On Sequential Functions and Fine-Grained Cryptography

Jiaxin Guan*
New York University
jiaxin@guan.io

Hart Montgomery
Linux Foundation
hart.montgomery@gmail.com

Abstract

A *sequential function* is, informally speaking, a function f for which a massively parallel adversary cannot compute "substantially" faster than an honest user with limited parallel computation power. Sequential functions form the backbone of many primitives that are extensively used in blockchains such as verifiable delay functions (VDFs) and time-lock puzzles. Despite this widespread practical use, there has been little work studying the complexity or theory of sequential functions.

Our main result is a black-box oracle separation between sequential functions and one-way functions: in particular, we show the existence of an oracle $\mathcal{O}$ that implies a sequential function but not a one-way function. This seems surprising since sequential functions are typically constructed from very strong assumptions that imply one-way functions and also since time-lock puzzles are known to imply one-way functions (Bitansky *et al.*, ITCS '16).

We continue our exploration of the theory of sequential functions. We show that, informally speaking, the decisional, worst-case variant of a certain class of sequential function called a *continuous iterative* sequential function (CISF) is PSPACE-complete. A CISF is, in a nutshell, a sequential function f that can be written in the form $f(k, x) = g^k(x)$ for some function g where k is an input determining the number of "rounds" the function is evaluated. We then show that more general forms of sequential functions are not contained in PSPACE relative to a random oracle.

Given these results, we then ask if it is possible to build any interesting cryptographic primitives from sequential functions that are not one-way. It turns out that even if we assume just the existence of a CISF that is not one-way, we can build certain "fine-grained" cryptographic primitives where security is defined similarly to traditional primitives with the exception that it is only guaranteed for some (generally polynomial) amount of time. In particular, we show how to build "fine-grained" symmetric key encryption and "fine-grained" MACs from a CISF. We also show how to build fine-grained public-key encryption from a VDF with a few extra natural properties and indistinguishability obfucsation (iO) for null circuits. We do not assume one-way functions. Finally, we define a primitive that we call a commutative sequential function–essentially a sequential function that can be computed in sequence to get the same output in two different ways–and show that it implies fine-grained key exchange.

This article is the full version of the article under the same name submitted by the authors to the IACR and to Springer-Verlag on May 31, 2024. The version published by Springer-Verlag will be available in the proceedings of CRYPTO 2024.

*Part of this research was conducted while this author was a Ph.D. student at Princeton University.

1 Introduction

Traditional cryptography is focused on security assumptions that assume an adversary's computational budget is limited to some polynomial function of a security parameter, or some concrete number of operations (e.g. 2^{128}). However, some exciting new applications of cryptography such as verifiable delay functions (VDFs) require *sequential* assumptions, which demand that adversaries cannot solve certain problems in less than a specified amount of time, even with substantial parallel computing resources.

Informally, to say that a function is sequential we need to define a model of computation for an adversary, which we call $\mathcal{M}_{\mathcal{A}}$, and a challenger, which we call $\mathcal{M}_{\mathcal{C}}$. In general, we will assume that $\mathcal{M}_{\mathcal{A}}$ has more parallel computation power than $\mathcal{M}_{\mathcal{C}}$. In this work, we will typically assume that $\mathcal{M}_{\mathcal{A}}$ and $\mathcal{M}_{\mathcal{C}}$ are essentially equivalent outside of parallel power. Although this is certainly not the case for many real-world applications where adversaries may have more powerful hardware than "honest" users, in most practical situations, we expect an adversary's parallel advantage to be much greater than their sequential advantage over typical honest users. For example, a supercomputing cluster may be able to perform sequential operations faster than a desktop computer, but its real advantage over the desktop lies in its parallel processing abilities.

We say that a function is $(t_{\mathcal{C}}, t_{\mathcal{A}})$-sequential if a challenger can compute the function in time $t_{\mathcal{C}}$ in the model $\mathcal{M}_{\mathcal{C}}$, while no adversary can compute the function faster than time $t_{\mathcal{A}}$ in the model $\mathcal{M}_{\mathcal{A}}$. We note that it is far from trivial to even define a sequential function, and we use the definitions from [JMRR21] which generalize those of [BBBF18].

Sequential functions are used in a wide variety of practical deployments. For instance, the Solana blockchain uses proof of history in its consensus layer [Yak18], which relies on a verifiable delay function. The Chia network blockchain also relies on a VDF for its proof of space and time consensus protocol [CP18a]. In fact, the Ethereum Foundation and a number of other blockchain entities are pushing towards building practical VDFs in order to better scale [CHI+20] and currently [KMT22] plan on using a VDF based on using the Nova proof system [KST22] as a recursive SNARK. Potentially billions of dollars [eth] will rely on a secure VDF construction in the near future, so it is important that we have confidence in our constructions that rely on sequential assumptions.

Sequential Assumptions. Despite this, almost all of the constructions (that are not based on random oracles) of sequential functions and VDFs with security proofs use essentially the same sort of assumption. The original construction of time-lock puzzles [RSW96] makes the assumption that repeated squaring in a group of unknown order is sequential, and almost all other practical time-delay constructions [BN00, GMPY06, LW17, Wes19, Pie19, DGMV20, Sha19, BDGM19] rely on variants of this assumption or efficient CCA timed commitments in class groups [BBBF18, DMPS19]. Most of these constructions assume that with a description of a group $\mathbb{G}$ that does not include the order, and a generator $g \in \mathbb{G}$, then it takes $\Omega(T)$ time to compute g^{2^T}.

However, this assumption is false: Bernstein and Sorenson [BS07] proved that computing g^{2^T} in a group of unknown order can be parallelized with $T^{1+o(1)}$ processors to a depth of $O(T/\lg\lg T)$, emphasizing that these assumptions could probably use more study. On the other hand, Rotem and Segev [RS20] showed that generically speeding up repeated squaring on *generic* groups is equivalent to factoring, but it is still not known how to tie repeated squaring to any sort of standard model assumption, even with some slack factor (which could be compatible with the attack from [BS07]). We additionally note that [RSS20] showed that delay functions on groups require an unknown order,

indicating that it is likely we will need to completely scrap group-based assumptions of sequentiality if quantum computing becomes viable.

It has long been folklore that the repeated computation of a random oracle [BR93] is sequential, and more recent work has expanded on this idea [MMV13, CP18b]. However, random oracle-based primitives have not typically been useful for building cryptosystems that rely upon sequential assumptions because they lack any kind of structure, which is often useful for things like verifying computations. In fact, Mahmoody *et al.* show that VDFs satisfying perfect uniqueness and tight VDFs are impossible to construct in a black-box way solely from ideal hash functions [MSW20], which may indicate some practical limitations of RO-based constructions.

