An Introduction To Time-lock Puzzles

Alexander Haberl
Technical University of Munich
Munich, Germany
alexander.haber@tum.de

some applications?

***Abstract*—Timed release cryptography aims to hide secrets from anyone for a specified amount of time and wants to eliminate any advantage a highly parallel party might have over a single non-parallel party. This notion that has often been called sending messages to the future seems to have great potential for inventing new protocols and upgrading old ones.**

One approach to timed release cryptography relies on trusted third parties to publish decryption keys for time-locked secrets in intervals or at specific points in the future. The other is time-lock puzzles that hide secrets behind sequential non-parallelizable computations. Of these time-lock puzzles have seen less progress with the big hurdle being, that for a long time only one method of construction was known.

But now that multiple well-founded constructions are known we want to give a simple introduction to them and motivate research in this field. To this end we have compiled three constructions of time-lock puzzles with different basis, which are repeated squaring in finite fields, the random oracle model and randomized encodings. We analyze their security and compare them to one another.

We find all three constructions to be secure time-lock puzzles that are worth exploring more. We conclude that now is the time for more research about applying time-lock puzzles to new or old protocols and applications, as overreliance on one type of time-lock puzzle is no longer of imminent concern.

I. Introduction

Timed release cryptography first introduced by May [1] is an interesting cryptographic concept. Standard cryptography aims to hide messages from everyone but a specific group of recipients. On the other hand the primary purpose of timed release encryption is to hide a secret from anyone for a specified amount of time. To this end May proposed a system which relies on trusted third parties to keep the secret save for a specified amount of time after which it can be accessed or distributed.

Later Rivest, Shamir and Wagner were the first to propose a different approach which they named time-lock puzzles [2]. The idea behind this approach is to have a computational puzzle that takes a specified amount of time to solve and hides a secret. This way anyone trying to attain the secret has to first solve the puzzle. So far this could also be achieved with any other cryptographic puzzle, but time-lock puzzles add the constraint that it cannot be solved significantly faster when using a high degree of parallel computing power. This guarantees that the secret is save even from adversaries that have much more computational power than assumed when constructing the puzzle.

Since the invention of timed release cryptography many applications have been proposed. This includes everything from a person wanting to encrypt their diaries until after their death, encrypting your bids in an auction, using it in cryptographic protocols, to using it in encapsulated key escrow. With these applications in mind both implementations have upsides and downsides.

We now introduce the properties of both approaches to timed release cryptography. May's approach using trusted agents is exact and uses relatively little power. Time-lock puzzles suffer from the problem of approximating real time through computational time which becomes more inexact with longer encryption periods. They also use a lot of power for the needed computation. But they have the upside of not relying on third parties which might be unacceptable when locking highly sensitive information. They differ in one aspect, as in May's approach the information is available after a specific point in time while with time-lock puzzles it becomes available after a specific time period after starting the decryption. Especially this last difference makes both approaches viable for different applications.

In the case of the diary encryption May's approach works better, as the point in time could be many years in the future and can be given more exactly with this approach. But for encapsulated key escrow time-lock puzzles are the better option.

Key escrow is the concept of storing cryptographic keys of people so that a higher authority, for example the government, may be able to wiretap encrypted traffic of certain suspects, for criminal investigation for example. A big concern of key escrow is that these keys could easily be used by corrupt individuals or organizations to perform wiretapping on a large scale. This is where encapsulated key escrow comes in. Here the actual keys first have to be recovered through some lengthy process, thereby limiting the amount of keys that can be obtained in a short amount of time. Here we are dealing with highly sensitive information, so trusted third parties that can be pressured into giving up these keys by corrupt governments are undesirable. And the fact that time-lock puzzles lock the information for a time period after starting decryption is also a desirable property in this case.

Both branches of timed-release cryptography have seen much research and in this paper we want to focus on time-lock puzzles as we think that timed-release encryption could be used in cryptographic protocols and time-lock puzzles would be better suited for this purpose than trusted third parties. This will be discussed in further detail in section V. As an example time-lock puzzles have already been used for

timed commitments introduced in [3] which fulfill the same role as commitments, but a commitment can be forced open by solving a time-lock puzzle if the other party does not cooperate.

Even though a lot of research has gone into finding more candidates for time-lock puzzles, we only know of three different constructions. The combined constraints of having a puzzle that can not be solved faster through parallelization while preferably being faster to generate than to solve has made finding suitable candidates for time-lock puzzles rather difficult.

Our contributions consist of highlighting the three known constructions of time-lock puzzles to heighten the presence of this cryptographic concept and comparing them to one another on their merits and demerits. To this end we describe construction and prove security of three time-lock puzzles based on:

- **repeated squaring in finite fields** (, also the first ever time-lock puzzle proposed by Rivest, Shamir and Wagner in [2])
- **the random oracle model** (, where the existence of time-lock puzzles has been analyzed by Mahmoody et al. in [4])
- **randomized encodings** (these time-lock puzzles were introduced by Bitansky et al. in [5] where they analyze time-lock puzzles from a complexity theoretic stand point)

II. Background

A. Finite Fields

For any integer n we define the group of coprime residues modulo n as $\mathbb{Z}_n^* = \{x \in \mathbb{Z} : 0 < x < n, gcd(x, n) = 1\}$. In a finite multiplicative group G every element a has an order which is defined as follows: $ord(a) = \langle a \rangle = \min\{k : a^k = 1, k \in \mathbb{N}\}$. From this it follows that we can simplify large exponentiation as follows: $a^k = a^{k \bmod \langle a \rangle}$, $k \in \mathbb{N}$. Since $|G|$ will always be a multiple of $\langle a \rangle$ in finite groups we can simplify exponentiation without knowing the order of an element. This means: $\forall a \in G$, $a^k = a^{k \bmod |G|}$, $k \in \mathbb{N}$.

