf{FastSolve}(\mathsf{pp}, k, T, B_k, \mathsf{aux}_{k-1})$**:** *The fast solving algorithm is deterministic and takes as input the public parameters* pp*, a block index* k*, a block number* T*, a block* B_k *and auxiliary information* aux_{k-1}*. The output is a solution* r*, a proof* w *and auxiliary information* aux_k*.*

$\mathsf{Verify}(\mathsf{pp}, k, B_1, \dots, B_T, r, w)$**:** *The verification algorithm is deterministic and takes as input the public parameters* pp*, a block index* k*, blocks* $B_1, \dots, B_T$*, a solution* r *and a proof* w*. The output is a bit* b *expressing whether* r *is a solution for* $B_1, \dots, B_T$ *or not.*

In the above definition, we equipped the primitive with two algorithms for the computation of the random strings produced by the beacon. The first one, Solve, is stateless and processes T blocks at the time, returning the random string they generate. The second one, $\mathsf{FastSolve}$, instead streams over the blockchain, processing the blocks one at the time and returning the corresponding random strings as the information is published on the chain. This second algorithm keeps an internal state, represented by the auxiliary information aux, which is updated and passed on from execution to execution. As the name suggest, $\mathsf{FastSolve}$ usually allows to stream through the outputs of the beacon with higher efficiency.

Below, we formalise the completeness and the correctness of the randomness beacon. The former guarantees that the proofs output by the solving algorithm always verify. The latter ensures instead that the output of Solve and $\mathsf{FastSolve}$ always coincide.

Definition 38 (Completeness). *A non-interactive verifiable randomness beacon* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{FastSolve}, \mathsf{Verify})$ *is complete if, for every* $\lambda, T, k \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{ll}\mathsf{pp} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda) & \\ \forall i \in [T] : B_i \leftarrow\!\!\$\ \mathsf{Gen}(\mathsf{pp}) & : \mathsf{Verify}(\mathsf{pp}, k, B_1, \ldots, B_T, r, w) = 1 \\ (r, w) \leftarrow \mathsf{Solve}(\mathsf{pp}, k, B_1, \ldots, B_T) & \end{array}\right] = 1$$

Definition 39 (Correctness). *A non-interactive verifiable randomness beacon* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{FastSolve}, \mathsf{Verify})$ *is perfectly correct if, for every* $\lambda, M, T \in \mathbb{N}$ *and* $\mathsf{pp}, B_1, \ldots, B_M$*, it holds that*

$$\Pr\left[\begin{array}{ll}\mathsf{aux}_0 \leftarrow \bot & \\ \forall i \in [M] : (r_i, w_i, \mathsf{aux}_i) \leftarrow \mathsf{FastSolve}(\mathsf{pp}, i, T, B_i, \mathsf{aux}_{i-1}) & : (r_i, w_i) = (r'_i, w'_i)\ \ \forall i = T, \ldots, M \\ \forall i \in [M] : (r'_i, w'_i) \leftarrow \mathsf{Solve}(\mathsf{pp}, i, B_{i-T+1}, \ldots, B_i) & \end{array}\right] = 1.$$

We define efficiency of randomness beacons with respect to circuit classes $\mathcal{C}_1 \subseteq \mathcal{C}_2, \mathcal{C}_3$. The block generation algorithm and the verification belong to the class of quickly computable circuits $\mathcal{C}_1$, whereas Solve and $\mathsf{FastSolve}$ belong to the more slower classes $\mathcal{C}_2$ and $\mathcal{C}_3$ respectively.

Definition 40 (Efficiency). *Let* $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ *be circuit classes. We say that a non-interactive verifiable randomness beacon* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{FastSolve}, \mathsf{Verify})$ *is* $(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3)$*-efficient if* $\mathsf{Gen}, \mathsf{Verify} \in \mathcal{C}_1$*,* $\mathsf{Solve} \in \mathcal{C}_2$*,* $\mathsf{FastSolve} \in \mathcal{C}_3$*.*

Next, we formalise the security properties of the primitive. We start from soundness. The latter ensures that no efficient adversary (even one that is allowed to performed highly non-parallelisable operations) can generate valid proofs for any beacon outputs other than the correct ones.

Definition 41 (Soundness). *A non-interactive verifiable randomness beacon* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{FastSolve}, \mathsf{Verify})$ *is sound if, for every PPT adversary* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{ll}\mathsf{pp} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda) & \\ (k, B_1, \ldots, B_T, r, w) \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, \mathsf{pp}) & : \begin{array}{c}\mathsf{Verify}(\mathsf{pp}, k, B_1, \ldots, B_T, r, w) = 1 \\ r \neq r'\end{array} \\ (r', w') \leftarrow \mathsf{Solve}(\mathsf{pp}, k, B_1, \ldots, B_T) & \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Finally, we define simulation security in the programmable random oracle model. We define the security property by relying on a real world - ideal world paradigm parametrised by three values: $\mathcal{C}$, describing the efficiency class of the adversary and time parameters $T, D \in \mathbb{N}$.

In the real world, after being provided with the public parameters of the randomness beacon, the adversary is repeatedly faced with the decision of whether it wants the next block to be generated by the honest parties according to the protocol or whether it wants to publish a possibly malicious block of its own choice. With the delay of D blocks (used to model the time necessary for the honest parties to run $\mathsf{FastSolve}$ or Solve), the adversary is also gradually provided with the randomness beacon outputs and proofs confirming their validity. Notice that the adversary would be able to compute these values on its own by relying on Solve or $\mathsf{FastSolve}$.

In the ideal world, on the other hand, the adversary interacts with simulators that model the blockchain execution and the random oracle responses. Every time a new block B is published, the simulators are provided with a new random string r helping them simulate the real world. Upon receiving r, the simulators get to immediately choose a proof w confirming its validity. The pair (r, w) are revealed to the adversary with a delay of D blocks.

We ask that for any T-respecting adversary in $\mathcal{C}$, meaning an adversary which ensures that, for any sequence of T consecutive blocks, at least one is honestly generated, the two worlds are indistinguishable.

Definition 42 (($\mathcal{C}, T, D$)-simulation security). *Let* $\mathsf{RB} := (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{FastSolve}, \mathsf{Verify})$ *be a non-interactive verifiable randomness beacon where* Ext *outputs* $L(\lambda)$*-bit strings. Let* $\mathcal{H}$ *be a random oracle outputting* $n(\lambda)$*-bit responses.*

We say that an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathcal{C}$ *for the experiment* $\mathsf{Exp}^{\mathsf{RBS},T,D}_{\mathcal{A},\mathsf{Sim},\mathsf{RB}}$ *(see Figure 12) is* T*-respecting if it calls the procedure* ***New Block*** *at most a polynomial number of times and never outputs* $c = 1$ *in* T *consecutive calls to such procedure.*

We say that the scheme is $(\mathcal{C}, T(\lambda), D(\lambda))$*-simulation secure in the programmable random oracle model, if, for every* T*-respecting adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathcal{C}$*, there exists a simulator* $\mathsf{Sim} := (\mathsf{Sim}_0, \mathsf{Sim}_1, \mathsf{SimRO}) \in \mathsf{poly}(\lambda) \times \mathcal{C} \times \mathcal{C}$ *such that*

$$\mathsf{Adv}^{\mathsf{RBS},T,D}_{\mathsf{RB},\mathsf{Sim}}(\mathcal{A}) := \left| \Pr[\mathsf{Exp}^{\mathsf{RBS},T,D}_{\mathcal{A},\mathsf{Sim},\mathsf{RB}}(1^\lambda) = 1] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda).$$

The above definition essentially guarantees that as long as we are guaranteed the existence of an honest block every T blocks, the outputs produced by the beacon look random. Furthermore, at the time the i-th block is published on the blockchain, the $(i+1)$-th output of the beacon, as well as all the following ones, are unpredictable. Notice, however, that the adversary may discover the beacon outputs D blocks in advance. In other words, by the time the honest parties discover the i-th output, the adversary may already know also the following D outputs.

We highlight that the above form of simulation-based security definition for randomness beacons is achievable only in the random oracle model.

6.2 Constructing Randomness Beacons from Time-Lock Puzzles

Our first randomness beacon construction is described in Fig.13. The idea is pretty simple: each block of the randomness beacon includes of a time-lock puzzle z hiding a high-entropy random string s. The output of the beacon are obtained by solving the time-lock puzzles in the last T blocks and querying the random oracle with the concatenation of the resulting strings.

In order to obtain fast verifiability, we augment every block with a commitment c to the string s. We hide the opening v of such commitment inside z so that, once the time-lock puzzle is solved, the

SIMULATION SECURITY FOR RANDOMNESS BEACONS $\mathsf{Exp}^{\mathsf{RBS},T,D}_{\mathcal{A},\mathsf{Sim},\mathsf{RB}}$

Initialisation: This procedure is run only once at the beginning of the game.

1. $b \leftarrow\$ \{0,1\}$
2. $i \leftarrow 1$
3. $Q \leftarrow \emptyset$
4. $\mathsf{pp}_0 \leftarrow\$ \mathsf{Setup}(1^\lambda)$
5. $(\mathsf{pp}_1, \phi) \leftarrow\$ \mathsf{Sim}_0(1^\lambda)$
6. $\psi \leftarrow \mathcal{A}_0^{\mathcal{H}}(1^\lambda, \mathsf{pp}_b)$
7. $B_0 \leftarrow \bot$
8. $r_{-T+1}, r_{-T+2}, \ldots, r_{-1}, r_0, r_1, \ldots, r_{T-1} \leftarrow \bot$

New Block: This procedure is run repeatedly after the initialisation, until the adversary halts.

1. $(c, \psi) \leftarrow\$ \mathcal{A}_1^{\mathcal{H}}(B_{i-1}, r_{i-D}, w^b_{i-D}, \psi)$
2. $r_i^1 \leftarrow\$ \{0,1\}^{L(\lambda)}$
3. If $c = 0$:
   (a) $B_i^0 \leftarrow\$ \mathsf{Gen}(\mathsf{pp}_0)$
   (b) $(B_i^1, w_i^1, \phi) \leftarrow\$ \mathsf{Sim}_1(\phi, \bot, r_i^1)$
   (c) $B_i \leftarrow B_i^b$
4. If $c = 1$:
   (a) $(B_i, \psi) \leftarrow\$ \mathcal{A}_1^{\mathcal{H}}(\psi)$
   (b) $(w_i^1, \phi) \leftarrow\$ \mathsf{Sim}_1(\phi, B_i, r_i^1)$
5. $(r_i^0, w_i^0) \leftarrow \mathsf{Solve}(\mathsf{pp}_b, B_{i-T+1}, \ldots, B_i)$
6. If $i \geq T$: $r_i \leftarrow r_i^b$
7. $i \leftarrow i + 1$

Oracle $\mathcal{H}$: The adversary has unbounded access to this oracle. Let $\widehat{x}$ be the queried valued.

1. If there exists a pair $(x, y) \in Q$ such that $x = \widehat{x}$, output y.
2. $y_0 \leftarrow\$ \{0,1\}^{n(\lambda)}$
3. $(y_1, \phi) \leftarrow\$ \mathsf{SimRO}(\phi, \widehat{x})$
4. $Q \leftarrow Q \cup \{(\widehat{x}, y_b)\}$
5. Output y_b

Output: This procedure is run only once when the adversary halts outputting a bit b'. Output $b = b'$.

Fig. 12. Randomness beacon security game RBS.

parties not-only obtain s but also v. In this way, anybody can efficiently and quickly verify the result of the computation validating it on c. Finally, we juxtapose a NIZK proving the well-formedness of the pair (c, z).

The reason why this scheme satisfies simulation security is that, in order to learn a beacon output, an adversary needs to solve all honestly generated time-lock puzzles in the previous T blocks. We choose the parameters of the puzzles so that solving them requires long enough that, almost certainly, other $T-1$ blocks will be published on the chain in the meantime. In other words, even if the adversary will decide to generate some of the T blocks from which the output is derived maliciously, its choice will be made obliviously of the random oracle query producing the result: the adversary would have no time to solve all the puzzles generated by the honest parties. Since the oracle query is highly unpredictable, the simulator is able to program the output of the random oracle without the adversary spotting any inconsistency.

Theorem 9. *Let $f = \Omega(\log \lambda)$ be a polynomial function in the security parameter. Let* (Commit, Check) *be a perfectly binding, computationally hiding non-interactive commitment. For every $\tau \in \mathbb{N}$, let $\mathcal{C}'_\tau := \overline{\mathcal{C}}_\tau$ and define $\mathcal{C}'$ as $(\mathcal{C}'_\tau)_{\tau \in \mathbb{N}}$. Let* TLP *be a $(\mathcal{C}', \overline{\tau})$-secure time-lock puzzle scheme. Finally,*

RANDOMNESS BEACON FROM TIME-LOCK PUZZLES

$\mathsf{Setup}(1^\lambda)$:
1. $\sigma \leftarrow\$ \mathsf{NIZK.Setup}(1^\lambda)$
2. $\mathsf{pp}' \leftarrow\$ \mathsf{TLP.Setup}(1^\lambda, 1^\tau)$
3. Output $\mathsf{pp} := (\mathsf{pp}', \sigma)$.