There are a handful of constructions (mostly focused on time-lock puzzles) that use indistinguishability obfuscation (iO) as a core assumption [BGJ+16, MT19]. However, these constructions typically require subexponential security, and while theoretically appealing, are far from practical. Finally, we note that [JMRR21] construct a concrete sequential function assuming *the existence* of a sequential function and fully homomorphic encryption, but this construction does not fundamentally help us to understand what assumptions are required for sequential functions.

The Complexity of Sequential Functions. Given the practical importance of sequential functions, we would like to understand more about sequential assumptions. This is especially true given that all of the known assumptions that imply sequentiality are either idealized (generic group or random oracle) or very impractical (iO with subexponential security). Can we characterize sequential functions from a cryptographic perspective?

We note that parallel complexity has been an important topic in complexity theory [AB09]. However, complexity theorists have typically not spent time on parallel *average-case* complexity, which is what would be necessary for sequential assumptions. Some notable exceptions include Applebaum, Ishaii, and Kushilevitz [AIK06], who study cryptography in NC^0, Alwen and Serbinenko, who study average-case parallel complexity with respect to memory hardness [AS15], and Bitansky *et al.*[BGJ+16], who study time-locked puzzles. Notably, the authors of [BGJ+16] show that time-lock puzzles imply one-way functions.

On the other hand, fine-grained complexity [Bri19]-on which practical sequential functions critically rely[1]–has been extensively studied recently in the theory community [WW10, AW14, AWY15, ABBK17]. However, it has received considerably less attention recently in the cryptographic community, and there are a comparatively smaller number of works on the subject [DVV16, BRSV17, CG18, LLW19, EWT21, WPC21, WP22, BC22, ACM22].

Ultimately we would like to be able to tie sequential functions to standard cryptographic assumptions. Unfortunately, this seems difficult: it is easy to define a one-way function that is *not* a sequential function, and, in practice, most attacks on cryptographic primitives are parallelizable. It seems that sequential assumptions are, in fact, more powerful than standard cryptographic assumptions. Could we, for instance, show that a sequential function implies a one-way function?

Constructions from Sequential Functions. In addition to the major applications listed above, there are a number of interesting and potentially practical constructions that rely on sequential functions, including modern protocols that have blockchain applications like delay encryption [BD21] and short-lived zero knowledge proofs and signatures [ABC22] as well as older things like time-locked

[1] If the reason why isn't clear now, hopefully our formal definitions of sequential functions in Section 4 will make it clear.

commitments [BN00]. Is it possible to build these kinds of constructions on simpler assumptions, or only rely on sequentiality (and not one-wayness)?

1.1 Our Contributions

In this paper, we show surprisingly that sequential functions do not imply one-way functions in a black-box way. Despite this fact, we show that it is still possible to build some fine-grained cryptographic primitives. In particular, we show the following results:

An Oracle Separation. Our main result is an oracle separation between one-way functions and sequential functions.[1] More precisely, we show the existence of an oracle that implies a sequential function but does not imply a one-way function. Our construction and proof techniques here borrow from things as disparate as the Davies-Meyer construction and traditional black-box separations [IR89, BM09]. This is a technically involved construction and we present a full overview of it in the next section, so we will omit further description for now.

Worst-Case, Decisional Continuous Iterable Sequential Functions (CISFs) are PSPACE-Complete. A CISF is, roughly speaking, a sequential function that is defined by a "round" function. This round function takes polynomial time and has identical input and output domains. A CISF can (approximately) be written in the form $y = \mathsf{Round}^k(x)$ for input x, output y, and k rounds of evaluation. We formally define a CISF in Section 4. We define a class of functions that encapsulates natural worst-case, decisional versions of a CISF and show that this class is PSPACE-complete. This implies that "worst-case" sequential functions are seemingly in a different complexity class than "worst-case" one-way functions, which would be in NP.

More General Sequential Functions Are Not in PSPACE. In contrast to the previous result, we next show that a more general kind of sequential function called a *dynamic sequential function (DSF)* is not in PSPACE relative to a random oracle. In particular, we give an example of a DSF that is not evaluatable in PSPACE assuming the existence of a random oracle. Our proof technique relies on a technique by Dwork, Naor, and Wee [DNW05] that converts a graph to a function in the random oracle model, while preserving space and time lower bounds. We note that this contrasts with our earlier result on CISFs, as it indicates that CISFs could have a completely different complexity than general sequential functions.

Cryptographic Implications. Our work here seems to imply that sequential functions and one-way functions have a more complicated relationship than a strictly hierarchical one: sequential functions do not appear to be strictly stronger than OWFs. Unfortunately, this also means that it may be difficult to build cryptoprimitives with sequential functionalities from standard assumptions. Our results also beg the following question: what sort of cryptographic primitives can we build based on functions that are sequential but not one-way? Since sequential functions may not be incompatible with Pessiland [Imp95], the implications of any cryptoprimitives from sequential but not one-way functions might be quite interesting.

It turns out we can build a number of "fine-grained" primitives from certain types of sequential functions where we only assume the function is sequential (and not one-way). By fine-grained, we

[1] We technically separate one-way functions and CISFs, which we explain shortly.

(informally) mean that security only holds for a certain (polynomial) amount of time against all PPT adversaries, but after this time period has elapsed, there are no security guarantees.[1] These include the following:

Symmetric-Key Encryption and MACs from Continuous Iterative Sequential Functions (CISFs). We first show that CISFs imply fine-grained basic symmetric-key cryptoprimitives. Intuitively, these constructions are similar to their traditional cryptographic counterparts with the exception of the fact they only remain secure for some polynomial amount of time. More precisely, we show how to build fine-grained symmetric-key encryption and MACs from CISFs. The constructions are limited in the sense that users can only encrypt or MAC an apriori-determined number of messages, but only require a shared secret that is of size independent of this number of messages.

Public-Key Encryption from VDFs and Null iO. We next show how to build fine-grained PKE from VDFs with some extra natural properties and iO for null circuits. In particular, we need the VDF we use in the construction to have *unique accepting proofs* and a property that we call *proof indistinguishability*, which means, informally speaking, that the VDF proofs look indistinguishable from random for an adversary that has not computed the final output of the VDF. These extra requirements are not necessarily unreasonable in practice: for instance, the proof of the VDF in [Wes19] is a single group element for which it seems very plausible (i.e. under what appear to be reasonable security assumptions) to achieve these properties, although of course this VDF immediately implies a one-way function.