B. Random Oracle Model

The random oracle model is a computational model that makes use of random oracles for theoretical proofs in cryptography. As an example such a random oracle could replace a cryptographic hash function in proofs where strong randomness assumptions are needed. The random oracle is an oracle which answers any query randomly, chosen uniformly from all possible output, but the same query will always result in the same response. Results in the random oracle model cannot be directly translated to the standard model of cryptography, but are a first step in this direction.

For the purpose of this paper we will assume that any party has access to a random Oracle R that responds to query q with R(q). We will measure the complexity of time-lock puzzles in the random Oracle model according to the amount of sequential queries needed by the different algorithms.

C. Theoretical Computer Science

We will use Turing machines to model honest algorithms and non-uniform circuit families to model non-uniform adversaries. Turing machines are often used in theoretical analysis of complexity theory because they are as computationally powerful as any modern day programming language, but for them it is easy to define what a single step of a computation is. We will represent the running time f(n) of a deterministic Turing machine in $Dtime(f(n))$.

A circuit C is constructed from logic gates which can run in parallel. The parallel running time of a circuit corresponds to its Depth $dep(C)$ and it's total sequential running time would be the size of the circuit $|C|$. A non-uniform circuit family $\{C_\lambda\}_{\lambda \in \mathbb{N}}$ is a set of circuits in which there exists a different circuit for every possible input length. So C_i is used to compute an algorithm for inputs of size i. This means in particular that the same problem could be solved in different ways for input of different sizes.

D. Indistinguishability Obfuscation

Randomized encodings that form the basis of a time-lock puzzle rely on indistinguishability obfuscation at the moment. So we want to give a short introduction to what it achieves. Indistinguishability obfuscationis a method to obfuscate programs, such that to an outside observer the program is unintelligible while it still functions as designed. For a long time all known schemes relied on not well-studied hardness assumptions. But recently [6] proposed a construction for indistinguishability obfuscation for polynomially sized circuits while only relying on well-founded hardness assumptions. The assumptions made are the following:

- Learning with errors assumption
- Learning parity with noise assumption
- Existence of a boolean pseudo-random generator in NC^0
- Decision linear assumption on symmetric bilinear groups of prime order and what are those?

III. Time-Lock Puzzle Constructions

In this section we will introduce three different methods of time-lock puzzle constructions and prove their security. For this we first define what we mean by time-lock puzzles and introduce the three variants afterwards.

Definition 1 (Time-Lock Puzzle): A time-lock puzzle consists of a pair of algorithms (Puz.Gen, Puz.Solv) with the following properties.

- **Algorithms**:
  - Puz.Gen(s, t) is a probabilistic algorithm that takes a solution s and a difficulty parameter t and outputs the puzzle Z.
  - Puz.Solv(Z) is a deterministic algorithm that takes a time-lock puzzle Z and outputs the solution s.
- **Completeness**: We require that for every s, t and Z, output by Puz.Gen(s, t), Puz.Solv(Z) outputs s.

- **Security**: Any poly-size adversary running in $o(Puz.Solv)$ must have a negligible chance of arriving at solution s.
- **Efficiency**: There exists a difficulty gap between Puz.Gen and Puz.Solv. More precisely we will define strong time-lock puzzles for which $Puz.Gen \in o(Puz.Solv)$ holds and weak time-lock puzzles for which $Puz.Gen \in O(Puz.Solv)$ holds but Puz.Gen can use parallelism while Puz.Solv and any adversary can not make use of parallelism. So for strong time-lock puzzles we have a large difficulty gap between generating and solving the puzzle and for weak time-lock puzzles we only have a linear difficulty gap in parallel time.

A. Repeated Squaring

The first method relies on repeated squaring in finite fields and was first introduced in [2]. It qualifies as a basis for a time-lock puzzle as repeated squaring in finite fields is thought to be an inherently sequential computation. But there exists a shortcut if one knows the order of the finite field.This can easily be used during instantiation but requires extensive computation during solving.

Algorithm description:

- Puz.Gen(s, t) encrypts solution s using a secure algorithm which gives us ciphertext C_s. It then hides the decryption key k as cipher C_k.

$$C_k = k + b,\ b = a^{2^t} \bmod n, a \in \mathbb{Z}_n^*$$

  For this Puz.Gen randomly computes two large and distinct primes p, q with $pq = n$. This gives us the group $\mathbb{Z}_n^*$ and the group of quadratic residues $QR_n = \{x^2 \bmod n : x \in \mathbb{Z}_n^*\}$. The quadratic residues are a multiplicative subgroup of $\mathbb{Z}_n^*$ which has cardinality $\phi(n)/4$. This allows Puz.Gen to efficiently compute b. It first computes $e = 2^t \bmod \phi(n)$ and then computes $b = a^e \bmod n$. The resulting puzzle is Z = (C_k, C_s, a, t, n) why does this work

not introduced

- Puz.Solv(Z) computes b by repeatedly squaring a modulo n for a total of t times. It can now compute key $k = C_k - b$ and decrypt message s.