$\mathsf{Gen}(\mathsf{pp} = (\mathsf{pp}', \sigma))$:
1. $s \leftarrow\$ \{0,1\}^\lambda$
2. $(c, v) \leftarrow\$ \mathsf{Commit}(s; u_0)$
3. $z \leftarrow\$ \mathsf{TLP.Gen}(1^\lambda, \mathsf{pp}', (s, v); u_1)$
4. $\pi \leftarrow\$ \mathsf{NIZK.Prove}(\sigma, (z, c, \mathsf{pp}'), (s, v, u_0, u_1))$
5. Output $B := (c, z, \pi)$

$\mathsf{Solve}(\mathsf{pp} = (\mathsf{pp}', \sigma), k, B_1 = (c_1, z_1, \pi_1), \ldots, B_T = (c_T, z_T, \pi_T))$:
1. $\forall i \in [T] : b_i \leftarrow \mathsf{NIZK.Verify}(\sigma, (z_i, c_i, \mathsf{pp}'), \pi_i)$
2. $S \leftarrow \{i | b_i \neq 0\}$
3. $\forall i \in S : (s_i, v_i) \leftarrow \mathsf{TLP.Solve}(\mathsf{pp}', z_i)$
4. $\forall i \in [T] \setminus S : s_i \leftarrow \perp$
5. Output $r := \mathcal{H}(k \parallel s_1 \parallel \ldots \parallel s_T)$ and $w := (s_1, \ldots, s_T, v_1, \ldots, v_T)$.

$\mathsf{FastSolve}(\mathsf{pp} = (\mathsf{pp}', \sigma), k, T, B = (c, z, \pi), \mathsf{aux} = ((s_1, v_1), \ldots, (s_{T-1}, v_{T-1})))$:
1. $b \leftarrow \mathsf{NIZK.Verify}(\sigma, (z, c, \mathsf{pp}'), \pi)$
2. If $b = 0$, $s_T \leftarrow \perp$
3. If $b = 1$, $(s_T, v_T) \leftarrow \mathsf{TLP.Solve}(\mathsf{pp}', z)$
4. Output $r := \mathcal{H}(k \parallel s_1 \parallel \ldots \parallel s_T)$, $w := (s_1, \ldots, s_T, v_1, \ldots, v_T)$, $\mathsf{aux}' := ((s_2, v_2), \ldots, (s_T, v_T))$.

$\mathsf{Verify}(\mathsf{pp} = (\mathsf{pp}', \sigma), k, B_1 = (c_1, z_1, \pi_1), \ldots, B_T = (c_T, z_T, \pi_T), r, w = (s_1, \ldots, s_T, v_1, \ldots, v_T))$:
1. $\forall i \in [T] : b_i \leftarrow \mathsf{NIZK.Verify}(\sigma, (z_i, c_i, \mathsf{pp}'), \pi_i)$
2. $S \leftarrow \{i | b_i \neq 0\}$
3. $\forall i \in S : b_i' \leftarrow \mathsf{Check}(c_i, s_i, v_i)$
4. If $\exists i \in S$ such that $b_i' = 0$, output 0.
5. If $\exists i \in [T] \setminus S$ such that $s_i \neq \perp$, output 0
6. If $r = \mathcal{H}(k \parallel s_1 \parallel \ldots \parallel s_T)$, output 1. Otherwise, output 0.

Fig. 13. Randomness beacon from time-lock puzzles

let NIZK *be a zero-knowledge, simulation-extractable NIZK for the relation*

$$\mathcal{R} := \left\{ (z, c, \mathsf{pp}', \tau), (s, w, u_0, u_1) \middle| \begin{array}{l} (c, w) = \mathsf{Commit}(s; u_0) \\ z = \mathsf{TLP.Gen}(1^\tau, \mathsf{pp}', (s, w); u_1) \end{array} \right\}$$

Suppose that $\mathsf{NIZK.Sim}_2 \in \overline{\mathcal{C}}_{g_0}$*,* $\mathsf{NIZK.Ext} \in \overline{\mathcal{C}}_{g_1}$ *for functions* $g_0(\lambda)$ *and* $g_1(\lambda)$ *of the security parameter. Suppose also that* Commit*,* Check*,* TLP.Gen*,* NIZK.Prove*,* $\mathsf{NIZK.Verify} \in \mathcal{C}_f$*. Then, if*

$$\tau(\lambda) = g_0(\lambda) + (T(\lambda) - 1) \cdot (g_1(\lambda) + g(\lambda) \cdot \omega(\log \lambda)), \qquad \tau(\lambda) = \Omega(\overline{\tau}(\lambda)),$$

where $g(\lambda)$ *is a function of the security parameter, the scheme in Fig.13 is a non-interactive verifiable randomness beacon satisfying completeness, correctness, soundness,* $(\mathcal{C}_f, \mathcal{C}_\tau, \mathcal{C}_\tau)$*-efficiency and* $(\overline{\mathcal{C}}_g, T(\lambda), T(\lambda))$*-simulation security in the programmable random oracle model.*

Proof. We observe that our randomness beacon trivially satisfies completeness thanks to the completeness of the NIZK and the correctness of the commitment scheme and the time-lock puzzle. Also correctness and $(\mathcal{C}_f, \mathcal{C}_\tau, \mathcal{C}_\tau)$-efficiency is easily verifiable (notice that $\mathcal{C}_f$ and $\mathcal{C}_\tau$ are nicely closed). Soundness is instead implied by the soundness of the NIZK and the binding properties of the commitment. We therefore focus on simulation security.

We consider the simulator $\mathsf{Sim} = (\mathsf{Sim}_0, \mathsf{Sim}_1, \mathsf{SimRO})$ defined as follows:

- $\mathsf{Sim}_0(1^\lambda)$:
  1. $(\sigma, \zeta) \leftarrow\$ \mathsf{NIZK.Sim}_1(1^\lambda)$
  2. $\mathsf{pp}' \leftarrow\$ \mathsf{TLP.Setup}(1^\lambda, 1^\tau)$
  3. Output $\mathsf{pp} := (\mathsf{pp}', \sigma)$ and $\phi = (\mathsf{pp}', \sigma, \zeta)$
- $\mathsf{Sim}_1(\phi = (\mathsf{pp}', \sigma, \zeta, (j, r_j^1, s_j, v_j)_{j<i}), B_i, r_i^1)$:
  - If $B_i = \bot$:
    1. $s_i \leftarrow\$ \{0,1\}^\lambda$
    2. $(c_i, v_i) \leftarrow\$ \mathsf{Commit}(s_i; u_{i,0})$
    3. $z_i \leftarrow\$ \mathsf{TLP.Gen}(1^\lambda, \mathsf{pp}', (s_i, v_i); u_{i,1})$
    4. $\pi_i \leftarrow\$ \mathsf{NIZK.Prove}(\sigma, (z_i, c_i, \mathsf{pp}'), (s_i, v_i, u_{i,0}, u_{i,1}))$
    5. $w_i \leftarrow (s_{i-T+1}, \ldots, s_i, v_{i-T+1}, \ldots, v_i)$
    6. Output $B_i^1 := (c_i, z_i, \pi_i)$, w_i and $\phi := (\mathsf{pp}', \sigma, \zeta, (j, r_j^1, s_j, v_j)_{j \le i})$
  - If $B_i = (c_i, z_i, \pi_i) \ne \bot$:
    1. $(s_i, v_i, u_{i,0}, u_{i,1}) \leftarrow \mathsf{NIZK.Ext}(\zeta, (c_i, z_i, \mathsf{pp}'), \pi_i)$
    2. $w_i \leftarrow (s_{i-T+1}, \ldots, s_i, v_{i-T+1}, \ldots, v_i)$
    3. Output w_i and $\phi := (\mathsf{pp}', \sigma, \zeta, (j, r_j^1, s_j, v_j)_{j \le i})$
- $\mathsf{SimRO}(\phi, \widehat{x})$ rewrites $\widehat{x}$ as a string $(i \,\|\, s'_{i-T+1} \,\|\, \ldots \,\|\, s'_i)$. Then, it retrieves the tuples

  $$(i, r_i^1, s_i, v_i), (i-1, r_{i-1}^1, s_{i-1}, v_{i-1}), \ldots, (i-T+1, r_{i-T+1}^1, s_{i-T+1}, v_{i-T+1})$$

  stored in ϕ. Finally, if $s_{i-j+1} = s'_{i-j+1}$ for every $j \in [T]$, the algorithm outputs r_i^1. In all other cases, including when one of the above procedures fails, the algorithm outputs a random string in $\{0,1\}^{L(\lambda)}$.

We proceed by means of a sequence of hybrids.

Hybrid 0. This hybrid consists of the real world execution of the randomness beacon. In other words, the challenger of the simulation security game behaves as if $b = 0$.

Hybrid 1. In this hybrid, instead of generating σ using $\mathsf{NIZK.Setup}$, we compute it embedding a trapdoor in it. Specifically, we obtain the NIZK CRS as $(\sigma, \zeta) \leftarrow\!\$\ \mathsf{NIZK.Sim}_1(1^\lambda)$. This hybrid is computationally indistinguishable from Hybrid 0 thanks to zero-knowledge property of the NIZK.

Now, we define the event E_1 in which the adversary publishes a block where the corresponding NIZK verifies but the extraction of the witness fails. By the simulation extractability of the NIZK, the probability of the event E_1 occurring is negligible. This guarantees that the proofs w_i given to the adversary are consistent with the published blocks with overwhelming probability.

Now, we define the event E_2 as the event in which the adversary queries a tuple $(i \,\|\, s_1 \,\|\, \ldots \,\|\, s_T)$ to the random oracle, where s_j is the first part of the solution of the time-lock puzzle in B_{i-T+j} for every $j \in [T]$, before the block B_i is published.

We observe that the adversary can distinguish between Hybrid 1 and the ideal world execution of the security game (i.e. when $b = 1$) if and only if the probability of E_2 is non-negligible. Notice that the probability of E_2 in the two worlds are the same except for a negligible amount.

Suppose that the probability of E_2 in the ideal world is non-negligible. Since the adversary halts after a polynomial number of executions of the procedure **New Block**, there exists an index i_λ such that there exists a non-negligible probability that the adversary queries $(i_\lambda \,\|\, s_1 \,\|\, \ldots \,\|\, s_T)$ to the random oracle, where s_j is the first part of the solution of the time-lock puzzle in $B_{i_\lambda - T+j}$ for every $j \in [T]$, before the block B_{i_λ} is revealed. We call any of these queries *bad queries*. Let h be the highest index such that B_h is honestly generated and $h \leq i_\lambda$. Since the adversary is T-respecting, we know that $h \geq i_\lambda - T(\lambda) + 1$. We observe that the probability that the adversary issues a bad query before B_h is revealed is at most $2^{-\lambda}$. Indeed, the adversary would have no information about r_h which is sampled at random in $\{0,1\}^\lambda$. Therefore, the adversary must be able to issue a bad query in between the publication of B_h and B_{i_λ} with non-negligible probability.

Hybrid 2. In this hybrid, we change the distribution of the proof in the block $B_h = (z_h, c_h, \pi_h)$. In particular, we generate it as $\pi_h \leftarrow\!\$\ \mathsf{NIZK.Sim}_2(\zeta, (z_h, c_h, \mathsf{pp}'))$. Observe that this hybrid is indistinguishable from the previous one thanks to the zero-knowledge property of the NIZK. Therefore, the adversary issues a bad query in between the publication of B_h and B_{i_λ} with non-negligible probability. In particular, this means that there exists a non-negligible probability that the adversary issues an oracle query containing an opening of the commitment in B_h before B_{i_λ} is published.

Hybrid 3. In this hybrid, we change the distribution of the time-lock puzzle z_h. Specifically, we generate it as $z_h \leftarrow\!\$\ \mathsf{TLP.Gen}(1^\lambda, \mathsf{pp}', \perp)$. Thanks to the security of the time-lock puzzle and the fact that $\tau(\lambda) = g_0(\lambda) + (T(\lambda) - 1) \cdot (g_1(\lambda) + g(\lambda) \cdot \omega(\log \lambda))$, there is still a non-negligible probability that the adversary issues an oracle query containing an opening of the commitment in B_h before B_{i_λ} is published[7]. This contradict the hiding properties of the commitment scheme.