However, we note that VDFs are not known to imply PKE or even one-way functions[2] and even "full" iO without the existence of OWFs does not appear to be a very powerful primitive[3]. This result intimates that VDFs might be the "fine-grained public-key" equivalent of basic sequential functions, at least in some sense.

Commutative Sequential Functions and Key Exchange. Finally, we show that fine-grained key exchange can be built from what we define as a "commutative" sequential function. Informally speaking, a commutative sequential function allows two parties, Alice and Bob, to each compute a part of a sequential function, send it to the other party, and let the other party "finish" the computation, while any eavesdropper Eve must start from the beginning.

This allows us to more closely see the relationship between sequential functions and key exchange. Our definitions here are general enough to encapsulate any sort of key exchange protocol, and we emphasize that traditional key exchange protocols can be viewed as extremely strong fine-grained key exchange protocols and also, generally speaking, imply commutative sequential functions. Unfortunately, we know of no fine-grained key exchange protocols that are built from assumptions that do not already imply key exchange, so we leave the problem of finding these functions (or ruling them out) as interesting future work.

[1] We note that this definition does also imply security against an adversary with a *fixed* computational runtime, which is a more traditional way of viewing fine-grained assumptions, assuming parameters are set correctly; we discuss the nuances of our definitions in more detail later.

[2] Although [BGJ+16] showed that time-lock puzzles imply one-way functions, it was pointed out in [JMRR21] that it remains an open problem to show whether or not a VDF implies a time-lock puzzle.

[3] In fact, if $\mathsf{P} = \mathsf{NP}$, then iO exists trivially.

1.2 Paper Outline

The rest of the paper proceeds as follows. In section 2, we give an overview of our main result showing an oracle separation between a sequential function and a one-way function. Then, in section 3, we give an overview of our other results and proof techniques in the paper. The paper proper begins in section 4, where we define preliminary material and basic notation.

In section 5, we show our main technical construction: an oracle that implies a sequential function but not a one-way function. Then in section 6, where we prove that (informally speaking) worst-case, decisional CISFs are PSPACE-complete but DSFs are not contained in PSPACE in the random oracle model.

We next move to our constructions of fine-grained cryptoprimitives. We start by defining and showing how to build fine-grained symmetric-key encryption in section 7 and MACs in section 8, both directly from CISFs. In section 9, we define and show how to build fine-grained PKE from VDFs and iO. Finally, in section 10, we define commutative sequential functions and show that they are equivalent to fine-grained key exchange.

2 Technical Overview of the Oracle Separation

In this section, we explain our main oracle separation. We defer the overview of our other results to a later section. For the ease of exposition, here we use a set of simplified notations and symbols slightly different from those in the rest of the paper.

2.1 An Oracle Separation between OWFs and CISFs

How could we build a function that is sequential but not one-way? An initial starting point might be to consider a function f that is sequential (i.e. if it takes C time to compute f, then f^T takes time TC) but also similarly easy to invert (i.e. f^{-1} also takes T time to compute). We could abstract this as a random permutation $\mathcal{P}$ where we also provided (efficient) access to $\mathcal{P}^{-1}$.

As any block cipher expert would know, the above idea does not work: the Davies-Meyer construction uses exactly such a random permutation (and its inverse) to build an iterated hash function, exploiting the fact that, in the random permutation model, $\mathcal{P}(x) \oplus x$ is pseudorandom even in the presence of an oracle for $\mathcal{P}^{-1}$. So the basic idea of giving out a permutation $\mathcal{P}$ and its inverse $\mathcal{P}^{-1}$ will not work.

A More Sophisticated Idea. Instead, consider the following: what if we still consider a pseudorandom permutation $\mathcal{P}$, but instead of giving out just its inverse $\mathcal{P}^{-1}$, we give out a more general oracle $\mathcal{O}_{\mathcal{P}}$, which we will call a *full inversion oracle*? Suppose we let $\mathcal{O}_{\mathcal{P}} : \mathcal{C} \times \{0,1\}^{\ell'} \to \{0,1\}^{\ell}$ be an oracle that takes as input a polynomially-sized binary circuit augmented with gates that compute $\mathcal{P}$ and a target value for the output of the circuit and outputs a valid input (which could be arbitrarily polynomially long). We say that the *size* of the input circuits is equal to the number of $\mathcal{P}$ "gates" in the circuit. Critically, we require that $\mathcal{O}_{\mathcal{P}}\left(c \in \mathcal{C}, x \in \{0,1\}^{\ell'}\right)$ take time Tt to evaluate, where T is the size of c and $\mathcal{P}$ takes time t to evaluate.

Note that such an oracle lets us immediately compute $\mathcal{P}^{-1}$ as before (we can just ask our oracle to solve for x in $P(x) = y$), but also gives us substantially more power. For instance, we can use $\mathcal{O}_{\mathcal{P}}$ to efficiently break the Davies-Meyer construction and other related constructions.

Handling Recursion. While the above solution seems elegant, it does not get us all the way to eliminating the possibility of one-way functions. For instance, given that we can call $\mathcal{O}_{\mathcal{P}}$ on arbitrary inputs, what if we chain calls to $\mathcal{O}_{\mathcal{P}}$? Or, even worse, what if we chain calls and mix in some calls to $\mathcal{P}$ in the middle? One might imagine a circuit of the form, "solve this equation, use the solution in a circuit made of gates that call $\mathcal{P}$, and then use that as the solution to another equation to solve, and output the input that solves that equation." It's not necessarily clear that we can efficiently invert these kinds of circuits using just $\mathcal{O}_{\mathcal{P}}$ as we have currently defined, so we need to modify the oracle $\mathcal{O}_{\mathcal{P}}$ further.

More precisely, we will let $\mathcal{O}_{\mathcal{P}}$ recursively call the output of oracle calls and use them in its solver. In other words, we will modify the circuit class $\mathcal{C}$ which can be input to $\mathcal{O}_{\mathcal{P}}$ to also include "$\mathcal{O}_{\mathcal{P}}$-gates." We note that these "circuit-solver gates" will take the same time to evaluate as the sub-circuit would. We note that this technique has been used before in black-box separations [GMM17].

As an illustrating example, suppose we let C be the circuit that computes $\mathcal{P}^n$ for some n. Then $\mathcal{O}_{\mathcal{P}}(C, y)$ would find the input value x such that $P^n(x) = y$. We could then define $C' = \mathcal{O}_{\mathcal{P}}(C, \cdot)$ and then compute $\mathcal{O}_{\mathcal{P}}(C', y)$. Of course, an adversary (or anyone) wouldn't be restricted from calling $\mathcal{O}_{\mathcal{P}}$ on non-algebraic inputs (i.e. those that are not directly input into a $\mathcal{P}$ gate), either.