Property analysis: We now want to assert that these Algorithms constitute a time-lock puzzle. We first remark that group properties guarantee that the different computations of b in Puz.Gen and Puz.Solv result in the same number. As such for every well formed puzzle produced by Puz.Gen, Puz.solv will find the correct b. This allows for the correct decryption of s, so this time-lock puzzle fulfills completeness.

Secondly we require a difficulty gap between Puz.Gen and Puz.Solv. Knowing the primes p and q allows for the above mentioned shortcut. Therefore Puz.Gen takes $O(log\ t)$ multiplications to compute e and at most $O(log\ \phi(n))$ multiplications to compute b from there. Puz.Solv and any adversary do not have access to primes p and q and must perform at least $O(t)$ multiplications. It is easy to see that this holds for the above mentioned algorithm Puz.Solv, so we need to show that no poly-size adversary can do better.

Blum, Blum and Shub have proven in [7] that the $x^2 \bmod n$ sequence can be used as an unpredictable pseudo random number generator, relying on the assumption that the quadratic residuosity problem is hard to solve. Assuming a high enough order of a any attempt to predict or outright guess b can be seen as an attempt to predict the random sequence of the $x^2 \bmod n$ generator and will therefore only succeed with negligible probability. How to find an a with large order has also been discussed in [7].

Since prediction does not work, this leaves finding the prime factors of n to make use of the shortcut during solving. At this point in time factoring is still hard and will lead to faster running times with negligible probability. But with a future rise in quantum computing this last attack vector will be exploitable.

B. Random Oracle Model

This section concerns a time-lock puzzle construction in the random oracle model. We will first introduce the puzzle and prove it's security. Afterwards there will be an additional discussion on the space requirements of the puzzle and the existence of strong time-lock puzzles in the random oracle model.

This time-lock puzzle relies on the inherent randomness of a random oracle. It is a slight variation on the scheme proposed in [4] but uses the same underlying principle. We can hide a secret k_t by xor-ing it with the response of the random oracle R to another query k_{t-1}. We can now hide k_{t-1} behind another query k_{t-2}. Continuing in this way we get a series of hidden keys that will make up the time-lock puzzle with k_0 given in plain.

For this the query and response should be of the same bitwise length. We can assume w.o.l.g that we can query the oracle with any length keys. If the keys are to long we can split them into subkeys, query the oracle on them and concatenate the responses into one long response again. Should the keys be to short, we can add a prefix to them, query with the longer key and then truncate the response to fit the original key's length.

Algorithm description:

- Puz.Gen(s, t) uses a secure encryption algorithm to encrypt s with a long key k_t which gives the ciphertext C_s. It then generates t more keys $k_0, \ldots, k_{t-1}$ and queries R on every key. Now the result of query i is xor-ed with the (i+1)th key resulting in the hidden key $x_{i+1} = k_{i+1} \oplus R(k_i)$. The puzzle output will be Z = (C_s, k_0, x_1, ..., x_t).
- Puz.Solv(Z) takes key k_i queries the oracle and computes the xor of the hidden key x_{i+1} with the response $R(k_i)$.

$$x_{i+1} \oplus R(k_i) = k_{i+1} \oplus R(k_i) \oplus R(k_i) = k_{i+1}$$

  This yields the next plain key. Starting with k_0 and repeating this step t times gives k_t which Puz.Solv can use to decrypt C_s and return s.

Property analysis: It is easy to see that Puz.Solv will find the correct message s if it is supplied with a well formed puzzle generated by Puz.Gen. In terms of complexity both Puz.Gen and Puz.solv make t queries to the oracle but Puz.Gen can make these queries in parallel, while Puz.Solv can only make them sequentially. This means that we have a linear difficulty gap between Puz.Gen and Puz.Solv, so this is weak time-lock puzzle.

explain more abt this

The only thing remaining to show is that any poly-size adversary will also take t sequential rounds of queries to get k_t with non-negligible probability. Since we measure the complexity of time-lock puzzles according to the number of queries used the poly-size adversary will be able to make sequential rounds of polynomialy many parallel queries to R. We prove security of the time-lock puzzle as in [4].

We claim that finding k_i with non-negligible probability requires that the adversary knows $R(k_{i-1})$. Simply guessing will have negligible probability if the length of the keys is chosen long enough. And any query, not counting k_{i-1}, to R will not give any information about $R(k_{i-1})$. Therefore the probability of finding k_i, without knowing k_{i-1} to query R and get $R(k_{i-1})$, is negligible. We now consider an adversary doing multiple sequential rounds of parallel queries. Using the above claim we can use induction to see that after round i the adversary can know the keys k_j, $j \leq i$ with non-negligible probability and any other keys k_l only with negligible probability. As s is hidden behind key k_t the adversary will require at least t sequential rounds of queries.

Additional discussion: One disadvantage of this weak time-lock puzzle is that the description of the time-lock puzzle directly correlates to t, therefore it has a huge space requirement and communicating the time-lock puzzle could take as long if not longer than actually solving it. To counteract this Mahmoody et al. proposed a solution to increase the computation to communication ratio. Every oracle query $R(q)$ is replaced by d recursive queries, so $R^1(q) = R(q)$, $R^d(q) = R(R^{d-1}(q))$ for $d > 1$. With this the computation to communication ratio is raised to d. This modified time-lock puzzle will still be secure as akin to the security proof the probability of an adversary finding $R^d(q)$ without knowing $R^{d-1}(q)$ will be negligible.