We conclude that the probability of the event E_2 in the ideal world is negligible. This implies that E_2 occurs with negligible probability even in Hybrid 1. This is enough to prove that Hybrid 1 is indistinguishable from the ideal execution of the security game. This ends the proof. □

6.3 A More Efficient Construction from Homomorphic Time-Lock Puzzles

The randomness beacon we just finished presenting suffers from some disadvantages. First of all, while the amortised computational cost is one time-lock puzzle per beacon output, retrieving a single beacon output requires solving T puzzles. Furthermore, the parties would receive the beacon's

[7] $g(\lambda)$ is multiplied by $\omega(\log \lambda)$ due to the depth of the operations computed by SimRO for each oracle query (i.e. $O(\log \lambda)$).

results with a delay of at least T blocks. We show how to improve our initial construction by relying on a variant of linearly homomorphic time-lock puzzles [MT19] we name *batched homomorphic time-lock puzzles*.

We start by defining and constructing such primitive. Later on, we show how the latter can be used to build a randomness beacon for which each output requires the resolution of a single time-lock puzzle. Furthermore, the delay with which the honest parties receive the inputs is significantly lower.

Batched Homomorphic Time-Lock Puzzles. A batched homomorphic time-lock puzzle is a primitive closely related to linearly homomorphic time-lock puzzles [MT19]: given any linear function F, it is still possible to "quickly" assemble time-lock puzzles $z_1, \dots, z_M$ with the same choice of time parameter into a new time-lock puzzle which hides the evaluation of F on the values hidden in $z_1, \dots, z_M$. We have, however, an additional property: given multiple time-lock puzzles $z_1, \dots, z_L$ with time parameters $T_1 \geq \cdots \geq T_L$ and hiding values $m_1, \dots, m_L$ in a group $\mathbb{Z}_q$, it is possible to solve for $m_1 + \cdots + m_L$ in time proportional to $T_1 = \max(T_1, \dots, T_L)$.

Below, we formalise the syntax of the primitive. Observe that we allow the setup to run in time proportional to $\max(T_1, \dots, T_n)$. Notice also that the resolution of batches of time-lock puzzles does not need to receive all its input at once: we rely on an algorithm to which we gradually feed the puzzles in decreasing order with respect to their time parameters. Throughout its execution, the algorithm passes along a state a.

Definition 43 (Batched homomorphic time-lock puzzle). *A batched homomorphic time-lock puzzle scheme is a tuple of PPT algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{EvalBatchSolve})$ *with the following syntax:*

$\mathsf{Setup}(1^\lambda, 1^{t_1}, \dots, 1^{t_L})$**:** *The algorithm takes as input the security parameter 1^λ, and time parameters $1^{t_1}, \dots, 1^{t_L}$ for some $L \in \mathbb{N}$. The output consists of public parameters* pp *and $q \in \mathbb{N}$.*

$\mathsf{Gen}(1^\lambda, i, \mathsf{pp}, m)$**:** *The algorithm takes as input the security parameter 1^λ, an index $i \in [L]$, the public parameters* pp *and a solution $m \in \mathbb{Z}_q$. The output is a puzzle z with time parameter $T_i := \sum_{j=i}^{L} t_j$.*

$\mathsf{Solve}(\mathsf{pp}, z)$**:** *The algorithm is deterministic and takes as input the public parameters* pp *and a puzzle z. The output is a solution $m \in \mathbb{Z}_q$.*

$\mathsf{Eval}(\mathsf{pp}, F, z_1, \dots, z_M)$**:** *The algorithm is deterministic and takes as input the public parameters* pp*, a $\mathbb{Z}_q$-linear function $F : \mathbb{Z}_q^M \to \mathbb{Z}_q$ for any $M \in \mathbb{N}$ and M puzzles $z_1, \dots, z_M$. The output is a puzzle z.*

$\mathsf{BatchSolve}(\mathsf{pp}, a, z)$**:** *The algorithm is deterministic and takes as input the public parameters* pp*, a state a and a puzzle z. The output is information h.*

Next, we define correctness for batched homomorphic time-lock puzzles. Specifically, we require three properties: if we generate a puzzle hiding a message m, its resolution returns m; if we evaluate a linear function F on puzzles with the same time parameter hiding messages $m_1, \dots, m_M$, we obtain a third puzzle hiding the message $F(m_1, \dots, m_M)$; if we batch-solve L time-lock puzzles with time parameters $T_1, \dots, T_L$ and hiding messages $m_1, \dots, m_L$, we obtain $m_1 + \cdots + m_L$.

Definition 44 (Correctness). *We say that a batched homomorphic time-lock puzzle scheme* $\mathsf{BHTLP} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{BatchSolve})$ *is correct if, for every $\lambda, t_1, \dots, t_L \in \mathbb{N}$, we have the following properties*

– *For every* $i \in [L]$*, pair* (pp, q) *in the support of* $\mathsf{Setup}(1^\lambda, 1^{t_1}, \ldots, 1^{t_L})$*,* $m \in \mathbb{N}$ *and random string* $r \in \{0,1\}^*$*, it holds that*

$$\Pr\left[z \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda, i, \mathsf{pp}, m; r) \quad : \mathsf{Solve}(\mathsf{pp}, z) \equiv m \bmod q\right] = 1.$$

– *For every* $i \in [L]$*, pair* (pp, q) *such that* (pp, q) *is in the support of* $\mathsf{Setup}(1^\lambda, 1^{t_1}, \ldots, 1^{t_L})$*, linear function* $F : \mathbb{Z}_q^M \to \mathbb{Z}_q$*, values* $m_1, \ldots, m_M \in \mathbb{N}$ *and random strings* $r_1, \ldots, r_M \in \{0,1\}^*$*, there exists a random string* $r \in \{0,1\}^*$ *such that*

$$\mathsf{Eval}(\mathsf{pp}, f, z_1, \ldots, z_M) = \mathsf{Gen}(1^\lambda, i, \mathsf{pp}, F(m_1, \ldots, m_M); r),$$

where, for every $j \in [M]$*,* $z_j = \mathsf{Gen}(1^\lambda, i, \mathsf{pp}, m_j; r_j)$*.*

– *For every* $m_1, \ldots, m_L \in \mathbb{N}$*, it holds that*

$$\Pr\begin{bmatrix} (\mathsf{pp}, q) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, 1^{t_1}, \ldots, 1^{t_L}) & \\ \forall i \in [L] : z_i \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda, i, \mathsf{pp}, m_i) & : a_L \equiv \sum_{i=1}^{L} m_i \bmod q \\ a_0 \leftarrow \bot & \\ \forall i \in [L] : a_i \leftarrow \mathsf{BatchSolve}(\mathsf{pp}, a_{i-1}, z_i) & \end{bmatrix} = 1.$$

As for randomness beacons, we define the efficiency of batched homomorphic time-lock puzzles with respect to circuit classes $\mathcal{C}_1 \subseteq \mathcal{C}_2, \mathcal{C}_3$: the generation and addition algorithms belong to the class of quickly computable circuits $\mathcal{C}_1$, whereas Solve and $\mathsf{BatchSolve}$ belong to the slower classes $\mathcal{C}_2$ and $\mathcal{C}_3$.

Definition 45 (Efficiency). *We say that a batched homomorphic time-lock puzzle scheme* $\mathsf{BHTLP} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{Eval}, \mathsf{BatchSolve})$ *is* $(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3)$*-efficient if* $\mathsf{Gen}, \mathsf{Eval} \in \mathcal{C}_1$*,* $\mathsf{Solve} \in \mathcal{C}_2$ *and* $\mathsf{BatchSolve} \in \mathcal{C}_3$*.*

Finally, we formalise security by providing a game based definition parametrised by a batch size L, a time-parameter function t and a sequence of adversarial classes $\mathcal{C} := (\mathcal{C}_i)_{i \in \mathbb{N}}$. At the beginning of the game, we provide the adversary with the public parameters of the time-lock puzzle, asking it to choose two message m_0 and m_1, an index $i \in [L]$ and the code of a circuit $C \in \mathcal{C}_i$. We feed C with a time-lock puzzle with time parameter $T_i := \sum_{j=i}^{L} t(j)$. The adversary wins if C is capable of distinguishing whether the puzzle hides m_0 or m_1. In other words, the game ensures that the message in a puzzle with time parameter T_i is inaccessible to all adversaries in $\mathcal{C}_i$.

Definition 46 (Security). *Let* $\mathcal{C}_i = (\mathcal{C}_{i,\lambda})_{\lambda \in \mathbb{N}}$ *be a product circuit class for every* $i \in \mathbb{N}$*. Let* $\mathcal{C} := (\mathcal{C}_i)_{i \in \mathbb{N}}$*. Let* $t : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *be a function. Finally, let* $L(\lambda)$ *be a polynomial function in the security parameter. Consider a batched homomorphic time-lock puzzle scheme* $\mathsf{BHTLP} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{Eval}, \mathsf{BatchSolve})$*. We say that an adversary* $\mathcal{A}$ *is* $(\mathsf{BHTLP}, \mathcal{C}, L, t)$*-consistent if, for every sufficiently large* $\lambda \in \mathbb{N}$*,*

$$\Pr\begin{bmatrix} (\mathsf{pp}, q) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, 1^{t(1,\lambda)}, \ldots, 1^{t(L(\lambda),\lambda)}) & : \mathcal{A}' \in \mathcal{C}_{i,\lambda} \\ (m_0, m_1, i, \mathcal{A}') \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, \mathsf{pp}, q) & \end{bmatrix} = 1.$$

We say that BHTLP *is* $(\mathcal{C}, L, t)$*-secure if, for every* $(\mathsf{BHTLP}, \mathcal{C}, L, t)$*-consistent PPT adversary* $\mathcal{A}$*, it holds that*

$$\mathsf{Adv}_{\mathsf{BHTLP}}^{\mathsf{BHTLP\text{-}SEC},L,t}(\mathcal{A}) := \left|\Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{BHTLP}}^{\mathsf{BHTLP\text{-}SEC},L,t}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$$

where experiment $\mathsf{Exp}_{\mathcal{A},\mathsf{BHTLP}}^{\mathsf{BHTLP\text{-}SEC},L,t}$ *is specified in Figure 14.*

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underline{\mathsf{Exp}^{\mathsf{BHTLP\text{-}SEC},L,t}_{\mathcal{A},\mathsf{BHTLP}}(1^\lambda)}} \\
1: & (\mathsf{pp}, q) \leftarrow\$ \mathsf{Setup}(1^\lambda, 1^{t(1,\lambda)}, \ldots, 1^{t(L(\lambda),\lambda)}) \\
2: & (m_0, m_1, i, \mathcal{A}') \leftarrow\$ \mathcal{A}(1^\lambda, \mathsf{pp}, q) \\
3: & b \leftarrow\$ \{0,1\} \\
4: & z \leftarrow\$ \mathsf{Gen}(1^\lambda, i, \mathsf{pp}, m_b) \\
5: & b' \leftarrow\$ \mathcal{A}'(z) \\
6: & \textbf{return } b = b'
\end{array}
$$

Fig. 14. Batched homomorphic time-lock puzzle security game BHTLP-SEC.

Observe that the setup algorithm of batched homomorphic time-lock puzzles outputs also the modulus q describing the group $\mathbb{Z}_q$ over which the primitive is homomorphic. Now, in some instantiations of the primitive, q is an RSA modulus. It is not hard to imagine applications of this particular construction in contexts in which it is important that the factorisation of q remains secret. How can we be sure that the setup of the time-lock puzzle, which generated q in the first place, does not leak such factorisation or any other information that can compromise the security of the whole system?

The below definition tries to formalise the needed property. Let D be a distribution over the moduli q. We say that the batched homomorphic time-lock puzzle makes black-box group usage with respect to D if the output of Setup can be efficiently simulated given a modulus q sampled according to D.

Definition 47 (Black-box group usage). *Let* D *be a distribution. We say that a batched homomorphic time-lock puzzle* $\mathsf{BHTLP} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{Eval}, \mathsf{BatchSolve})$ *makes black box use of the group with respect to* D *if there exists a PPT simulator* Sim *such that, for every adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1) \in \mathsf{poly}(\lambda) \times \mathsf{poly}(\lambda)$*, it holds that*

$$
\mathsf{Adv}^{\mathsf{BBG},\mathsf{D}}_{\mathsf{BHTLP},\mathsf{Sim}}(\mathcal{A}) := \left| \Pr[\mathsf{Exp}^{\mathsf{BBG},\mathsf{D}}_{\mathcal{A},\mathsf{BHTLP},\mathsf{Sim}}(1^\lambda) = 1] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda)
$$

where experiment $\mathsf{Exp}^{\mathsf{BBG},\mathsf{D}}_{\mathcal{A},\mathsf{BHTLP},\mathsf{Sim}}$ *is specified in Figure 15.*

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underline{\mathsf{Exp}^{\mathsf{BBG},\mathsf{D}}_{\mathcal{A},\mathsf{BHTLP},\mathsf{Sim}}(1^\lambda)}} \\
1: & b \leftarrow\$ \{0,1\} \\
2: & (t_1, \ldots, t_L, \psi) \leftarrow\$ \mathcal{A}_0(1^\lambda) \\
3: & (\mathsf{pp}_0, q_0) \leftarrow\$ \mathsf{Setup}(1^\lambda, 1^{t_1}, \ldots, 1^{t_L}) \\
4: & (q_1, \mathsf{aux}) \leftarrow\$ \mathsf{D}(1^\lambda) \\
5: & \mathsf{pp}_1 \leftarrow\$ \mathsf{Sim}(1^\lambda, 1^{t_1}, \ldots, 1^{t_L}, q_1, \mathsf{aux}) \\
6: & b' \leftarrow\$ \mathcal{A}_1(\psi, \mathsf{pp}_b, q_b) \\
7: & \textbf{return } b = b'
\end{array}
$$

Fig. 15. Black-box group usage game BBG.