Non-One-Wayness. Do our oracles $\mathcal{P}$, $\mathcal{O}_{\mathcal{P}}$ imply a one-way function? We have deliberately designed them so it is straightforward to show that they do not. Any circuit that can be built using $\mathcal{P}$, $\mathcal{O}_{\mathcal{P}}$, and standard binary gates can, by definition, be inverted by a call to $\mathcal{O}_{\mathcal{P}}$ in time proportional to the size of the circuit, which must be polynomial. So our definitions make arguing (non-)one-wayness relatively straightforward; we have pushed all of the hardness of the proof to proving sequentiality.

Sequentiality. Can we prove that our oracle collection is sequential? In other words, given $\mathcal{P}$, $\mathcal{O}_{\mathcal{P}}$, and some element x, does it still take time proportional to T to compute $\mathcal{P}^T(x)$? This is considerably more complicated than showing non-one-wayness. If we only gave access to $\mathcal{P}$, then our function would clearly be sequential. Intuitively, what we need to show is that giving access to $\mathcal{O}_{\mathcal{P}}$ doesn't give an adversary any advantage.

This might seem straightforward to prove at a first glance: after all, being able to compute $\mathcal{P}^{-1}$ doesn't seem to give an adversary too much extra power, and using the inversion oracle on with bit strings that are unrelated to the existing labels seems even less likely to be useful to an adversary. However, formalizing this is complicated.

A Simpler Oracle. Analyzing a random permutation can be difficult. To be able to avoid some of the analytical difficulties involved with random permutations, we choose to work with a permutation that is a "single cycle". More precisely, we consider a very weak version of a generic group oracle: we essentially have a generic group where it is only allowed to multiply (assuming a multiplicative group) by an apriori-fixed group element. Consider some prime order, cyclic group $\mathbb{G}$ of order q, and let ℓ be an integer such that $2^\ell \gg q$. Finally, let $\mathcal{C}$ be a class of circuits that consists of the usual binary gates (any complete representation) as well as "multiply" gates. Notice that here in the overview we use a simplified version of syntax, which does not exactly match our syntax in Section 5.

We will define two oracles $\mathcal{O}_{\mathsf{Mult},x} : \{0,1\}^\ell \to \{0,1\}^\ell$ and $\mathcal{O}_{\mathsf{Inv}} : \mathcal{C} \times \{0,1\}^\ell \to \{0,1\}^\ell$ that work in approximately the following way. First, we note that the oracles share a "database" that takes

the form of a constraint graph (cycle) with the following properties:

- The database stores pairs of values $\left(g \in \mathbb{G}, y \in \{0,1\}^\ell\right)$. The element g tells us "where we are" on the cycle graph, and the bit string is the "label" for the element.
- The database is instantiated with a "base point" $\left(h \in \mathbb{G}, y \in \{0,1\}^\ell\right)$ for h and y sampled uniformly at random. WLOG, we will usually assume that h is the identity element of $\mathbb{G}$.
- The database is instantiated with pairs $\left(h' \in \mathbb{G}, y' \in \{0,1\}^\ell\right)$ for all other $h' \neq h \in \mathbb{G}$, where the corresponding y' for each h' are sampled uniformly at random. We note that in some of our hybrids we will lazily simulate this step.

In our full proof, we formalize a notion of time by using *rounds*. Each round acts as a discrete unit of time, and any reader familiar with universally composable security [Can01], memory-hard functions [AB16], or multiparty computation [Lin20] should be immediately familiar with this notion. This allows us to handle the fact that different operations take different (relatively) amounts of time: for instance, computing $\mathcal{O}_{\mathsf{Mult},x}(y)$ for some input y might take one round, and $\mathcal{O}^2_{\mathsf{Mult},x}(y)$ might take two rounds, modeling the fact that the latter operation should intuitively take twice as long as the former.

We can now define our multiplication oracle $\mathcal{O}_{\mathsf{Mult},x}$. Recall that $x \in \mathbb{G}$ will be our "generator element".

- On a query $z \in \{0,1\}^\ell$ the oracle does the following:
  - Checks if z forms part of a valid tuple. If not, outputs $\perp$.
  - Otherwise, finds the tuple (j, z). The oracle then computes xj and looks it up in the set of tuples.
  - Then, *in the next round*, the oracle outputs the bit string in the tuple associated with the group element xj.

Finally, we can define our inversion oracle $\mathcal{O}_{\mathsf{Inv}}$.

- On a query that contains a circuit $C \in \mathcal{C}$ *of size* d and a bit string $z \in \{0,1\}^\ell$ the oracle does the following:
  - Checks the database to see if there are satisfying inputs t such that $C(t) = z$. Note that this may not be an efficient process without a PSPACE oracle.
  - If there is at least one satisfying set of inputs, choose one at random and output it d *rounds later*. Otherwise, output $\perp$ after d rounds have elapsed.

Note that this oracle would be very similar if we used a random permutation instead of a "very weak" generic group (where we can only multiply by one element). The only difference is that we are enforcing the fact that there is only one "loop." We expect that our proof would go through with a proper random permutation than our oracle, but it would severely complicate bookkeeping, so we choose to use this oracle instead. We also want to reiterate that our inversion oracle $\mathcal{O}_{\mathsf{Inv}}$ can take as input circuits consisting of binary gates, calls to $\mathcal{O}_{\mathsf{Mult},x}$, and calls to itself.

Lazily Simulating the Oracle. It will be important for us to be able to lazily simulate the oracle in all of our proofs. How might this work? If we only gave an adversary access to $\mathcal{O}_{\mathsf{Mult},x}$ and not $\mathcal{O}_{\mathsf{Inv}}$, then it would be very simple: a simulation would work in a very similar way to any number of generic group oracle simulations. However, we also have to be able to simulate queries to $\mathcal{O}_{\mathsf{Inv}}$, so unfortunately the process for us is not so simple here.

We will need a PSPACE oracle in order to successfully lazily simulate the oracle. To see why this is the case, note that the adversary could embed arbitrarily difficult binary circuits into a circuit C which was then provided as input to $\mathcal{O}_{\mathsf{Inv}}$. Without the power of a PSPACE oracle, we would not be able to successfully find solutions to these circuits. In general, powerful oracles are necessary for many black-box separations to ensure that the only "hardness" that participants can rely upon comes from the oracles provided in the game descriptions. We note that other separations (e.g. [IR89] and [BM09]) use a similar approach. There are some subtleties in using a PSPACE oracle to help us answer queries, but we defer these to section 5.