Since weak time-lock puzzles only give us a linear difficulty gap in parallel time, finding a strong time-lock puzzle in the random oracle model would be great. But this is in fact impossible. In [4] they proved that for every puzzle constructed from n queries and an honest solver that successfully solves the puzzle with probability $1-v$ while making m queries there exists an adversary that arrives at the solution with probability $1 - v - \varepsilon$, makes $O(\frac{nm}{\varepsilon})$ queries in total and takes $O(\frac{n}{\varepsilon})$ sequential rounds of parallel queries.

To prove this we closely follow the proof used by Mahmoody et al. and elaborate on it in some parts. We take into consideration an adversary that will have unbounded computational power, as we are measuring the complexity of time-lock puzzles according to the number of sequential queries made. The main goal of our adversary is to guess the queries made by Puz.Gen during construction, as if we have those we can query all of them in parallel and solve the time-lock puzzle very fast.

In the following we will call the queries made by Puz.Gen critical queries. Our adversary takes multiple rounds of queries and works like follows. In every round it runs the honest solver on a private oracle P and remembers all queries made. The adversary then asks oracle R all queries in parallel and updates P to answer according to R. So P answers queries that have been asked before correctly and any others completely at random. The adversary does this for $\frac{n}{\varepsilon}$ rounds and outputs the secret obtained in the last round.

Intuitively this works because the honest solver can not gain much information from a non-critical query as R answers completely randomly. Thus in those cases the private oracle that also answers completely randomly does about as good a job as R. We now bring a more formal proof and introduce some notation.

$Pr[Success]$ denotes the probability that the adversary obtains the correct secret s. C_k denotes the event that k different critical queries were made during all runs of the simulated solver. We also define the event $Good_i$ that the honest solver did not ask any new critical queries in round i and call round i good. We observe that there can be at most n rounds that are not good as Puz.Gen uses n queries to construct the puzzle. Additionally the responses of oracles R and P will be equally distributed in a good round, as P knows all critical queries for this round and answers any others as randomly as R. This means the honest solver has equal chance to correctly solve the Puzzle with both oracles in a good round. We now compute the probability of the event $Good_i$ and substitute $\frac{n}{\varepsilon}$ by e in the first half for readability.

$$
\begin{aligned}
Pr[Good_i] &= \sum_{k=0}^{n} Pr[C_k] \cdot \binom{e-1}{k} \cdot \binom{e}{k}^{-1} \\
&= \sum_{k=0}^{n} Pr[C_k] \cdot \frac{(e-1)! \cdot k! \cdot (e-k)!}{k! \cdot (e-1-k)! \cdot e!} \\
&= \sum_{k=0}^{n} Pr[C_k] \cdot \frac{e-k}{e} \\
&= \sum_{k=0}^{n} Pr[C_k] \cdot (1 - \frac{\varepsilon \cdot k}{n}) \\
&\geq (1-\varepsilon) \cdot \sum_{k=0}^{n} Pr[C_k] = (1-\varepsilon)
\end{aligned}
$$

We are now ready to put all of this together and prove the above statement. Lastly we define $Pr[Puz.Solv^O]$ to be the probability that the honest solver is correct when using Oracle O and P will denote the private oracle used in the final round of simulation.

$$
\begin{aligned}
Pr[Success] &\geq Pr[Success \wedge Good_{\frac{n}{\varepsilon}}] \\
&= Pr[Puz.Solv^P \wedge Good_{\frac{n}{\varepsilon}}] \\
&= Pr[Puz.Solv^R \wedge Good_{\frac{n}{\varepsilon}}]
\end{aligned}
$$

$$\geq Pr[Puz.Solv^R] - (1 - Pr[Good_{\frac{n}{\varepsilon}}])$$
$$= 1 - v - \varepsilon$$

This adversary shows that we could not expect anything better than a linear gap between generating and solving a puzzle. Mahmoody et al. also proof a stronger theorem that there exists an adversary that makes more queries in total but only tales n sequential rounds of queries. This shows that the above time-lock puzzle is the best possible we could hope for in terms of difficulty gap in the random oracle model.

From these findings in the random oracle model we can conclude that strong time-lock puzzles based on one-way permutations or collision-resistant hash functions, as stand ins for the random oracle, will be impossible in the real world.

C. Randomized Encoding

For the last time-lock puzzles of this paper we first need to introduce randomized encodings and non-parallelizing languages. We then show how to construct a strong time-lock puzzle and prove it's security. Lastly we shortly discuss a weak and intermediate version of the introduced time-lock puzzle.

Randomized encodings can be used to hide a computation given by a function f or a Turing machine M behind another representation $\widehat{f}$ or $\widehat{M}$. The result y of the hidden computation is encoded in the distribution of $\widehat{f}$ or $\widehat{M}$. Furthermore they guarantee that randomized encodings of different computations with the same result will be indistinguishable from one another. So the underlying computation and input x are completely hidden. This property is defined with the help of a simulator that, given the result y, outputs a simulated randomized encoding $\widehat{S}_y$ that cannot be distinguished from an actual randomized encoding with any noticeable advantage. In the following section we will limit ourselves to randomized encodings of Turing machines.

explain more

We call a randomized Encoding succinct if the encoding time is virtually independent of the running time of the given Turing machine. The following is a formal definition of succinct randomized encodings as found in [5].