Building Batched Homomorphic Time-Lock Puzzles. We show that the Paillier-based time-lock puzzle of [MT19] is not only additively homomorphic but also a batched homomorphic time-lock puzzle. Furthermore, the construction can be generalised to different settings: inspired by [ADOS22, ADIN24, ARS24], we provide an algebraic framework over which it is possible to build batched homomorphic time-lock puzzles following the blueprint of [MT19]. We call such framework the *DLOG framework*. Among the various instantiations, we not only find the Paillier group but also class groups. In this way, we manage to build batched homomorphic time-lock puzzles with a transparent setup over any group $\mathbb{Z}_q$ where q is a prime.

The DLOG framework consists of a multiplicative, abelian group G which can decomposed as the direct product of subgroups F and H. The first subgroup, F, is cyclic, generated by an element f of known order q. Furthermore, there exists an efficient algorithm for the computation of discrete logarithms over F. The subgroup H, on the other hand, may not be cyclic and has unknown order. Furthermore, discrete logarithms are hard to compute over H. The group G is also equipped with an efficiently samplable distribution D.

Definition 48 (DLOG framework). *The DLOG framework consist of a triple of PPT algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{D}, \mathsf{DLOG})$ *with the following syntax:*

$\mathsf{Setup}(1^\lambda)$: *The algorithm takes as input the security parameter* 1^λ *and outputs a positive integer* q.

$\mathsf{Gen}(1^\lambda, q)$**:** *The algorithm takes as input the security parameter* 1^λ *and an integer* q. *The output is groups* G, H, F *such that* $G = F \times H$, *an element* f *such that* $F = \langle f \rangle$ *and* $q = \mathsf{ord}(f)$, *an integer* ℓ *and auxiliary information* $\mathtt{aux}$.

$\mathsf{D}(1^\lambda, \mathtt{aux})$**:** *The algorithm takes as input the security parameter* 1^λ *and auxiliary information* $\mathtt{aux}$. *The output is a group element* g *and information* ρ.

$\mathsf{DLOG}(h, \mathtt{aux})$**:** *The algorithm is deterministic and takes as input* $h \in F$ *and auxiliary information* $\mathtt{aux}$. *The output is a value* $m \in \mathbb{Z}_q$.

We also require that, for every $\lambda, m \in \mathbb{N}$, *it holds that*

$$\Pr\left[\begin{array}{l} q \leftarrow\$\ \mathsf{Setup}(1^\lambda) \\ (G, H, F, f, \ell, \mathtt{aux}) \leftarrow\$\ \mathsf{Gen}(1^\lambda, q) \\ h \leftarrow f^m \end{array} : \mathsf{DLOG}(h, \mathtt{aux}) = m \bmod q\right] = 1.$$

We highlight two facts: the generation algorithm outputs a positive integer ℓ. Such value will be used to sample random exponents $r \leftarrow\$ [\ell]$ so that g^r, where $(g, \rho) \leftarrow\$ \mathsf{D}(1^\lambda, \mathtt{aux})$, becomes unpredictable. Notice that g^r is not uniformly random as, in general, $G \neq \langle g \rangle$ and $\mathsf{ord}(g)$ is unknown and different from ℓ. The other observation concerns the information ρ output by $\mathsf{D}(1^\lambda, \mathtt{aux})$ along with the group element g. In many instantiations of the framework, ρ represents the randomness used to sample g. Notice that sometimes (for instance, this is the case for class groups), the randomness ρ is not simulatable from g. So, in order to make the sampling procedure $\mathsf{D}(1^\lambda, \mathtt{aux})$ transparent, ρ needs to be provided as part of the output.

We define the efficiency of the DLOG framework in terms of the depth of the circuit computing the group multiplication and the size of the parameters q and ℓ.

Definition 49 (Efficiency in the DLOG framework). *Let* $f_0(\lambda), f_1(\lambda)$ *be functions of the security parameter. We say that the DLOG framework* $\mathsf{DFW} := (\mathsf{Setup}, \mathsf{Gen}, \mathsf{D}, \mathsf{DLOG})$ *is* (f_0, f_1)-

efficient if, for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} q \leftarrow\$ \mathsf{Setup}(1^\lambda) \\ (G,H,F,f,\ell,\mathsf{aux}) \leftarrow\$ \mathsf{Gen}(1^\lambda,q) \end{array} : q,\ell \le f_0(\lambda)\right] = 1$$

and the computational depth of multiplications over G *is upper bounded by* $f_1(\lambda)$.

Finally, we describe the assumption we require in order to build batched homomorphic time-lock puzzles over the framework. We call the assumption the $(\mathcal{C},\bar{t})$-*time assumption*, where $\mathcal{C} = (\mathcal{C}_i)_{i\in\mathbb{N}}$ denotes a sequence of adversarial circuit classes and $\bar{t}$ is function of the security parameter. The assumption states that, for every polynomial function $t = \Omega(\bar{t})$ and adversary in $\mathcal{C}_t$, the tuples $(g,\rho,g^r,g^{r\cdot 2^t})$ and $(g,\rho,g^r,f^s\cdot g^{r\cdot 2^t})$, where $(g,\rho)\leftarrow\$ \mathsf{D}(1^\lambda,\mathsf{aux})$, $r\leftarrow\$ [\ell]$ and $s\leftarrow\$ \mathbb{Z}_q$, are indistinguishable.

Definition 50 **($(\mathcal{C},\bar{t})$-time assumption).** *Let* $\mathsf{DFW} := (\mathsf{Setup},\mathsf{Gen},\mathsf{D},\mathsf{DLOG})$ *be a DLOG framework. Let* $\mathcal{C}_i = (\mathcal{C}_{i,\lambda})_{\lambda\in\mathbb{N}}$ *be a product circuit class for every* $i\in\mathbb{N}$, *and let* $\mathcal{C} := (\mathcal{C}_i)_{i\in\mathbb{N}}$. *Let* $\bar{t}(\lambda)$ *be a function of the security parameter. We say that an adversary* $\mathcal{A}$ *is* $(\mathsf{DFW},\mathcal{C})$*-consistent if, for every* $t,\lambda\in\mathbb{N}$, *it holds that*

$$\Pr\left[\begin{array}{l} q \leftarrow\$ \mathsf{Setup}(1^\lambda) \\ (G,H,F,f,\ell,\mathsf{aux}) \leftarrow\$ \mathsf{Gen}(1^\lambda,q) \\ (g,\rho)\leftarrow\$ \mathsf{D}(1^\lambda,\mathsf{aux}) \\ \mathcal{A}' \leftarrow\$ \mathcal{A}_0(1^\lambda,1^t,G,H,F,f,q,\ell,\mathsf{aux},g,\rho) \end{array} : \mathcal{A}'\in\mathcal{C}_{t,\lambda}\right] = 1.$$

We say that the $(\mathcal{C},\bar{t})$*-time assumption holds in the DLOG framework* DFW *if, for every* $(\mathsf{DFW},\mathcal{C})$*-consistent PPT adversary* $\mathcal{A}$ *and polynomial function* $t = \Omega(\bar{t}(\lambda))$, *it holds that*

$$\mathsf{Adv}^{\mathsf{TIME},t}_{\mathsf{DFW}}(\mathcal{A}) := \left|\Pr[\mathsf{Exp}^{\mathsf{TIME},t}_{\mathcal{A},\mathsf{DFW}}(1^\lambda)=1] - \frac{1}{2}\right| \le \mathsf{negl}(\lambda)$$

where experiment $\mathsf{Exp}^{\mathsf{TIME},t}_{\mathcal{A},\mathsf{DFW}}$ *is specified in Figure 16.*

We are finally ready to present how to build a batched homomorphic time-lock puzzle over the DLOG framework. The idea is the same as in [MT19]: in the setup phase, we generate the parameters for the group $G = F\times H$ and we sample an element $g\in G$. A puzzle with time parameter T hiding a message m consists of a pair $(x,y) := (g^r, f^m\cdot g^{r\cdot 2^T})$ where $r\leftarrow\$ [\ell]$. Notice that $g_T := g^{2^T}$ can be precomputed during the setup phase, so the generation of the puzzle is fast: we require only $O(\log\ell)$ group multiplications instead of $O(T)$ as $f^m\cdot g^{r\cdot 2^T} = f^m\cdot g_T^r$. To retrieve m, it is sufficient to compute $\mathsf{DLOG}(y/x^{2^T})$ which requires $O(T)$ sequential group multiplications. The idea is that under the time assumption of Def. 50, if T is sufficiently large, the message m remains hidden from any computationally bounded adversary of depth $o(T)$.

It is easy to see that the scheme is linearly homomorphic: given $\alpha,\beta\in\mathbb{Z}_q$, two puzzles $(g^{r_1}, f^{m_1}\cdot g^{r_1\cdot 2^T})$ and $(g^{r_2}, f^{m_2}\cdot g^{r_2\cdot 2^T})$ can be quickly combined into another puzzle

$$(g^{\alpha\cdot r_1+\beta\cdot r_2}, f^{\alpha\cdot m_1+\beta\cdot m_2}\cdot g^{(\alpha\cdot r_1+\beta\cdot r_2)\cdot 2^T})$$

by just raising the two pairs by α and β component-wise and then multiplying the results together. As for the batched resolutions of puzzles $(x_1,y_1) := (g^{r_1}, f^{m_1}\cdot g^{r_1\cdot 2^{T_1}}),\ldots,(x_L,y_L) := (g^{r_L}, f^{m_L}\cdot$

$\mathsf{Exp}^{\mathsf{TIME},t}_{\mathcal{A},\mathsf{DFW}}(1^\lambda)$

1 : $q \leftarrow\$ \mathsf{Setup}(1^\lambda)$
2 : $(G, H, F, f, \ell, \mathtt{aux}) \leftarrow\$ \mathsf{Gen}(1^\lambda, q)$
3 : $(g, \rho) \leftarrow\$ \mathsf{D}(1^\lambda, \mathtt{aux})$
4 : $\mathcal{A}' \leftarrow\$ \mathcal{A}_0(1^\lambda, 1^{t(\lambda)}, G, H, F, f, q, \ell, \mathtt{aux}, g, \rho)$
5 : $b \leftarrow\$ \{0, 1\}$
6 : $r \leftarrow\$ [\ell]$
7 : $s \leftarrow\$ \mathbb{Z}_q$
8 : $x \leftarrow g^r$
9 : $y_0 \leftarrow x^{2^{t(\lambda)}}$
10 : $y_1 \leftarrow f^s \cdot x^{2^{t(\lambda)}}$
11 : $b' \leftarrow\$ \mathcal{A}'(x, y_b)$
12 : **return** $b = b'$

Fig. 16. Time assumption experiment for the DLOG framework.

$g^{r_L \cdot 2^{T_L}})$, where $T_1 \geq \cdots \geq T_L$, the operation can be performed in two steps: on one hand, we compute

$$y = y_1 \cdot \cdots \cdot y_L = f^{m_1 + \ldots m_L} \cdot g^{r_1 \cdot 2^{T_1} + \cdots + r_L \cdot 2^{T_L}},$$

on the other, we compute

$$x = \left(\left(\left(\left(x_1^{2^{T_1 - T_2}} \cdot x_2 \right)^{2^{T_2 - T_3}} \cdot x_3 \right)^{2^{T_3 - T_4}} \cdot \ldots \right)^{2^{T_{L-1} - T_L}} \cdot x_L \right)^{2^{T_L}} = g^{r_1 \cdot 2^{T_1} + \cdots + r_L \cdot 2^{T_L}}.$$

By computing $\mathsf{DLOG}(y/x)$, we obtain $m_1 + \cdots + m_L$ at the cost of $O(T_1)$ group operations.