To lazily simulate the oracle, we will start with just the instantiation of the base point (h, y). When an inversion query is asked, we will use the PSPACE oracle to compute the probability that such a query is satisfied with respect to the current values of the database (including values that have not yet been set). Based on the probability that a query has a solution or not, we will then either respond with $\perp$ or use the PSPACE oracle to sample a random response from the set of satisfying states.

Note that answering a query will involve either new assignments to or, in the case of a $\perp$ response, new constraints (we need to be sure our simulation is consistent) to our database. After a query, we will update our database with new values (i.e. add labels to nodes on the cycle) or constraints, respectively. We can simulate an arbitrary polynomial number of queries in this way. Since we are actually writing down values and constraints in our simulation, we can only handle a polynomial number of queries, which is overall the reason why we must use a fine-grained notion of sequential function like those of [JMRR21] rather than those of [BBBF18].[1]

Splitting up the Oracle. Now that we can simulate the oracle, we will explain how we can argue that it still implies a sequential function. Our core technique will essentially be a hybrid argument where we "split" the oracle into two parts, one of which we simulate arbitrarily, and the other which is responsible for queries that correspond to small powers of x relative to our base point h which we answer honestly. We explain this in more detail below.

Suppose we consider our oracle as before, but instead of a tuple $(\mathcal{O}_{\mathsf{Mult},x}, \mathcal{O}_{\mathsf{Inv}})$ we provide a tuple $\left(\mathcal{O}'_{\mathsf{Mult},x}, \mathcal{O}'_{\mathsf{Mult},x^{-1}}\right)$, where $\mathcal{O}'_{\mathsf{Mult},x^{-1}}$ simply does the exact same thing as $\mathcal{O}_{\mathsf{Mult},x}$ except it multiplies by x^{-1} instead of x. It is very simple to see that it will take at least T queries to compute hx^T $(= x^T$ since we assume W.L.O.G. that $h = 1)$ given h (assuming WLOG it is the identity element) and these oracles with high probability; an adversary must successfully guess a label representing a group element (which happens with extremely low probability) to do this and a proof follows as a simple extension of the result that random oracles are sequential [MMV13].

So, we know that, given only the above "simple" oracle, our function would be sequential. But can we extend this to a full proof? We would need to be able to simulate all of the queries, including those coming from the inversion oracle, *to be consistent with this tuple of oracles* (or at least statistically close). This simulation would have to be indistinguishable from the "real" simulation, which we have outlined before.

A Dual Simulation. It turns out we can! Our reduction will boil down to the following: we will start with a tuple of oracles $\left(\mathcal{O}'_{\mathsf{Mult},x}, \mathcal{O}'_{\mathsf{Mult},x^{-1}}\right)$ and show how to simulate the full tuple of oracles

[1] We think this is the more intuitive notion anyway since it is usually against convention for adversaries to be able to perform superpolynomial work.

$(\mathcal{O}_{\mathsf{Mult},x}, \mathcal{O}_{\mathsf{Inv}})$ in a way that is indistinguishable to an adversary that can only make $T-1$ sequential queries (for some T) with an arbitrary polynomially bounded parallelism from our earlier, perfect simulation. The main challenge is showing that using the tuple of oracles we are given as input, $\left(\mathcal{O}'_{\mathsf{Mult},x}, \mathcal{O}'_{\mathsf{Mult},x^{-1}}\right)$, which outputs values *independent of the rest of the simulation* (although the simulated terms can be dependent on terms we receive from these oracles), instead of faithfully simulating, does not change the overall distribution of the outputs of the simulation (responses to queries) in a statistically significant way.

Our simulation will essentially work as follows:

- In the first round, we (as the simulator) will query $\mathcal{O}_{\mathsf{Mult},x}(1)$ and $\mathcal{O}_{\mathsf{Mult},x^{-1}}(1)$. In the k-th round, we will query $\mathcal{O}_{\mathsf{Mult},x}(s_{x^{k-1}})$ and $\mathcal{O}_{\mathsf{Mult},x^{-1}}(s_{x^{-k+1}})$, where $s_{x^{k-1}}$ and $s_{x^{-k+1}}$ are the labels representing x^{k-1} and x^{-k+1}, respectively.
- We will add these values to the input/output tuples to our query list, with the appropriate algebraic representations.
- We will simulate using our general lazy simulation as before, with the addition of the above operations.

Why is this a simulation that looks correct to the adversary? Well, very informally speaking, the distribution will be perfectly correct–in other words, all of the tuples $\left(h' \in \mathbb{G}, y' \in \{0,1\}^{\ell}\right)$ are distributed appropriately–as long as the simulation doesn't conflict non-negligibly with the queries from the multiplication oracle tuple we used as input. In other words, the only time this simulation does not work is if the simulator is forced to instantiate or provide some kind of non-negligible distinguishing information on a value that can be efficiently reached using the multiplication oracles, which correspond to small (positive and negative) powers of x. For some intuition, note that in the absence of information from inversion queries where we receive $\perp$ (which we discuss later), we fail if our general lazy simulation technique creates a label corresponding to a value x^c for some $c \in [-k, k]$ before round k has been reached.

It's worth it to think for a little bit about how an adversary can "misbehave". At a first glance, it might seem possible to show that an adversary cannot generate new elements without using the "basic" multiplication oracle, but this is not true. As an example, note that we can, in fact, generate valid labels using the $\mathcal{O}_{\mathsf{Inv}}$ oracle: on an input circuit C which is just a single multiplication gate, we could tell the oracle to "solve" for an element that has a label with first bit one, which happens with high probability. Then we could tell the oracle to solve for an element that had a label with first bit one and second bit one, and then first bit one and second bit zero. One of these would have to succeed, and we could repeat until we got a proper label.

This seems like it might be a problem for our proof: how do we handle these rogue inversion circuits? The answer is actually pretty simple: intuitively, if we ignore constraints implied by '$\perp$' responses to queries, if we stumble upon a label that is not tied to any existing other label (it is not the output of $\mathcal{O}_{\mathsf{Mult},x}$ or $\mathcal{O}_{\mathsf{Mult},x^{-1}}$ for any existing labels in our lazily simulated database) then it is equally likely to be assigned to any non-assigned group element. Thus, it is likely to be "far away" on the cycle graph from existing group elements (small powers of x) that we are trying to reach, and thus is unlikely to conflict with these terms or be useful for distinguishing them from faithfully sampled values. Formalizing this "location independence" is one major crux of our argument.