Definition 2 (Succinct Randomized Encoding): A succinct randomized encoding scheme RE consists of two algorithms (RE.Enc, RE.Dec) satisfying the following requirements:

- **Syntax**:
  - $\widehat{M}(x) \leftarrow \text{RE.Enc}(M, x, t, 1^\lambda)$ is a probabilistic algorithm that takes as input a machine M, input x, time bound t and a security parameter 1^λ. The algorithm outputs a randomized Encoding $\widehat{M}$(x).
  - $y \leftarrow \text{Re.Dec}(\widehat{M}(\text{x}))$ is a deterministic algorithm that takes as input a randomized encoding $\widehat{M}$(x) and computes an output $y \in \{0,1\}^\lambda$.
- **Completeness**: for every input x and machine M such that, on input x, M halts in t steps and produces a λ-bit output, it holds that $y = M(x)$ with overwhelming probability over the coins of RE.Enc.
- **Security**: there exists a probabilistic polytime (PPT) simulator Sim satisfying: for any poly-size distinguisher $D = \{D_\lambda\}_{\lambda\in\mathbb{N}}$ and polynomials $m = m(\cdot), n = n(\cdot), t = t(\cdot)$, there exists a negligible $\mu(\cdot)$, such that for any $\lambda \in \mathbb{N}$, machine $M \in \{0,1\}^m$, input $x \in \{0,1\}^n$:
$$\left|Pr[D_\lambda(\widehat{M}(x)) = 1 : \widehat{M}(x) \leftarrow \text{RE.Enc}(M, x, t, 1^\lambda)] - Pr[D_\lambda(\widehat{S}_y) = 1 : \widehat{S}_y \leftarrow \text{Sim}(y, 1^m, 1^n, t, 1^\lambda)]\right| \leq \mu(\lambda)$$
where y is the output of $M(x)$ after t steps.
- **Efficiency**: For any machine M that on input x produces a λ-bit output in t steps:
  - $\text{RE.Enc}(M, x, t, 1^\lambda)$ can be computed in sequential time $\text{polylog}(t) \cdot \text{poly}(|M|, |x|, \lambda)$.
  - $\text{RE.Dec}(\widehat{M}(x))$ can be computed in sequential time $t \cdot \text{poly}(|M|, |x|, \lambda)$.

Normal randomized encoding schemes are known based on the existence of one-way functions and an example of such a scheme that works for any Turing machine in $P/poly$ is given by Yao's garbled circuits [8]. But non-succinct randomized encodings do not suffice to construct strong time-lock puzzles. For these succinct randomized encoding schemes are necessary. One such scheme that satisfies all formal requirements was proposed by Bitansky et al. in [9]. It assumes the existence of one-way functions and indistinguishability obfuscation for all circuits.

We now define what it means for a language to be non-parallelizing. The language is decidable in polytime using sequential computation and no parallel algorithm can do significantly better.

Definition 3 (Non-Parallelizing Languages): We call a Language L non-parallelizing if $L \in Dtime(t(\cdot))$ if every family of non-uniform polysize circuits $D = \{D_\lambda\}_{\lambda\in\mathbb{N}}$ with $dep(D_\lambda) < t(\lambda)$ fails to recognize the language $L_\lambda = L \cap \{0,1\}^\lambda$ for large enough λ.

We can now give another time-lock puzzle construction with the help of randomized encodings, assuming that non-parallelizing languages exist for every polynomially bounded function $t(\cdot)$. This time-lock puzzle was first introduced in [5]. The idea behind this puzzle is that we can define a Turing machine that waits for $t = t(\lambda)$ steps and then outputs the secret s. Since randomized encodings hide all information about the computation, a randomized encoding of this machine will also hide secret s. Security is given by the fact that any adversary of depth less than $t(\cdot)$ that can differentiate time-lock puzzles with different secrets, can be used to construct a decider with depth less than $t(\cdot)$ for non-parallelizing languages in $Dtime(t(\cdot))$.This is a contradiction, as we assume the existence of such languages. Therefore the only possibility is that such an adversary cannot exist.

Algorithm:

- Puz.Gen(s, t) interprets secret s as a bit-string of length λ and makes sure that $t \leq 2^\lambda$, if not it may pad s. It now constructs a Turing machine M_s^t that, for every input x, outputs s after t steps. Puz.Gen now invokes a succinct randomized encoding scheme $\text{RE.Enc}(M_s^t, 0^\lambda, t, 1^\lambda)$ and outputs the generated encoding as puzzle Z = $\widehat{M}_s^t(0^\lambda)$.

- Puz.Solv simply invokes RE.Dec(Z) = RE.Dec($\widehat{M}_s^t(0^\lambda)$) = s and outputs the resulting secret s.

An important fact is, that for large enough λ we may assume that the Turing machine constructed by Puz.Gen is described by 3λ bits.

Properties: Completeness and efficiency follow from the definition of a succinct randomized encoding. From this we also see that this is a strong time-lock puzzle. We are now going to give an overview of the security proof and refer to [5] for a more rigorous proof. As stated above we will use an adversary of depth less than t to construct a quick parallel decider for a t-non-parallelizing language. Since such a decider cannot exist, such an adversary can also not exist.