Theorem 10. *Let* $\mathsf{DFW} := (\mathsf{Setup}, \mathsf{Gen}, \mathsf{D}, \mathsf{DLOG})$ *be a DLOG framework and let* $d(\lambda), f_0(\lambda), f_1(\lambda), \bar{t}(\lambda), L(\lambda)$ *be polynomial functions of the security parameter. Define* $\mathcal{C}_i$ *as* $\overline{\mathcal{C}}_{i \cdot d}$ *and* $\mathcal{C}'_i$ *as* $\overline{\mathcal{C}}_{i \cdot d + f_1 + 1}$*. Let* $\mathcal{C} := (\mathcal{C}_i)_{i \in \mathbb{N}}$ *and* $\mathcal{C}' := (\mathcal{C}'_i)_{i \in \mathbb{N}}$*. Define the function* $t(i, \lambda) = \bar{t}(\lambda)$*.*

If the $(\mathcal{C}', \bar{t})$*-time assumption holds for a* (f_0, f_1)*-efficient DLOG framework* DFW*, the construction in Fig.17 is a batched homomorphic time-lock puzzle satisfying correctness,* $(\mathcal{C}_{f_0 \cdot f_1}, \mathcal{C}_{L \cdot \bar{t} \cdot f_1}, \mathcal{C}_{\bar{t} \cdot f_1})$*-efficiency and* $(\mathcal{C}, L, t)$*-security. Moreover, the scheme makes black-box group usage with respect to* $\mathsf{DFW.Setup}$*.*

Proof. Our construction clearly makes black-box group usage with respect to $\mathsf{DFW.Setup}$. We start by proving the correctness of the batched homomorphic time-lock puzzle. We observe that $\mathsf{Gen}(1^\lambda, i, \mathsf{pp}, m; r)$ outputs a triple $z := (i, x, y)$ where $x = g^r$ and $y = f^m \cdot g^{r \cdot 2^{(L(\lambda) - i + 1) \cdot \bar{t}(\lambda)}}$. Therefore, during the execution of $\mathsf{Solve}(\mathsf{pp}, z)$, we obtain that

$$h = y \cdot x^{-2^{(L(\lambda) - i + 1) \cdot \bar{t}(\lambda)}} = f^m.$$

We conclude that the first correctness property is satisfied.

Batched Homomorphic Time-Lock Puzzle

$\mathsf{Setup}(1^\lambda, 1^{t_1}, \ldots, 1^{t_L})$:

1. $q \leftarrow\$ \mathsf{DFW.Setup}(1^\lambda)$
2. $(G, H, F, f, \ell, \mathsf{aux}) \leftarrow\$ \mathsf{DFW.Gen}(1^\lambda, q)$
3. $(g, \rho) \leftarrow\$ \mathsf{DFW.D}(1^\lambda, \mathsf{aux})$
4. $\forall i \in [L] : g_i \leftarrow g^{2^{\sum_{j \geq i} t_j}}$
5. $\mathsf{pp} \leftarrow (f, \ell, \mathsf{aux}, g, \rho, (g_1, 1^{t_1}), \ldots, (g_L, 1^{t_L}))$
6. Output pp and q.

$\mathsf{Gen}(1^\lambda, i, \mathsf{pp} = (f, \ell, \mathsf{aux}, g, \rho, (g_1, 1^{t_1}), \ldots, (g_L, 1^{t_L})), m)$:

1. $r \leftarrow\$ [\ell]$
2. $x \leftarrow g^r$
3. $y \leftarrow f^m \cdot g_i^r$
4. Output $z := (i, x, y)$

$\mathsf{Solve}(\mathsf{pp} = (f, \ell, \mathsf{aux}, g, \rho, (g_1, 1^{t_1}), \ldots, (g_L, 1^{t_L})), z = (i, x, y))$:

1. $h \leftarrow y \cdot x^{-2^{\sum_{j \geq i} t_j}}$
2. Output $\mathsf{DFW.DLOG}(h, \mathsf{aux})$

$\mathsf{Eval}(\mathsf{pp} = (f, \ell, \mathsf{aux}, g, \rho, (g_1, 1^{t_1}), \ldots, (g_L, 1^{t_L})), F = (\alpha_1, \ldots, \alpha_M), z_1 = (i, x_1, y_1), \ldots, z_M = (i, x_M, y_M))$:

1. $x \leftarrow x_1^{\alpha_1} \cdots x_M^{\alpha_M}$
2. $y \leftarrow y_1^{\alpha_1} \cdots y_M^{\alpha_M}$
3. Output $z := (i, x, y)$

$\mathsf{BatchSolve}(\mathsf{pp} = (f, \ell, \mathsf{aux}, g, \rho, (g_1, 1^{t_1}), \ldots, (g_L, 1^{t_L})), a = (a_1, a_2), z = (i, x, y))$:

1. If $a = \perp$, $(a_1, a_2) \leftarrow (1, 1)$
2. $h_2 \leftarrow a_2 \cdot y$
3. $h_1 \leftarrow (a_1 \cdot x)^{2^{t_i}}$
4. If $i = L$, output $\mathsf{DFW.DLOG}(h_1 \cdot h_2^{-1}, \mathsf{aux})$
5. Otherwise, output $h := (h_1, h_2)$

Fig. 17. Batched homomorphic time-lock puzzle

Moving on to the second correctness property, we observe that, if $F(m_1, \dots, m_M) = (\alpha_1 \cdot m_1 + \dots + \alpha_M \cdot m_M,\ z_1 = (i, x_1, y_1), \dots, z_M = (i, x_M, y_M)$, where

$$x_1 = g^{r_1} \quad \dots \quad x_M = g^{r_M} \qquad y_1 = f^{m_1} \cdot g^{r_1 \cdot 2^{(L(\lambda)-i+1)\cdot \bar{t}(\lambda)}} \quad \dots \quad y_M = f^{m_M} \cdot g^{r_M \cdot 2^{(L(\lambda)-i+1)\cdot \bar{t}(\lambda)}},$$

the algorithm $\mathsf{Eval}(\mathsf{pp}, F, z_1, \dots, z_M)$ outputs a tuple (i, x, y) where

$$x = g^{\alpha_1 \cdot r_1 + \dots + \alpha_M \cdot r_M} \quad \text{and} \quad y = f^{\alpha_1 \cdot m_1 + \dots + \alpha_M \cdot m_M} \cdot g^{(\alpha_1 \cdot r_1 + \dots + \alpha_M \cdot r_M)\cdot 2^{(L(\lambda)-i+1)\cdot \bar{t}(\lambda)}}.$$

Therefore, $z = \mathsf{Gen}(1^\lambda, i, \mathsf{pp}, F(m_1, \dots, m_M); r)$ where $r = \alpha_1 \cdot r_1 + \dots + \alpha_M \cdot r_M$.

Finally we prove the last correctness property. For every $i \in [L(\lambda)]$, let $z_i := (i, x_i, y_i)$ be the output of $\mathsf{Gen}(1^\lambda, i, \mathsf{pp}, m_i)$. Observe that, for every $i \in [L(\lambda)]$, there exists a r_i such that $x_i = g^{r_i}$ and $y_i = f^{m_i} \cdot g^{2^{(L(\lambda)-i+1)\cdot \bar{t}(\lambda)}}$. Now, recursively define $a_i = (a_{i,1}, a_{i,2})$ as the output of $\mathsf{BatchSolve}(\mathsf{pp}, a_{i-1}, z_i)$ where $a_0 := \perp$. We claim that, for every $i \in [L(\lambda)]$,

$$a_{i,1} = \prod_{j=1}^{i} g^{r_j \cdot 2^{(i-j+1)\cdot \bar{t}(\lambda)}} \qquad \text{and} \qquad a_{i,2} = f^{\sum_{j=1}^{i} m_j} \cdot \prod_{j=1}^{i} g^{r_j \cdot 2^{(L(\lambda)-j+1)\cdot \bar{t}(\lambda)}} \tag{1}$$

It is easy to verify the claim for $i = 1$. Now, by induction suppose that the claim holds for $i - 1$. We prove it for i. It is easy to see that $a_{i,2}$ satisfies the second part of (1). As for $a_{i,1}$, we observe that

$$a_{i,1} = (a_{i-1,1} \cdot x_i)^{2^{\bar{t}(\lambda)}} = \prod_{j=1}^{i-1} g^{r_j \cdot 2^{(i-j+1)\cdot \bar{t}(\lambda)}} \cdot x_i^{2^{\bar{t}(\lambda)}} = \prod_{j=1}^{i-1} g^{r_j \cdot 2^{(i-j+1)\cdot \bar{t}(\lambda)}} \cdot g^{r_i \cdot 2^{\bar{t}(\lambda)}} = \prod_{j=1}^{i} g^{r_j \cdot 2^{(i-j+1)\cdot \bar{t}(\lambda)}}$$

We conclude by observing that $a_{L(\lambda),2} \cdot a_{L(\lambda),1}^{-1} = f^{\sum_{i=1}^{L(\lambda)} m_i}$ and $\mathsf{ord}(f) = q$.

Since efficiency can be easily verified (we just need to apply exponentiation by squaring), we focus on security. Suppose by contradiction that there exists a $(\mathsf{BHTLP}, \mathcal{C}, L, t)$-consistent PPT adversary $\mathcal{A}$ that wins the $(\mathcal{C}, L, t)$-security game with non-negligible advantage. We show how to build a $(\mathsf{DFW}, \mathcal{C}')$-consistent adversary $\mathcal{B}$ and a polynomial function $\tau = \Omega(\bar{t}(\lambda))$ such that $\mathsf{Adv}_{\mathsf{DFW}}^{\mathsf{TIME},\tau}(\mathcal{B})$ is non-negligible.

We start by building τ. For every $\lambda \in \mathbb{N}$, we define $\tau(\lambda)$ as $j \cdot \bar{t}(\lambda)$ where j is the value in $[L(\lambda)]$ that maximises

$$\delta(\lambda, j) := \left| \Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{BHTLP}}^{\mathsf{BHTLP\text{-}SEC},L,t}(1^\lambda) = 1, i = j] - \frac{1}{2} \cdot \Pr[i = j] \right|.$$

Above, i denotes the index output by $\mathcal{A}$ in $\mathsf{Exp}_{\mathcal{A},\mathsf{BHTLP}}^{\mathsf{BHTLP\text{-}SEC},L,t}(1^\lambda)$. We observe that $\delta(\lambda, \tau(\lambda))$ is a non-negligible function in λ. Indeed, $L(\lambda)$ is polynomial and

$$\mathsf{Adv}_{\mathsf{BHTLP}}^{\mathsf{BHTLP\text{-}SEC},L,t}(\mathcal{A}) \leq \sum_{j=1}^{L(\lambda)} \delta(\lambda, j) \leq L(\lambda) \cdot \delta(\lambda, \tau(\lambda)).$$

We also observe that τ is polynomial as it is upper bounded by $L(\lambda) \cdot \bar{t}(\lambda)$. Furthermore, τ is $\Omega(\bar{t}(\lambda))$.

Next, we build $\mathcal{B}$. Such adversary receives $(G, H, F, f, q, \ell, \mathsf{aux}, g, \rho)$ from the challenger of $\mathsf{Exp}_{\mathcal{B},\mathsf{DFW}}^{\mathsf{TIME},\tau}(1^\lambda)$. Then, it performs the following operations:

1. $\forall i \in [L(\lambda)] : g_i \leftarrow g^{2^{(L(\lambda)-i+1)\cdot \bar{t}(\lambda)}}$
2. $\mathtt{pp} \leftarrow (f, \ell, \mathtt{aux}, g, \rho, (g_1, 1^{t_1}), \ldots, (g_L, 1^{t_L}))$
3. $(m_0, m_1, i, \mathcal{A}') \leftarrow\$ \mathcal{A}(1^\lambda, \mathtt{pp}, q)$

Finally, it outputs the adversary $\mathcal{B}'$ defined as follows. If $i \neq \tau(\lambda)$, on input a pair of group elements (x, y), $\mathcal{B}'$ outputs a uniformly random bit. Otherwise, it performs the following operations:

1. $b \leftarrow\$ \{0, 1\}$
2. $y_b \leftarrow f^{m_b} \cdot y$
3. $b' \leftarrow \mathcal{A}'(i, x, y_b)$
4. If $b = b'$, $\mathcal{B}'$ outputs 0, otherwise, it output 1.

We notice that $\mathcal{B}$ is $(\mathsf{DFW}, \mathcal{C}')$-consistent. Indeed, the first two operations performed by $\mathcal{B}'$ require depth $f_1(\lambda)$ (f^{m_b} can be precomputed by $\mathcal{B}$, so we just need to perform a multiplication). The third step requires depth at most $i \cdot d(\lambda)$ as $\mathcal{A}' \in \mathcal{C}_{i,\lambda}$. Finally, the last step has depth 1.

Concerning the advantage of $\mathcal{B}$, we observe that, when $y = f^s \cdot x^{2^{\bar{t}(\lambda)}}$ and $i = \tau(\lambda)$, the probability that $b = b'$ is exactly $1/2$ (y_b is independent of b). So, the probability that $\mathcal{B}$ wins the game if $y = f^s \cdot x^{2^{\bar{t}(\lambda)}}$ is $1/2$. If instead $y = x^{2^{\bar{t}(\lambda)}}$, the probability that $\mathcal{B}$ wins the game is $1/2$ when $i \neq \tau(\lambda)$, and $1/2 + \delta(\lambda, \tau(\lambda))/\Pr[i = \tau(\lambda)]$ or $1/2 - \delta(\lambda, \tau(\lambda))/\Pr[i = \tau(\lambda)]$ when $i = \tau(\lambda)$. We conclude that $\mathsf{Adv}^{\mathsf{TIME},\tau}_{\mathsf{DFW}}(\mathcal{B}) = \delta(\lambda, \tau(\lambda))/2$. Since this is a non-negligible amount, we reached a contradiction. □

Randomness Beacon from Batched Homomorphic Time-Lock Puzzles. We finally show how it is possible to improve the randomness beacon described at the beginning of this section thanks to batched homomorphic time-lock puzzles.