Proving This Simulation Works. As an astute reader can probably guess at this point, the tricky part in our proof is responding to the queries to the inversion circuit. Recall that we respond to inversion queries on circuits of size d after a wait of time d. Proving that this simulation works

takes three general steps:

- We show that any new labels (or chains of labels) instantiated by the inversion circuit that are not "connected" on the cycle graph to any other existing labels are located randomly on the cycle graph.
- Next, we show that any existing chain of labels cannot be extended by d labels in $< d$ rounds with non-negligible probability, assuming there is no "joining" of chains of labels. This follows from a minor adapatation of standard argument showing that random oracles are sequential.
- Finally, we show that, except with negligible probability, chains of labels cannot be joined together. This follows from the above two facts, in that these chains must be far apart and can only grow a polynomial amount during our whole game.

All of these steps – lemmas in our proof – together show that the inversion oracle can not be used to reach small powers of x, which lets us complete the proof. We note that our proof approach here has some similarities with the "two oracle trick" in [HR04] as we are hybridizing over the oracles.

Handling "$\perp$" Query Responses. One thing that we have not mentioned much so far is the impact that queries that have $\perp$ as an output have on our overall arguments: they do, in fact, change our distributions. But, it turns out they change them in such a small way that they can essentially be ignored. Why is this the case? To see why, think about when a query to $\mathcal{O}_{\mathsf{Inv}}$ returns $\perp$: with all but negligible probability, this happens when the probability the number of remaining, legal ways to instantiate the oracle (i.e. assign labels to all of the group elements) for which there is *no satisfying assignment* to the circuit is *at least* $\frac{1}{poly()}$ of all potential remaining, legal ways to instantiate the oracle.

By corollary, such $\perp$ responses only reduce the number of possible label assignments by a $\frac{1}{poly}$ multiplicative factor. However, since we have a doubly exponential amount of possible label assignments, this turns out to not be that substantial. For instance, this could correspond to fixing a *logarithmic* number of bits on a *single label*. This doesn't intuitively seem like it could help an adversary too much. If an adversary learned a negligible amount about every label (maybe no label could have some particular assignment) or a substantial information about a label of a randomly selected group element, then these $\perp$ outputs wouldn't be a useful source of information to the adversary.

However, what if the adversary could concentrate all of this "advantage" from a $\perp$ response into a single *useful* label (i.e. x^T for some small T)? Perhaps even a label that would let it guess a new label when it shouldn't be able to do so? In theory, this could constitute an attack: if an adversary somehow managed to concentrate its advantage from "$\perp$" query responses onto a single label which it was trying to predict (say, in a way that extends an chain of labels that it shouldn't be able to do), it might be a problem for our proof.

If it seems confusing as to how the above might work, that's OK: we can show that no such attack can actually work. To see this, note that any query that gets a $\perp$ response on a circuit of size d will take time d to be returned. Suppose, in this time, the adversary also "extends" the label chains it knows by computing calls to $\mathcal{O}_{\mathsf{Mult},x}$ and $\mathcal{O}_{\mathsf{Mult},x^{-1}}$ on the respective labels at the "edges" of chains. It will create d new labels on each side of existing label chains. The circuit for which $\perp$ was ouput may not be independent of all of the labels—after all, the adversary could incorporate known labels into the circuit—but it will be "almost" independent of the d labels on each end of any given chain (unless we somehow had prior non-negligible information on these labels, but we

can assume by induction that that is not the case) because these labels are determined after the circuit with the $\perp$ output was submitted to the adversary.

Now, if the adversary wants to use information from the $\perp$ queries to predict labels, it needs to apply the circuit in a specific location (with known or unknown labels) and then see if any specific label values or combinations are eliminated. To predict labels near the boundary of any chains, a circuit of depth d will only use the d labels closest to the edges of the chain. But we know that, from our above argument, these labels are independent (or statistically close to independent) from our circuit that output $\perp$. So therefore we aren't likely to learn more information about these labels (in parcticular) than any other labels, and thus we learn no more than negligible information about these labels from $\perp$ queries. Formalizing this argument is the key to one of the steps in our overall hybrid argument.

Implications. Our oracle – and its related separation – show that we could potentially live in a world where one-way functions do not exist but sequential functions do exist. This would be a very interesting portion of Pessiland [Imp95], and, as we show below, it wouldn't completely rule out cryptography (at least in a fine-grained sense).

Why Circuit Size and not Circuit Depth for the Inversion Circuit? At a first glance, it would appear that we should have defined our inversion oracle to return results in time proportional to the *depth* of a circuit, not the overall size. However, this turns out to be problematic, and we thank *anonymized* for pointing this out to us. To see this, consider a circuit C with two "layers" (illustrated in figure 1): the bottom layer is sets of bits which can represent valid inputs and outputs to $\mathcal{O}_{\mathsf{Mult},x}$, which we label $P_0, ..., P_k$. For each $i \in [0, k-1]$, we "connect" P_i and P_{i+1} with a circuit stating that $\mathcal{O}_{\mathsf{Mult},x}(P_i) = P_{i+1}$.[1] This results in a circuit of constant depth and width $k+1$ (or k if we only count multiplication gates).

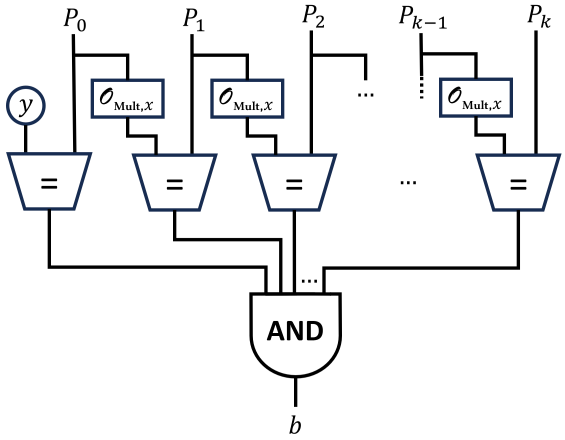


Figure 1: Illustration of the circuit C with input wires $P_0, P_1, \ldots, P_k \in \{0,1\}^\ell$ and output wire $b \in \{0,1\}$. $y \in \{0,1\}^\ell$ is the label for the group identity and is hard-coded as a constant.

Note that if we give this circuit C to $\mathcal{O}_{\mathsf{Inv}}$, it will be the case that, with high probability, $P_k = \mathcal{O}^k_{\mathsf{Mult},x}(P_0)$. If we allowed $\mathcal{O}_{\mathsf{Inv}}$ to return a solution to this circuit in time less than k, we

[1] We can easily check equality with AND gates in binary circuits.

would break the sequentiality of our candidate sequential function. Thus, we must not return any outputs of $\mathcal{O}_{\mathsf{Inv}}$ in time less than the number of multiplication gates in the input circuit. This unfortunately means that we do not have a tight separation: we only rule out one-way functions that take superpolynomial time to invert relative to the time necessary to compute them (and not fine-grained OWFs). However, we consider this satisfactory since this rules out all standard definitions of OWFs.