Let $q_{RE}(\lambda)$ be the efficiency polynomial from the succinct randomized encoding scheme. We now assume that there exists a polysize adversary $A = (A_\lambda)_{\lambda\in\mathbb{N}}$ with $dep(A_\lambda) < t(\cdot)$ for some polynomial $t(\lambda) > q_{RE}(\lambda)$. This adversary will be able to differentiate the secrets of two different time-lock puzzles with difficulty $t(\cdot)$ with a certain advantage. So for a polynomial $p(\cdot)$ and for infinitely many $\lambda \in \mathbb{N}$ there exists a pair of secrets s_0, s_1 such that for $b \in \{0, 1\}$:

$$Pr[A_\lambda(Z) = s_b : Z \leftarrow \text{Puz.Gen}(t(\lambda), s_b)] \geq \frac{1}{2} + \frac{1}{p(\lambda)}$$

We now construct a Turing machine $M^{L,t}_{s_0,s_1}$ which decides the non-parallelizing language $L \in Dtime(t(\cdot))$. For input $x \in \{0,1\}^\lambda$ it outputs s_1 if $x \in L$ and s_0 if $x \notin L$ after $t(\lambda)$ steps. We may assume for large enough λ that $M^{L,t}_{s_0,s_1}$ is described by 3λ bits. We notice that for a fixed x this machine has the same size, same running time and output as the machine $M^t_{s_b}$ of one of the time-lock puzzles differentiated by the adversary. In the following we will define $b := 1$ if $x \in L$ and $b := 0$ if $x \notin L$.

We now construct a probabilistic decider for L that has a certain noticeable advantage and later argue how it can be turned into a deterministic decider. For this the decider generates a succinct randomized encoding of $M^{L,t}_{s_0,s_1}(x)$ and invokes the adversary to decide if x is in L. Since the security of succinct randomized encodings guarantees that the encoding of a machine cannot be differentiated from the encoding generated by a simulator we see that $\widehat{M}^{L,t}_{s_0,s_1}(x)$ and $\widehat{S}_{s_b}$ will be indistinguishable and similarly $\widehat{S}_{s_b}$ and $\widehat{M}^t_{s_b}$.

The probability that the decider is correct now directly depends on the probability that the adversary is correct minus the negligible $\mu(\lambda)$ that the randomized encodings can actually be differentiated by the adversary. But since $\mu(\lambda)$ is negligible we find that the decider has a noticeable advantage just like the adversary.

Now that we have a probabilistic decider with a certain advantage we can use a parallel repetition argument to attain a deterministic decider. The idea behind this is that we can run polynomialy many instances of the probabilistic decider in parallel and take the outcome produced by the majority of instances. If enough parallel instances are run the probability of the majority of instances being incorrect will become negligibly small. All in all this will only require a polynomial number of parallel instances. This way the deterministic decider will be correct with overwhelming probability.

In this overview we have not shown that this deterministic decider fulfills depth and run time requirements, but it can be constructed to meet these requirements. The complete proof in [5] gives more information on this.

Additional time-lock puzzles: We now discuss the weak and intermediate time-lock puzzle constructions proposed in [5]. The weak version which is based on normal randomized encodings, this means that encoding takes time proportional to the running time of the Turing machine. But this construction is then only based on the existence of one-way functions and is highly parallelizable during creation.

The construction is the same as for the strong time-lock puzzle above and just uses a non-succinct randomized encoding scheme. The intermediate time-lock puzzle uses preprocessing to generate a reusable randomized encoding. This phase takes time t to complete and can also be parallelized. Afterwards any number of time-lock puzzles can be generated independent of the difficulty parameter t. For more information on the construction of the intermediate time-lock puzzle we again refer to [5]. Constructions for reusable randomized encodings are again achieved through reusable garbled circuits. Constructions for these are known based on the sub-exponential LWE assumption and have been proposed Goldwasser et al. in [10]. Agrawal proposed a similar construction based on the previous to achieve higher security [11].

As Bitansky et a. pointed out themseves, one problem with both of these schemes is that the description of the time-lock puzzle is of size t, so the difficulty of the puzzle. This means these time-lock puzzles only make sense in a setting where communication is cheaper than computation.

We also do not see any way to increase the computation to communication ratio as we did for the time-lock puzzle in the random oracle model, as the size here is not proportional to some amount of keys we send but the running time of the encoded Turing machine. We cannot change this running time without also changing the length of the randomized encoding and vice versa.

IV. Evaluation

Now that we have introduced all of the time-lock puzzles we are going to compare and discuss them. We will start by mentioning that while strong time-lock puzzles seem superior to weak time-lock puzzles in almost every way, weak puzzles that are based on less or more well founded assumptions than their stronger counter part would still be a great addition as another option of time-lock puzzle.

This is what we have seen in the case of the randomized encoding time-lock puzzles. But here the drawback of having a time-lock puzzle of size proportional to t makes these particular weak time-lock puzzles impractical under current conditions where compuation is much cheaper than communication. Just sending the time-lock puzzle to another party takes as long as solving the puzzle, therefore the premise of hiding

a secret s for time t is already accomplished after the puzzle has been fully sent, defeating their purpose of publishing a secret that can only be retrieved at a later point in time.

Here the weak time-lock puzzle proposed in the random oracle is much more competitive to strong time-lock puzzles concerning space requirements of the puzzle. It loses to the strong time-lock puzzles in terms of efficiency, especially since the recursive oracle queries made to heighten the computation to communication ratio d cannot be parallelized. And it's security has not been considered in the normal cryptographic model. Even so we think that this is a time-lock puzzle that, if proven secure in the normal model, could be a contender to be used in protocols or other applications where shorter encryption periods are expected. Especially it's simplicity should make implementation of the generator and solver very robust, as the only complicated part will be how to instantiate an oracle.