Modified oracle queries and homomorphic puzzles. We start by modifying the construction in Fig. 13, changing the oracle queries producing the beacon's output: instead of querying the concatenation of the solutions of the puzzles in the last T blocks, we instead query their addition over $\mathbb{Z}_q$. In order to avoid replay attacks, if there exist indexes $j < i$ among the last T published blocks such that $B_i = B_j$, we ignore B_i. Furthermore, we rely on a simulation-extractable NIZK to make the locks non-malleable. We observe that the linearly homomorphic properties of the puzzles immediately give us a way to retrieve the each beacon output at the costs of a single puzzle resolution: we just homomorphically combine all puzzles in honestly generated blocks (i.e. those where the NIZK verifies and that are not duplicates of previous blocks) and we solve the resulting object.

Security. Notice that the adversary is not able to bias the beacon's outputs as long as there is at least one honest block every T blocks and the time parameters of the puzzles are chosen sufficiently large: in order to bias the output, the adversary would need to solve the puzzle published in the last block generated by an honest party. The operation is however slow enough that, by the time the adversary finishes, the honest parties have already managed to publish other $T - 1$ blocks. In other words, by the time the adversary solves the puzzle, all beacon's outputs depending on its solution have already been fixed.

Efficient computation of proofs of correct evaluation. We have seen how the homomorphic properties of the puzzles and the modified oracle queries allowed us to obtain every beacon's output at the cost of a single puzzle resolution. However, what about the quickly verifiable proof of correct evaluation?

If we use generic commitments as in the construction in Fig. 13, the proof would need to include T openings containing the solutions of the T puzzles in the last T blocks. In other words, in order to obtain them proof of correct evaluation, we would still need to solve T puzzles. We solve this issue by using a commitment scheme that is additively homomorphic over $\mathbb{Z}_q$ with respect to both the the values and the openings. In other words, two commitments c_1 and c_2 with openings (s_1, v_1) and (s_2, v_2) can be quickly combined into a commitment c_3 with opening $(s_1+s_2 \bmod q, v_1+v_2 \bmod q)$. Notice that if q is a prime (this is for instance the case if we instantiate the puzzles using class groups), we can rely on Pedersen commitments. In other words, by solving the addition of the puzzles in the last T blocks, we obtain the value and the relative opening of the sum of the commitments in the last T blocks: in conclusion, we obtain each beacon's output and a proof of correct evaluation at the cost of a single puzzle resolution.

Decreasing the delay. In the construction in Fig. 13, the honest parties discover the beacon outputs with a delay of at least T blocks. We wonder: is it possible to decrease this delay? We show that the answer is yes. Our idea is the following: in each block B_i, we now include T time-lock puzzles hiding independently sampled secrets (the puzzles are juxtaposed with equally as many commitments to the secrets). The solution to the j-th puzzle in B_i will only contribute to the derivation of the j-th beacon output after the publication of B_i (i.e. r_{i+j-1}). We therefore choose the time parameter of the j-th puzzle so that the adversary cannot find the solution before B_{i+j-1} is published. Indeed, once B_{i+j-1} is published, the adversary cannot bias the beacon output r_{i+j-1} anymore as its value is fixed.

To summarise, the T puzzles in B_i will be generated with increasing time parameters. Furthermore, every beacon's output is obtained by solving for the addition of the solutions of T time-lock puzzles with different time parameters. Notice that since we rely on a batched homomorphic time-lock puzzle, each beacon's output and the corresponding correctness proof have the same computational complexity as the resolution of a single time-lock puzzle. Moreover, the delay with which the honest parties learn the outputs is smaller: we can now choose the time parameters of the puzzles so that the honest parties obtain the sum of the solutions necessary for r_i soon after the block B_i is published. Notice, however, that we still need to pay attention that the time parameters are large enough to prevent the adversary from solving any of the T puzzles before B_i is published.

Theorem 11. *Let $T(\lambda), t(\lambda), f(\lambda), f_0(\lambda), f_1(\lambda), f_2(\lambda), f_3(\lambda), f_4(\lambda)$ be polynomial functions in the security parameter. Suppose that $f_1(\lambda) = \omega(\log \lambda)$ and $f_3(\lambda) = \Omega(f_1(\lambda))$. For every $i \in \mathbb{N}$, let $\mathcal{C}_i := \mathcal{C}_{i \cdot f_0}$ and define $\mathcal{C} := (\mathcal{C}_i)_{i \in \mathbb{N}}$.*

Let $\mathsf{BHTLP} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{Eval}, \mathsf{BatchSolve})$ be a $(\mathcal{C}_{f_1}, \mathcal{C}_{f_2}, \mathcal{C}_{f_3})$-efficient, $(\mathcal{C}, T, t)$-secure batched homomorphic time-lock puzzle. Let $\mathsf{C} = (\mathsf{D}, \mathsf{Setup}, \mathsf{Commit}, \mathsf{Check})$ be a hiding and binding, strongly homomorphic commitment. Suppose that BHTLP makes black-box group usage with respect to $\mathsf{C.D}$. Suppose also that the modulo q generated by $\mathsf{C.D}$ is $\omega(\log \lambda)$-bit long. Let NIZK be a simulation-extractable NIZK for the relation

$$\mathcal{R} := \left\{ \begin{array}{l} (\lambda, (z_j, z'_j, c_j)_{j \in [T]}, \mathsf{pp}', \omega), (s_j, v_j, u_{j,0}, u_{j,1}, u'_{j,1})_{j \in [T]} \\ \quad \left| \begin{array}{l} \forall j \in [T(\lambda)] : (c_j, v_j) = \mathsf{C.Commit}(\omega, s_j; u_{j,0}) \\ \forall j \in [T(\lambda)] : z_j = \mathsf{BHTLP.Gen}(1^\lambda, j, \mathsf{pp}', s_j; u_{j,1}) \\ \forall j \in [T(\lambda)] : z'_j = \mathsf{BHTLP.Gen}(1^\lambda, j, \mathsf{pp}', v_j; u'_{j,1}) \end{array} \right. \end{array} \right\}$$

Randomness Beacon from Homomorphic Time-Lock Puzzles

$\mathsf{Setup}(1^\lambda)$:
1. $\sigma \leftarrow\!\!\$\ \mathsf{NIZK.Setup}(1^\lambda)$
2. $(\mathsf{pp}', q) \leftarrow\!\!\$\ \mathsf{BHTLP.Setup}(1^\lambda, 1^{t(\lambda)\cdot T(\lambda)})$
3. $\omega \leftarrow\!\!\$\ \mathsf{C.Setup}(1^\lambda, q)$
4. Output $\mathsf{pp} := (\mathsf{pp}', \sigma, \omega)$.

$\mathsf{Gen}(\mathsf{pp} = (\mathsf{pp}', \sigma, \omega))$:
1. $\forall j \in [T] : s_j \leftarrow\!\!\$\ \mathbb{Z}_q$
2. $\forall j \in [T] : (c_j, v_j) \leftarrow\!\!\$\ \mathsf{C.Commit}(\omega, s_j; u_{j,0})$
3. $\forall j \in [T] : z_j \leftarrow\!\!\$\ \mathsf{BHTLP.Gen}(1^\lambda, j, \mathsf{pp}', s_j; u_{j,1})$
4. $\forall j \in [T] : z'_j \leftarrow\!\!\$\ \mathsf{BHTLP.Gen}(1^\lambda, j, \mathsf{pp}', v_j; u'_{j,1})$
5. $\pi \leftarrow\!\!\$\ \mathsf{NIZK.Prove}(\sigma, (\lambda, (z_j, z'_j, c_j)_{j\in[T]}, \mathsf{pp}', \omega), (s_j, v_j, u_{j,0}, u_{j,1}, u'_{j,1})_{j\in[T]})$
6. Output $B := ((c_j, z_j, z'_j)_{j\in[T]}, \pi)$

$\mathsf{Solve}(\mathsf{pp} = (\mathsf{pp}', \sigma, \omega), k, B_1 = ((c_{j,1}, z_{j,1}, z'_{j,1})_{j\in[T]}, \pi_1), \ldots, B_T = ((c_{j,T}, z_{j,T}, z'_{j,T})_{j\in[T]}, \pi_T))$:
1. $\forall i \in [T] : b_i \leftarrow \mathsf{NIZK.Verify}(\sigma, (\lambda, (z_{j,i}, z'_{j,i}, c_{j,i})_{j\in[T]}, \mathsf{pp}', \omega), \pi_i)$
2. $\forall i < j$ such that $B_i = B_j$, set $b_j \leftarrow 0$
3. $S \leftarrow \{i | b_i \neq 0\}$
4. $\forall i \in [T] \setminus S : z_{i,i} \leftarrow \mathsf{BHTLP.Gen}(1^\lambda, i, \mathsf{pp}', 0; 0)$
5. $\forall i \in [T] \setminus S : z'_{i,i} \leftarrow z_{i,i}$
6. $a_0 \leftarrow \bot$
7. $a'_0 \leftarrow \bot$
8. $\forall i \in [T] : a_i \leftarrow \mathsf{BHTLP.BatchSolve}(\mathsf{pp}', a_{i-1}, z_{i,i})$
9. $\forall i \in [T] : a'_i \leftarrow \mathsf{BHTLP.BatchSolve}(\mathsf{pp}', a'_{i-1}, z'_{i,i})$
10. $s \leftarrow a_T$
11. $v \leftarrow a'_T$
12. Output $r := \mathcal{H}(k \parallel s)$ and $w := (s, v)$.

$\mathsf{FastSolve}(\mathsf{pp} = (\mathsf{pp}', \sigma, \omega), k, T, B = ((c_j, z_j, z'_j)_{j\in[T]}, \pi), \mathsf{aux} = ((a_{k-1,1}, a'_{k-1,1}, B_1), \ldots, (a_{k-1,T}, a'_{k-1,T}, B_{T-1}))$:

1. $b \leftarrow \mathsf{NIZK.Verify}(\sigma, (\lambda, (z_j, z'_j c_j)_{j\in[T]}, \mathsf{pp}', \omega), \pi)$
2. If $\exists i \in [T-1]$ such that $B = B_i$, set $b \leftarrow 0$
3. If $b = 0$, $\forall j \in [T] : z_j \leftarrow \mathsf{BHTLP.Gen}(1^\lambda, j, \mathsf{pp}', 0; 0)$
4. If $b = 0$, $\forall j \in [T] : z'_j \leftarrow z_j$
5. $a_{k-1,0} \leftarrow \bot$
6. $a'_{k-1,0} \leftarrow \bot$
7. $\forall j \in [T] : a_{k,j} \leftarrow \mathsf{BHTLP.BatchSolve}(\mathsf{pp}', a_{k-1,j-1}, z_j)$
8. $\forall j \in [T] : a'_{k,j} \leftarrow \mathsf{BHTLP.BatchSolve}(\mathsf{pp}', a'_{k-1,j-1}, z'_j)$
9. $s \leftarrow a_{k,T}$
10. $v \leftarrow a'_{k,T}$
11. $\mathsf{aux}' \leftarrow ((a_{k,0}, a'_{k,0}, B_2), \ldots, (a_{k,T-2}, a'_{k,T-2}, B_{T-1}), (a_{k,T-1}, a'_{k,T-1}, B))$
12. Output $r := \mathcal{H}(k \parallel s)$, $w := (s, v)$ and aux'.

$\mathsf{Verify}(\mathsf{pp} = (\mathsf{pp}', \sigma, \omega), k, B_1 = ((c_{j,1}, z_{j,1}, z'_{j,1})_{j\in[T]}, \pi_1), \ldots, B_T = ((c_{j,T}, z_{j,T}, z'_{j,T})_{j\in[T]}, \pi_T), r, w = (s, v))$:
1. $\forall i \in [T] : b_i \leftarrow \mathsf{NIZK.Verify}(\sigma, (\lambda, (z_{j,i}, z'_{j,i}, c_{j,i})_{j\in[T]}, \mathsf{pp}', \omega), \pi_i)$
2. $\forall i < j$ such that $B_i = B_j$, set $b_j \leftarrow 0$
3. $S \leftarrow \{i | b_i \neq 0\}$
4. $\iota \leftarrow \min S$
5. $c \leftarrow c_{\iota,\iota}$
6. $\forall i \in S \setminus \{\iota\} : c \leftarrow \mathsf{C.Add}(\omega, c, c_{i,i})$
7. $b' \leftarrow \mathsf{C.Check}(c, s, v)$
8. If $r = \mathcal{H}(k \parallel s)$, output b'. Otherwise, output 0.