3 Overview of Other Results

We now outline our other results, which further explore the theory of sequential functions and connections to cryptography.

3.1 Sequential Functions and PSPACE

We next explain our results concerning the equivalence of a deterministic, worst-case CISF and PSPACE, and the non-equivalence of a worst-case variant of a general sequential function and PSPACE.

CISFs and PSPACE Are "Equivalent". Informally speaking, recall that a CISF is a sequential function f that can be parameterized by a Round function, where f can be written in the form $f = \mathsf{Round}^k(x)$ for some input x and some number of rounds k. We define and explain this fully in Section 6. Importantly, the Round function must be evaluatable in polynomial time and have polynomial input (and output) size. From this, it follows almost immediately that CISFs are in PSPACE.

Showing that worst-case, decisional CISFs are PSPACE-complete is trickier. It's straightforward to define a worst-case, decisional CISF variant: to make the problem decisional, we can just define the problem in the form of whether or not a particular value is the correct evaluation of the CISF on a particular input, and to make it worst-case, define the problem to hold for *all* choices of valid public parameters for the CISF. For a full, formal definition, please see Section 6.

To show that this version of a CISF is PSPACE complete, we use the *true quantified boolean formula* (TQBF) problem. Recall a TQBF problem statement is of the form $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi(x_1, \ldots, x_n)$ where $Q_1, \ldots Q_n \in \{\exists, \forall\}$ are quantifiers, $x_1, \ldots, x_n$ are boolean variables, and ϕ is a boolean formula. The answer to the problem is true if and only if there exists an assignment of $x_1, \ldots, x_n \in \{0,1\}$ such that the overall boolean formula $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi(x_1, \ldots, x_n)$ is true.

A folklore result (explained in complexity theory textbooks, including [Pap07]) shows how to recursively evaluate the validity of a TQBF statement. We write this recursion in the form of a binary tree evaluation, and then show how to write a Round function that appropriately handles each step of the tree evaluation, one after another (if desired). A naive evaluation of this recursion might use exponential space. To ensure that this evaluation remains in PSPACE, we evaluate the tree using a depth-first search style of evaluation. For full details, we again refer the reader to Section 6.

General Sequential Functions Are Probably Not in PSPACE. A *dynamic sequential function* (DSF) is a more general form of sequential function that does not necessarily have a nice round

structure like a CISF. We define DSFs precisely in Section 4, but the exact properties of a DSF beyond being a sequential function are not extremely important here. We show that there exists a DSF that is not evaluatable in PSPACE in the random oracle model.

Our proof technique borrows from the constructions of memory-hard functions in the random oracle model. As in the construction of many memory-hard functions, we first construct a graph which requires a certain minimal number of pebbles to *pebble*, which we formally define in Section 6. Then we use a technique by [DNW05] to convert the graph to a function, where roughly speaking, each node in the graph represents a random oracle evaluation and the edges represent the input/output flow. [DNW05] show that the time and space lower bounds for pebbling the graph readily translate to time and space lower bounds for evaluating the resulting function. However, the parameters we need are quite different from those used in memory-hard functions: secure memory-hard functions require some minimal polynomial memory to evaluate efficiently, while we need to show that a DSF exists that requires superpolynomial memory to compute.

One line of memory-hard functions is based on a well-known sequence of graph constructions: concentrators $\rightarrow$ hyperconcentrators $\rightarrow$ superconcentrators[1]. We use this stack of graphs to build a graph of random oracle evaluations that takes superpolynomial memory to evaluate. Unfortunately, there are quite a few complications in getting this approach to work: for instance, we need to show that all of our graphs can be *compactly described* in polynomial space–even if the graphs themselves are superpolynomially large–and that, given a node on a graph and polynomial advice (in our case, the description of the graph), its neighbors can be found, and these properties were not shown previously in the literature. So, we have to take a deep dive into classic graph constructions, which is a little bit tedious. However, this does, in fact, work out; for full details, please see Section 6.

3.2 Fine-Grained Cryptography from Sequential Primitives

We next explain the core ideas behind our constructions of fine-grained cryptoprimitives from sequential functions and VDFs. We assume that we have a CISF (or, later, a VDF) that is *not necessarily one-way*; our proofs only rely on sequentiality and not one-wayness. Of course, if we had access to one-way functions, we could immediately build much stronger cryptoprimitives.

Fine-Grained Symmetric Key Encryption. We start by showing how to build fine-grained symmetric key encryption from a sequential function. Recall that traditional symmetric key encryption, informally speaking, is an encryption scheme where two participants Alice and Bob have a shared secret key and must send message(s) between each other in a way that an eavesdropping adversary Eve cannot learn anything about the messages being sent. In traditional symmetric key encryption, security must hold against any PPT adversary Eve; but in fine-grained symmetric key encryption, security must only hold against a PPT adversary for some (polynomial) amount of time.

With this in mind, a simplified version of our scheme is (informally) as follows. In this overview, we will assume a CISF f that outputs bit strings and takes time T to compute for both honest and adversarial parties, although our construction can be generalized to other outputs and parameters.

- **Setup**: Alice and Bob begin with a shared secret, which consists of the description of a CISF f as well as a shared key x that is a valid input for f.
- **Precomputation**: Alice and Bob both take Tnk time to compute $f^{nk}(x)$ for some integers n, k, and they record each output $f^{ik}(x)$ for integers $i \in [0, n]$.

[1]For an unfamiliar reader, we define all of these explicitly in Section 6.

- **Initial Encryption**: To encrypt an initial (bit) message m_1, Alice samples a random binary string r_1 with the same length as the output of f and computes $c_1 = \langle f^k(x), r_1 \rangle \oplus m_1$, where $\langle \cdot, \cdot \rangle$ denotes the inner product, and $\oplus$ denotes the XOR operation. The ciphertext consist of the tuple $\mathsf{ct}_1 = (c_1, r_1)$.
- **Time Passes**: Suppose Tk time passes, and during this time Alice and Bob continue to iteratively compute f.
- **Next Encryption**: To encrypt a message m_2, Alice first samples a random r_2, computes $c_2 = \langle f^{3k}(x), r_2 \rangle \oplus m_2$ and outputs $\mathsf{ct}_2 = (c_2, r_2)$.[1]
- **The Ending**: This process can continue until the encryptions "catch up" to Alice and Bob with respect to f. In our example here, Alice and Bob will be able to exchange $\left\lfloor \frac{n}{2} \right\rfloor - 1$ messages, since each message "uses" k computations.