Comparing the strong puzzles to one another is a lot more nuanced. Both time-lock puzzles are currently based on different hardness assumptions which is very good, since there would be at least two different schemes that can be relied upon even if the assumptions of one will be disproven in the future.

Both puzzles are also not secure in a post-quantum era of computing. Rivest, Shamir and Wagner's construction does not hold up as factoring of numbers will become easy and adversaries could use the same shortcut that the puzzle generator uses. This would effectively make it another weak time-lock puzzle. The puzzle based on randomized encodings relies on indistinguishability obfuscation which is currently either built on not well-founded assumptions that are much more likely to be disproven in the future or in the case of [6] based on well founded assumptions of which not all are quantum save. This means that indistinguishability would be broken rendering randomized encodings useless and breaking the time-lock puzzle.

It should also be mentioned that even though non-parallelizing languages have not been proven to exist, this does not reflect negatively on the randomized encoding time-lock puzzle. The reason being, that non-parallelizing languages are a prerequisit for the existence of time-lock puzzles. If no non-parallelizing language exists that would mean that any problem can be solved more efficiently through the use of parallelism, making time-lock puzzles as defined so far impossible. So if any time-lock puzzle exists, then the construction based on randomized encodings will also constitute a time-lock puzzle.

The time-lock puzzle based on repeated squaring has been known for far longer, so more schemes have been proposed using it as a basis to offer more additional attributes like non-maleability or homomorphic time-lock puzzles. In this aspect the strong time-lock puzzle based on randomized encodings falls behind. But with the recent progress on indistinguishability obfuscation this time-lock puzzle will be a great second option and should therefore be considered as a basis for future research into time-lock puzzles.

All in all future work related to protocols or applications making use of time-lock puzzles should be based on the general definition of time-lock puzzles and not on specific facets of a certain puzzle. While time-lock puzzle specific solutions might be more efficient and therefore of some value, we now have multiple candidates for time-lock puzzles that we can swap in and out of such protocols or applications without adversely affecting their security. This will make such applications of time-lock puzzles much more secure as they will not rely on specific hardness assumptions. It will also mean that any time-lock puzzle found in the future will have many useful applications at once.

V. Discussion

We have decided to focus our attention on the time-lock puzzle side of timed release cryptography. That is because we believe that the side using trusted third parties like agents or trusted servers has seen much more progress so far as compared to time-lock puzzles. Until recently only one candidate puzzle based on well founded assumptions was known. Therefore we want to give people an easier introduction to the state of the currently existing time-lock puzzles to further research in this field. That is also why not all theorems or security proofs have been rigorously argued. This paper serves as a starting point to introduce the concepts that the time-lock puzzles are based on and to give intuition about their security. Formal and rigorous proofs for those more interested can be found in the referenced papers instead.

Many variants of timed-release encryption depending on third parties exist with different basis and different security guarantees. One we want to highlight here is introduced in [12] and guarantees scalalbe, server-passive, user-anonymous time release cryptography. This means no direct communication between client and serer has to occur for encryption and decryption. But the server only publishes periodic time-stamps which can be used for decryption. It can also be used in conjunction with identity based encryption making it impossible for the server to gain early access to encrypted messages.

We see two main problems with this and all other third party protocols. First trusted servers or agents could be forced to reveal their secret keys by corrupt governments or through bribery. Secondly since the time server and as such the party running the time server has access to all future time-keys one could not encrypt announcements meant for everyone with these methods as the time-server has instant access to the encryption key, and identity based encryption in this case does not help out.

In our view timed release cryptography is useful for hiding messages with very long encryption periods, so multiple years for example. Here time-lock puzzles lack both in accuracy of the encryption period and would take an incomparable amount of computing time and power when compared to the trusted third party solution. But for highly sensitive protocols or applications like key escrow where trust in third parties might not be acceptable and encryption periods are short, so at most a few weeks or months, time-lock puzzles seem a much better option than using third parties. They also allow

for public announcements to be encrypted while not giving any recipient an advantage over others. Seeing as we primarily want to motivate the research into protocols using timed release encryption we decided to focus on time-lock puzzles, as they are a less researched field and have advantages over third party approaches regarding sensitive protocols.

VI. Related Work

We start by introducing recent research by Baum et al. [13] in which they highlight that time-lock puzzles have been used in many protocols in composition with other primitives. But time-lock puzzles have not been proven secure in the universal composability (UC) model. This means that using time-lock puzzles under composition could hamper their security. That is why they propose a foundation to analyze and prove the security of composite protocols using time-lock puzzles in the UC-model. Furthermore they adapt Rivest, Shamir and Wagner's time-lock puzzle to work in their new environment and prove it's security. Finally they prove that random oracles are necessary to construct UC-secure time-lock puzzles.

Malavolta and Thyagarajan introduce a fully homomorphic time-lock puzzle in [14]. They reason that proposed applications like e-voting or multi-party coin flipping using time-lock puzzles may end up with a great number of time-lock puzzles that need to be solved to attain their secrets and compute some function with them. Since all time-lock puzzles would take a long time to solve the overall running time to attain all secrets would be astronomical. Therefore Malavolta and Thyagarajan introduce homomorphic time-lock puzzles to help with this problem. Here any number of puzzles are combined into one puzzle with the solution being the desired function evaluated over all secrets. This homomorphic time-lock puzzle is again based on the construction proposed by Rivest, Shamir and Wagner. They first introduce linear homomrphic and multiplicative homomorphic time-lock puzzles based on the attributes of groups used in the basic time-lock puzzle and go on to construct fully homomorphic time-lock puzzles relying on indistinguishability obfuscation.