Fig. 18.

Suppose that $\mathcal{C}.\mathsf{Commit}$*,* $\mathcal{C}.\mathsf{Add}$*,* $\mathcal{C}.\mathsf{Check}$*,* $\mathsf{NIZK.Prove}$*,* $\mathsf{NIZK.Verify} \in \mathcal{C}_{f_1}$*. Suppose also that the depth of* $\mathsf{NIZK.Ext}$ *is upper-bounded by* $f_1(\lambda)$ *and the depth of* $\mathsf{NIZK.Sim}_2$ *is upper-bounded by* $f_4(\lambda)$*. Then, if the* $f_0(\lambda) = f(\lambda) \cdot \omega(\log \lambda) + f_1(\lambda) + f_4(\lambda) + \omega(\log \lambda)$*, the construction in Fig. 18 is a randomness beacon satisfying completeness, correctness, soundness,* $(\mathcal{C}_{f_1}, \mathcal{C}_{T \cdot f_3}, \mathcal{C}_{f_3})$*-efficiency and* $(\overline{\mathcal{C}}_f, T, T)$*-simulation security in the programmable random oracle model.*

Proof. We start our proof by considering two hybrids.

Hybrid 0. This correspond to the original randomness beacon described in Fig. 18.

Hybrid 1. In this hybrid, we change the distribution of the setup of the randomness beacon: instead of computing (pp', q) using $\mathsf{BHTLP.Setup}$, we sample $q \leftarrow\!\!\$\ \mathsf{C.D}(1^\lambda)$ and $\mathsf{pp}' \leftarrow\!\!\$\ \mathsf{BHTLP.Sim}(1^\lambda, 1^{t(\lambda) \cdot T(\lambda)}, q)$. Observe that the distribution of the public parameters of the randomness beacon in Hybrid 1 is computationally indistinguishable from the original one.

We show the completeness and correctness of the construction. Consider T blocks $B_1, \dots, B_T$ generated according to $\mathsf{Gen}(\mathsf{pp})$ where $\mathsf{pp} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)$. Let $(s_{j,i}, v_{j,i})_{j \in [T]}$ be the values hidden in the time-lock puzzles $(z_{j,i}, z'_{j,i})_{j \in [T]}$ in the block B_i. Notice that for every $i, j \in [T]$, the pair $(s_{j,i}, v_{j,i})$ is an opening of $c_{j,i}$. We observe that with overwhelming probability $B_i \neq B_j$ for every $i \neq j$. Furthermore, thanks to the completeness of NIZK, all the NIZKs in the T blocks verify. So, during the execution of $\mathsf{Solve}(\mathsf{pp}, k, B_1, \dots, B_T)$, the set S coincides with $[T]$. Therefore, by the correctness of the batched homomorphic time-lock puzzle, we obtain that $w = (s, v)$ is such that $s = (s_{1,1} + s_{2,2} + \cdots + s_{T,T}) \bmod q$ and $v = (v_{1,1} + v_{2,2} + \cdots + v_{T,T}) \bmod q$. By the correctness of the strongly homomorphic commitment, we conclude that, in Hybrid 1, (s, v) is a valid opening of the commitment c computed during the execution of $\mathsf{Verify}(\mathsf{pp}, k, B_1, \dots, B_t, r, w)$. Since Hybrid 1 is indistinguishable from the original construction, the same must hold also in Hybrid 0 with overwhelming probability. Completeness therefore follows from the fact that $\mathsf{Solve}(\mathsf{pp}, k, B_1, \dots, B_T)$ output $r = \mathcal{H}(k \parallel s)$.

Correctness can be easily verified: let $B_1, \dots, B_M$ be a sequence of blocks. Let $k \in [M]$ be an index greater than T. We observe that the value a_i computed in the execution of $\mathsf{Solve}(\mathsf{pp}, k, B_{k-T+1}, \dots, B_k)$ coincides with the value $a_{k-T+i,i}$ computed by $\mathsf{FastSolve}(\mathsf{pp}, k-T+i, T, B_{k-T+i}, \mathsf{aux})$. This can be easily be verified by induction over i. In a similar way, the value a'_i computed in the execution of $\mathsf{Solve}(\mathsf{pp}, k, B_{k-T+1}, \dots, B_k)$ coincides with the value $a'_{k-T+i,i}$ computed by $\mathsf{FastSolve}(\mathsf{pp}, k-T+i, T, B_{k-T+i}, \mathsf{aux})$.

Moving on to soundness, suppose that there exists a PPT adversary capable of generating, with non-negligible probability, an index k, blocks $B_1, \dots, B_T$ and values r, w such that $\mathsf{Verify}(\mathsf{pp}, k, B_1, \dots, B_T, r, w) = 1$ but r is different from the value output by $\mathsf{Solve}(\mathsf{pp}, k, B_1, \dots, B_T)$. We consider a new hybrid.

Hybrid 2. In this hybrid we change the distribution of the setup of the randomness beacon: instead of generating σ suing $\mathsf{NIZK.Setup}(1^\lambda)$, we generate using $\mathsf{NIZK.Sim}_1(1^\lambda)$. Along with σ, this algorithm provides us with a trapdoor ζ. We also change the verification and solving algorithm. For every, $i \in [T]$, instead of computing $\mathsf{NIZK.Verify}(\sigma, (\lambda, (z_{j,i}, z'_{j,i}, c_{j,i})_{j \in [T]}, \mathsf{pp}', \omega), \pi_i)$, we compute

$$\mathsf{NIZK.Ext}(\zeta, (\lambda, (z_{j,i}, z'_{j,i}, c_{j,i})_{j \in [T]}, \mathsf{pp}', \omega), \pi_i).$$

If the extraction of the witness fails, we set $b_i \leftarrow 0$, otherwise, we set $b_i \leftarrow 1$.

By the simulation extractability of the NIZK, Hybrid 2 is computationally indistinguishable from Hybrid 0. So, soundness is broken even in such setting. Now, consider the evaluation of $\mathsf{Verify}(\mathsf{pp}, k, B_1, \dots, B_T, r, w) = 1$ in Hybrid 2 and let $(s_{i,j}, v_{i,j}, u_{j,0}, u_{j,1}, u'_{j,1})_{j \in [T]}$ be the witness

extracted from the NIZK π_i for every $i \in S$. Let E be the event in which $s \neq s'$ where $w = (s', v')$ and $s = \sum_{i \in S} s_{i,i} \bmod q$. Define $v = \sum_{i \in S} v_{i,i}$. By the strong binding property of the commitment scheme C, the probability of the event E is negligible. Indeed, by the correctness of the commitment, $\mathcal{C}.\mathsf{Check}(\omega, c, s, v) = 1$. Moreover, with non-negligible probability, the adversary would generate (s', v') such that $\mathcal{C}.\mathsf{Check}(\omega, c, s', v') = 1$ and $s' \neq s$. We conclude that $s' = s$ with overwhelming probability. By the correctness of the batched homomorphic time-lock puzzle, in Hybrid 2, $\mathsf{Solve}(\mathsf{pp}, k, B_1, \ldots, B_T)$ outputs (s, v). The last check performed in the randomness beacon verification ensures soundness.

The efficiency of the construction can be easily verified. Notice indeed that $\mathcal{C}_{f_1}, \mathcal{C}_{T \cdot f_3}, \mathcal{C}_{f_3}$ are nicely closed classes. Furthermore, we observe that while Solve needs to run $\mathsf{BHTLP.BatchSolve}$ in series, $\mathsf{FastSolve}$ can parallelise the procedure. For this reason, the depth of Solve turns out to be T times larger than the depth of $\mathsf{FastSolve}$.

Finally, we focus on simulation security. We consider the simulator $\mathsf{Sim} = (\mathsf{Sim}_0, \mathsf{Sim}_1, \mathsf{SimRO})$ defined as follows:

- $\mathsf{Sim}_0(1^\lambda)$:
  1. $(\sigma, \zeta) \leftarrow\$ \mathsf{NIZK.Sim}_1(1^\lambda)$
  2. $q \leftarrow\$ \mathsf{C.D}(1^\lambda)$
  3. $\mathsf{pp}' \leftarrow\$ \mathsf{BHTLP.Sim}(1^\lambda, 1^{t(\lambda) \cdot T(\lambda)}, q)$
  4. $\omega \leftarrow\$ \mathsf{C.Setup}(1^\lambda, q)$
  5. Output $\mathsf{pp} := (\mathsf{pp}', \sigma, \omega)$ and $\phi = (1^\lambda, q, \mathsf{pp}', \sigma, \omega, \zeta)$
- $\mathsf{Sim}_1(\phi = (1^\lambda, q, \mathsf{pp}', \sigma, \omega, \zeta, (j, r^1_j, s_j, v_j)_{j < i+T-1}), B_i, r)$:
  - If $B_i = \perp$:
    1. $\forall j \in [T(\lambda)] : s_{j,i} \leftarrow\$ \mathbb{Z}_q$
    2. $\forall j \in [T(\lambda)] : (c_{j,i}, v_{j,i}) \leftarrow\$ \mathsf{Commit}(\omega, s_{j,i})$
    3. $\forall j \in [T(\lambda)] : z_{j,i} \leftarrow\$ \mathsf{BHTLP.Gen}(1^\lambda, j, \mathsf{pp}', s_{j,i})$
    4. $\forall j \in [T(\lambda)] : z'_{j,i} \leftarrow\$ \mathsf{BHTLP.Gen}(1^\lambda, j, \mathsf{pp}', v_{j,i})$
    5. $\pi_i \leftarrow\$ \mathsf{NIZK.Sim}_2(\zeta, (\lambda, (z_{j,i}, z'_{j,i}, c_{i,j})_{j \in [T]}, \mathsf{pp}', \omega))$
    6. $s_{i+T-1} \leftarrow 0$
    7. $v_{i+T-1} \leftarrow 0$
    8. $r^1_{i+T-1} \leftarrow \perp$
    9. $r^1_i \leftarrow r$
    10. $\forall j \in [T(\lambda)] : s_{i-1+j} \leftarrow (s_{i-1+j} + s_{T+1-j,i}) \bmod q$
    11. $\forall j \in [T(\lambda)] : v_{i-1+j} \leftarrow (v_{i-1+j} + v_{T+1-j,i}) \bmod q$
    12. Output $B^1_i := ((c_{j,i}, z_{j,i}, z'_{j,i})_{j \in [T]} \pi_i)$, $w_i = (s_i, v_i)$ and $\phi := (1^\lambda, q, \mathsf{pp}', \sigma, \omega, \zeta, (j, r^1_j, s_j, v_j)_{j \leq i+T-1})$
- If $B_i = ((c_{j,i}, z_{j,i}, z'_{j,i})_{j \in [T]}, \pi_i) \neq \perp$:
  1. $x := (s_{j,i}, v_{j,i}, u_{j,i,0}, u_{j,i,1}, u'_{j,i,1})_{j \in [T]} \leftarrow \mathsf{NIZK.Ext}(\zeta, (\lambda, (c_{j,i}, z_{j,i}, z'_{j,i})_{j \in [T]}, \mathsf{pp}', \omega), \pi_i)$
  2. If $x = \perp$, $\forall j \in [T(\lambda)] : (s_{j,i}, v_{j,i}) \leftarrow (0, 0)$
  3. $s_{i+T-1} \leftarrow 0$
  4. $v_{i+T-1} \leftarrow 0$
  5. $r^1_{i+T-1} \leftarrow \perp$
  6. $r^1_i \leftarrow r$
  7. $\forall j \in [T(\lambda)] : s_{i-1+j} \leftarrow (s_{i-1+j} + s_{T+1-j,i}) \bmod q$
  8. $\forall j \in [T(\lambda)] : v_{i-1+j} \leftarrow (v_{i-1+j} + v_{T+1-j,i}) \bmod q$
  9. Output $w_i = (s_i, v_i)$ and $\phi := (1^\lambda, q, \mathsf{pp}', \sigma, \omega, \zeta, (j, r^1_j, s_j, v_j)_{j \leq i+T-1})$

– $\mathsf{SimRO}(\phi, \widehat{x})$ rewrites $\widehat{x}$ as a string $(i \parallel s_i')$. Then, it retrieves the tuple (i, r_i^1, s_i, v_i) stored in ϕ. Finally, if $s_i = s_i'$, the algorithm outputs r_i^1. In all other cases, including when one of the above procedures fails, the algorithm outputs a random string in $\{0,1\}^{L(\lambda)}$.

We consider a new hybrid.

Hybrid 3. In this hybrid we change the distribution of the honestly generated blocks: instead of generating the NIZKs π using $\mathsf{NIZK.Prove}$, we rely on $\mathsf{NIZK.Sim}_2$. This hybrid is computationally indistinguishable from Hybrid 2 thanks to the zero-knowledge property of the NIZK.