Another scheme that adds useful attributes to time-lock puzzles was proposed by Freitag et. al. in [15]. They raise concern for mauling attacks where a man in the middle attacker recieves the puzzle of a party participating in a protocol and tries to send out a puzzle with a slightly altered secret. They construct one non-malleable time-lock puzzle proven secure in the auxiliary random oracle model that does not rely on any other assumptions. They also give another construction proven secure in the plain model assuming existence of a keyless multi-collision resistant hash function, a non-interactive witness indistingushalbe proof for NP and injective one-way functions. A key fact to highlight is that both constructions will work with any time-lock puzzle. They then go on to achieve fair multi-party protocols for coin flipping in both the plain model and random oracle model using non-malleable time-lock puzzles.

Timed commitments are a protocol based on the time-lock puzzle based on repeated squaring in finite fields. It was proposed in [3] and uses the time-lock puzzle to add a potential forced opening phase to the standard commitment scheme. They take into account that party B can verify in the commitment phase that party A has committed a valid and solvable time-lock puzzle. This is done with help of zero knowledge proofs. After this party A can send additional information which makes solving the time-lock puzzle trivial and fast. But should party A not send anything B can invest the time to solve the time-lock puzzle and still retrieve the committed value. This scheme heavily relies on the use of the underlying time-lock puzzle which cannot be switched out, none the less they use these timed commitments to construct a fair protocol for contract signing.

Encapsulated key escrow [16] is an escrow scheme that can make any kind of time-lock puzzle verifiable and works with keys of any cryptographic schemes. Furthermore it avoids early recovery, which means that the computation to attain the key can only be done after the someone has gotten the escrowed part of the key.

We see many improvements that add useful attributes like non-malleability or verification to time-lock puzzles and some applications of time-lock puzzles in protocols. But many of them rely on the first time-lock puzzle of repeated squaring in finite fields. We hope to give new researches to this field an overview of the currently available time-lock puzzles and motivate more protocols and applications relying only on the general notion of time-lock puzzles.

VII. Conclusion

We think time-lock puzzles are a very interesting cryptographic tool that could be used in the construction of new protocols or even non-crpytographic applications. They seem especially useful under circumstances like the ones in timed commitments where you would want to guarantee the attainment of some secret while not giving it up right away. As this is something where all of the qualities of a time-lock puzzle are needed.

Now that we have schemes for multiple time-lock puzzles based on different well-founded assumptions we think that the time has come to earnestly review in what ways time-lock puzzles can be used to invent new protocols or refine and upgrade already existing protocols. We surmise that this timed primitive has much untapped potential and is well worth it to study.

For further research in this field we would propose more additional qualities for time-lock puzzle in general as well as research into the existence of non-parallelizing languages as they are vital for the existence of all time-lock puzzles. Additionally more time-lock puzzles based on different assumptions are also of high interest. Even weak or intermediate time-lock puzzles that have space-requirements that are not proportional to t would be a big achievement in diversifying our current toolbox of puzzles, since some will be better suited for some applications than others and time-lock puzzles will be based on a wider variety of well founded assumptions.

REFERENCES

[1] T. May, "Timed-release crypto," *http://www. hks. net. cpunks/cpunks-0/1560. html*, 1992.

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

[3] D. Boneh and M. Naor, "Timed commitments," in *Annual international cryptology conference*. Springer, 2000, pp. 236–254.

[4] M. Mahmoody, T. Moran, and S. Vadhan, "Time-lock puzzles in the random oracle model," in *Annual Cryptology Conference*. Springer, 2011, pp. 39–50.

[5] N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters, "Time-lock puzzles from randomized encodings," in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, 2016, pp. 345–356.

[6] A. Jain, H. Lin, and A. Sahai, "Indistinguishability obfuscation from well-founded assumptions," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, 2021, pp. 60–73.

[7] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM Journal on computing*, vol. 15, no. 2, pp. 364–383, 1986.

[8] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 1986, pp. 162–167.

[9] V. Koppula, A. B. Lewko, and B. Waters, "Indistinguishability obfuscation for turing machines with unbounded memory," in *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, 2015, pp. 419–428.

[10] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable garbled circuits and succinct functional encryption," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 555–564.

[11] S. Agrawal, "Stronger security for reusable garbled circuits, general definitions and attacks," in *Annual international cryptology conference*. Springer, 2017, pp. 3–35.

[12] A.-F. Chan and I. F. Blake, "Scalable, server-passive, user-anonymous timed release cryptography," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE, 2005, pp. 504–513.

[13] C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner, "Tardis: a foundation of time-lock puzzles in uc," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 429–459.

[14] G. Malavolta and S. A. K. Thyagarajan, "Homomorphic time-lock puzzles and applications," in *Annual International Cryptology Conference*. Springer, 2019, pp. 620–649.

[15] C. Freitag, I. Komargodski, R. Pass, and N. Sirkin, "Non-malleable time-lock puzzles and applications," in *Theory of Cryptography Conference*. Springer, 2021, pp. 447–479.

[16] M. Bellare and S. Goldwasser, "Encapsulated key escrow," 1996.