Now, we define the event E_1 in which the adversary publishes a block where the corresponding NIZK verifies but the extraction of the witness fails. By the simulation extractability of the NIZK, the probability of the event E_1 occurring is negligible. Along with the correctness of BHTLP and C, this guarantees that the proofs w_i given to the adversary are consistent with the published blocks with overwhelming probability.

Now, we define the event E_2 as the event in which the adversary queries a tuple $(i \parallel s)$ to the random oracle, where $w_i = (s, v)$, before the block B_i is published.

We observe that the adversary can distinguish between Hybrid 3 and the ideal world execution of the security game (i.e. when $b = 1$) if and only if the probability of E_2 is non-negligible. Notice that the probability of E_2 in the two worlds are the same except for a negligible amount. Suppose that the probability of E_2 in the ideal world is non-negligible. Since the adversary halts after a polynomial number of executions of the procedure **New Block**, there exists an index i_λ such that there exists a non-negligible probability that the adversary queries $(i_\lambda \parallel s)$ to the random oracle, where $w_{i_\lambda} = (s, v)$, before the block B_{i_λ} is revealed. We call any of these queries *bad queries*. Let h be the highest index such that B_h is honestly generated and $h \leq i_\lambda$. Since the adversary is T-respecting, we know that $h \geq i_\lambda - T(\lambda) + 1$.

We consider two new hybrids.

Hybrid 4. In this hybrid, we change the distribution of the time-lock puzzles $z_{T-i+h,h}$ and $z'_{T-i+h,h}$ in block B_h. Specifically, we generate both of them using $\mathsf{BHTLP.Gen}(1^\lambda, T-i+h, \mathsf{pp}', 0)$. We observe that, since E_2 can be verified in depth $f_4(\lambda)+(i-h)\cdot f(\lambda)\cdot O(\log\lambda)+(i-h)\cdot f_1(\lambda)+(i-h)\cdot O(\log\lambda)$ since the generation of $z_{T-i+h,h}$ and $z'_{T-i+h,h}$ (notice that this depth is smaller than $(i-h)\cdot f_0(\lambda)$ for sufficiently large λ), the probability of the event E_2 must still be non negligible thanks to the $(\mathcal{C}, T, t)$-security of the batched homomorphic time-lock puzzle[8].

Hybrid 5. In this hybrid, we change the distribution of the commitment $c_{T-i+h,h}$ in the block B_h. Specifically, instead of committing to $s_{T-i+h,h}$, we commit to 0. Hybrid 5 is computationally indistinguishable from Hybrid 4 thanks to the hiding property of the commitment scheme. This implies that the probability of the event E_2 is still non-negligible.

We reach a contradiction: the adversary receives no information about $s_{T-i+h,h}$, so the probability that it guesses $s_i = \sum_{j\in[T]} s_{T-j+1,i-j+1}$ is negligible in λ. □

Acknowledgements. The initial version of this work contained an additional result about a construction of fully homomorphic time-lock puzzles. It was pointed out to us by Nico Döttling that the exact same construction we proposed, had already appeared in prior work by Brakerski et al. [BDGM19]. For this reason, we have removed this part of the paper.

[8] $f(\lambda)$ is multiplied by $O(\log\lambda)$ due to the depth of the operations computed by SimRO for each oracle query.

References

ADIN24. Damiano Abram, Jack Doerner, Yuval Ishai, and Varun Narayanan. Constant-Round Simulation-Secure Coin Tossing Extension with Guaranteed Output. In *EUROCRYPT 2024*, 2024. 1.1, 6.3

ADOS22. Damiano Abram, Ivan Damgård, Claudio Orlandi, and Peter Scholl. An algebraic framework for silent preprocessing with trustless setup and active security. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 421–452, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany. 1.1, 2.1, 6.3

ARS24. Damiano Abram, Lawrence Roy, and Peter Scholl. Succinct Homomorphic Secret Sharing. In *EUROCRYPT 2024*, 2024. 1.1, 6.3

BBBF18. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 757–788, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany. 1, 1.1, 1.2, 2.1, 6

BCP14. Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In Yehuda Lindell, editor, *TCC 2014: 11th Theory of Cryptography Conference*, volume 8349 of *Lecture Notes in Computer Science*, pages 52–73, San Diego, CA, USA, February 24–26, 2014. Springer, Heidelberg, Germany. 4

BD21. Jeffrey Burdges and Luca De Feo. Delay encryption. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 302–326, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany. 1, 1.1, 2.3, 3.2

BDD+21. Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. TARDIS: A foundation of time-lock puzzles in UC. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part III*, volume 12698 of *Lecture Notes in Computer Science*, pages 429–459, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany. 1.2

BDD+23. Carsten Baum, Bernardo David, Rafael Dowsley, Ravi Kishore, Jesper Buus Nielsen, and Sabine Oechsner. CRAFT: Composable randomness beacons and output-independent abort MPC from time. In *PKC 2023: 26th International Conference on Theory and Practice of Public Key Cryptography, Part I*, Lecture Notes in Computer Science, pages 439–470. Springer, Heidelberg, Germany, May 10–13, 2023. 1.2, 1.2

BDGM19. Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part II*, volume 11892 of *Lecture Notes in Computer Science*, pages 407–437, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany. 6.3

BGG+14. Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 533–556, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany. 2.3, 5.2, 5.2, 6, 7

BGI+01. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany. 4

BGJ+16. Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 345–356, Cambridge, MA, USA, January 14–16, 2016. Association for Computing Machinery. 1.2

BHR12. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 784–796, Raleigh, NC, USA, October 16–18, 2012. ACM Press. 2.2, 4.1, 30, 31, 32, 4.1

BN00. Dan Boneh and Moni Naor. Timed commitments. In Mihir Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 236–254, Santa Barbara, CA, USA, August 20–24, 2000. Springer, Heidelberg, Germany. 1, 1.2, 1.2

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press. 1.2

Cle86. Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th Annual ACM Symposium on Theory of Computing*, pages 364–369, Berkeley, CA, USA, May 28–30, 1986. ACM Press. 1

CLSY93. Jin-yi Cai, Richard J. Lipton, Robert Sedgewick, and Andrew Chi-Chih Yao. Towards uncheatable benchmarks. In *Proceedings of the Eigth Annual Structure in Complexity Theory Conference, San Diego, CA, USA, May 18-21, 1993*, pages 2–11. IEEE Computer Society, 1993. 1, 1.2

CMB23. Kevin Choi, Aathira Manoj, and Joseph Bonneau. Sok: Distributed randomness beacons. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 75–92. IEEE, 2023. 1, 1.2

DGKR18. Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany. 1

EFKP20. Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Continuous verifiable delay functions. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 125–154, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany. 1.2

Eth. Ethereum randomness beacon randao. https://eth2book.info/capella/part2/building_blocks/randomness/ (Accessed 2024-03-06). 1

FKPS21. Cody Freitag, Ilan Komargodski, Rafael Pass, and Naomi Sirkin. Non-malleable time-lock puzzles and applications. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part III*, volume 13044 of *Lecture Notes in Computer Science*, pages 447–479, Raleigh, NC, USA, November 8–11, 2021. Springer, Heidelberg, Germany. 1.2, 1.2

GGSW13. Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 467–476, Palo Alto, CA, USA, June 1–4, 2013. ACM Press. 1.1, 3.3

GHM+17. Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017. 1

GO07. Jens Groth and Rafail Ostrovsky. Cryptography in the multi-string model. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 323–341, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany. 3.5

GS98. David M. Goldschlag and Stuart G. Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In Rafael Hirschfeld, editor, *FC'98: 2nd International Conference on Financial Cryptography*, volume 1465 of *Lecture Notes in Computer Science*, pages 214–226, Anguilla, British West Indies, February 23–25, 1998. Springer, Heidelberg, Germany. 1, 1.1, 4

HLL23. Yao-Ching Hsieh, Huijia Lin, and Ji Luo. Attribute-based encryption for circuits of unbounded depth from lattices. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 415–434. IEEE, 2023. 2.3, 5.2

JLS21. Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In Samir Khuller and Virginia Vassilevska Williams, editors, *53rd Annual ACM Symposium on Theory of Computing*, pages 60–73, Virtual Event, Italy, June 21–25, 2021. ACM Press. 4

JMR20. Samuel Jaques, Hart Montgomery, and Arnab Roy. Time-release cryptography from minimal circuit assumptions. Cryptology ePrint Archive, Report 2020/755, 2020. https://eprint.iacr.org/2020/755. 1.2

Kil88. Joe Kilian. Founding cryptography on oblivious transfer. In *20th Annual ACM Symposium on Theory of Computing*, pages 20–31, Chicago, IL, USA, May 2–4, 1988. ACM Press. 1

KLX20. Jonathan Katz, Julian Loss, and Jiayu Xu. On the security of time-lock puzzles and timed commitments. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part III*, volume 12552 of *Lecture Notes in Computer Science*, pages 390–413, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany. 1.2

KS08. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498, Reykjavik, Iceland, July 7–11, 2008. Springer, Heidelberg, Germany. 4.1

LMP+23. Gaëtan Leurent, Bart Mennink, Krzysztof Pietrzak, Vincent Rijmen, Alex Biryukov, Benedikt Bunz, Anne Canteaut, Itai Dinur, Yevgeniy Dodis, Orr Dunkelman, Ben Fisch, Ilan Komargodski, Nadia Heninger, Maria Naya Plasencia, Leo Perrin, Christian Rechberger, Gil Segev, Martin Stam, Stefano Tessaro, Benjamin Wesolowski, Mike Schaffstein, Dankrad Feist, Herold Gottfried, Antonio Sanso, Mark Simkin, and Dmitry Khovratovich. Analysis of minroot. 2023. 1

May93. Timothy C. May. Time-release crypto. 1993. https://mailing-list-archive.cryptoanarchy.wiki/archive/1993/02/a421c6fc805dfb4ae4197521e8a9e91dd456e3deab855f12af31a4b1ccccf6cb/ (Accessed 2024-03-08). 1, 1.2

MMV11. Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Time-lock puzzles in the random oracle model. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 39–50, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany. 1.2

MSW20. Mohammad Mahmoody, Caleb Smith, and David J. Wu. Can verifiable delay functions be based on random oracles? In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *ICALP 2020: 47th International Colloquium on Automata, Languages and Programming*, volume 168 of *LIPIcs*, pages 83:1–83:17, Saarbrücken, Germany, July 8–11, 2020. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. 1.2

MT19. Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. Homomorphic time-lock puzzles and applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 620–649, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany. 1, 1.1, 1.2, 2.1, 2.1, 3.1, 6, 6.3, 6.3, 6.3, 6.3

PF79. Nicholas Pippenger and Michael J Fischer. Relations among complexity measures. *Journal of the ACM (JACM)*, 26(2):361–381, 1979. 3

Pie19. Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, *ITCS 2019: 10th Innovations in Theoretical Computer Science Conference*, volume 124, pages 60:1–60:15, San Diego, CA, USA, January 10–12, 2019. LIPIcs. 1.2

PT23. Chris Peikert and Yi Tang. Cryptanalysis of lattice-based sequentiality assumptions and proofs of sequential work. *IACR Cryptol. ePrint Arch.*, 2023. 1

Rab83. Michael O. Rabin. Transaction protection by beacons. *J. Comput. Syst. Sci.*, 27(2):256–267, 1983. 1

Rog91. Phillip Rogaway. *The Round Complexity of Secure Protocols*. PhD thesis, Massachusetts Institute of Technology, USA, 1991. 2.2, 3.6, 4.1, 4.1, 4.1, 4.1, 2

RS20. Lior Rotem and Gil Segev. Generically speeding-up repeated squaring is equivalent to factoring: Sharp thresholds for all generic-ring delay functions. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 481–509, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany. 1.2

RSS20. Lior Rotem, Gil Segev, and Ido Shahaf. Generic-group delay functions require hidden-order groups. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 155–180, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany. 1.2

RSW96. Ronald Rivest, Adi Shamir, and David Wagner. Time-Lock Puzzles and Time-Released Crypto. Technical Report MIT/LCS/TR-684, Massachusetts Institute of Technology, Cambridge, USA, 1996. 1, 1.2, 3.1, 6

SW05. Amit Sahai and Brent R. Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473, Aarhus, Denmark, May 22–26, 2005. Springer, Heidelberg, Germany. 1.1

Val76. Leslie Valiant. Universal circuits (Preliminary Report). In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, page 196–203, New York, NY, USA, 1976. Association for Computing Machinery. 2.2

VWW22. Vinod Vaikuntanathan, Hoeteck Wee, and Daniel Wichs. Witness encryption and null-IO from evasive LWE. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part I*, volume 13791 of *Lecture Notes in Computer Science*, pages 195–221, Taipei, Taiwan, December 5–9, 2022. Springer, Heidelberg, Germany. 2.3, 5.1

Wes19. Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 379–407, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany. 1.2

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press. 2.2, 3.6, 4.1, 2