Decryption in the above scheme by Bob is immediate since Bob will have precomputed and stored all of the output values of f that appear in the ciphertexts, but what about security? As an example, let's consider the second encryption ct_2. Even if Eve sees $f^k(x)$ in the clear, she sees ct_2 only Tk time later. However, to successfully decrypt ct_2 at the time it is sent, Eve would need to compute $f^{3k}(x)$ given $f^k(x)$ in time Tk, which would contradict the sequentiality property of the CISF. However, note that security would only last for Tk time, because Eve could have computed $f^{3k}(x)$ by that time. We formalize our definitions of fine-grained symmetric key encryption and our scheme (and provide a security proof) in Section 7.

Fine-Grained MACs. It turns out there is a simple extension of the above argument to *message authentication codes* (MACs) as well. Given a sequence of bits $b_{i,j}$ for $i \in [1, \ell]$ and $j \in \{0, 1\}$ (for some ℓ that approximates a security parameter) that are known to Alice and Bob but unpredictable to Eve for some amount of time–which we can generate using the construction above–we can construct a MAC of a *single bit* message representing a bit x by outputting the bitstring $b_{1,x}||b_{2,x}||...||b_{\ell,x}$. We can obtain these bits in a "fine-grained secure" way using the same idea as from our symmetric-key encryption scheme. For messages longer than one bit, we can just repeat this core idea for each bit.

This MAC is obviously extremely inefficient, and unfortunately is not very useful for MACing large messages either: to do so would seemingly require a CRHF, which would imply a OWF, which we are not assuming. So this construction would be very bad practically, but we still find it interesting because generic constructions of MACs from symmetric-key encryption, while known in the standard model, would not necessarily hold in the fine-grained setting.

Fine-Grained PKE from VDFs and null iO. We next show how to build fine-grained PKE from verifiable delay functions (VDFs) with certain additional properties and indistinguishability obfuscation (iO) for null circuits. Recall that a VDF is, informally, a sequential function that can be computed along with an (efficiently verifiable) proof that it was computed correctly. In our work, we will assume some (somewhat strong) additional properties on the VDFs we need; we refer the reader to Section 9 for the full details.

To build fine-grained PKE from these assumptions, Alice first samples a VDF f, an input x, and computes $y = f(x)$ as well as a proof π. Alice's public key is the tuple (x, π) and the secret key is y. To encrypt a message, Bob obfuscates a program that, informally speaking, does the following:

- On input y', if the VDF proof verification holds for (x, y', π), output the message m.
- Otherwise, output $\perp$.

[1] Note that an adversary could have computed $f^{2k}(x)$ by this time period, which is why Alice must use $f^{3k}(x)$.

Note that the security of construction almost immediately follows if we were to use virtual black-box (VBB) obfuscation: the only effective way to make the oracle output the message m is to provide the correct output of the VDF y. Moreover, note that proof verification of VDFS is required to be efficient (with time uncorrelated to the length of time necessary to compute the VDF) so the program being obfuscated can be efficient as well. The main challenge in the proof is using null iO instead of VBB; we defer the details of this to Section 9 but offer some intuition as to why the proof works below.

Our proof for iO is actually quite subtle: a natural approach would be to use a puncturing argument, but we are assuming that OWFs do not exist, and therefore have no PRFs (or other minicrypt primitives) to puncture! Instead, we assume that our VDF has a property called *proof indistinguishability*: namely, an adversary that has not computed the final output of the VDF cannot distinguish a correct proof of a particular input (when given the input) from random. While we do not know whether or not this would hold for a VDF that does not imply a one-way function (because we don't know of any such VDFs), it does appear to essentially hold for some existing VDFs (e.g. that of [Wes19]) in that the proofs will look random unless "almost all" of the computation has been done.

With this property, as well as the fact that proofs are *unique*, we can construct a hybrid argument where we switch between encryptions of 0 and 1 in a way that is undetectable to an adversary. Roughly speaking, given an encryption of zero, we first switch out a correct proof with a proof that doesn't verify for *any input* because they are assumed to be indistinguishable (assuming only a certain amount of time has elapsed). At this point, we have a program that outputs $\perp$ on all programs, so we can apply null iO and switch to a different program–one that has one as a message instead of zero. Then, we can switch the proof back to being valid and we have an encryption of zero.

Note that, unlike our previous symmetric-key scheme, this construction allows us to encrypt arbitrary-length messages.

Fine-Grained Key Exchange and Commutative Sequential Functions. In order to explore what sort of primitive is necessary for fine-grained key exchange, we show how to build fine-grained key exchange from a primitive that we call a *commutative sequential function*. Informally speaking, a commutative sequential function is one where two players Alice and Bob can each "partially" compute a sequential function, send the partial computation to each other, and have the other party finish the computation, all while an eavesdropping adversary Eve, upon seeing the exchange of messages, cannot compute the function faster than if she started from the beginning. We explain this in detail in Section 10.

4 Preliminaries

Notation-wise, for $n \in \mathbb{N}$, we let $[n]$ denote the ordered set $\{1, 2, \ldots, n\}$.

For random distributions X and Y, let $H_\infty(X|Y)$ denote the min-entropy of X conditioned on Y. Furthermore, let U_m denote a uniformly distributed random variable of m bits for some positive integer m.

Definition 1 (Statistical Distance)**.** *Let D_1 and D_2 be two distributions with support in X. The*

statistical distance between D_1 and D_2 is

$$\Delta(D_1, D_2) = \frac{1}{2} \sum_{x \in X} |\Pr[D_1 = x] - \Pr[D_2 = x]|$$

Remark 1. *Let A and B be two random variables with support in X. We use $\Delta(A, B)$ to denote the statistical distance $\Delta(P_A, P_B)$ between the underlying distributions of the random variables.*

Remark 2. *Let $X \approx_\epsilon Y$ denote that the two distributions are statistically close, or ϵ-close, i.e. the statistical distance between these two distributions $\Delta(X, Y) \leq \epsilon$.*

Definition 2 (Mutual Information)**.** *Let X and Y be two two random variables with sample space $\mathcal{X}$ and $\mathcal{Y}$ respectively. The mutual information between X and Y is given by*

$$I(X;Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)} \right),$$

where $p(x, y)$ is the joint probability mass function of X and Y, and $p(x)$ and $p(y)$ are the marginal probability mass functions of X and Y respectively.

Remark 3. *One useful property that we will utilize of mutual information is*

$$I(X;Y) \equiv H(X) - H(X|Y),$$

where $H(X)$ is the marginal entropy of X and $H(X|Y)$ is the conditional entropy of X conditioned on Y. Notice that if X and Y are independent, then $I(X;Y) = 0$; if X and Y are identical, then $I(X;Y) = H(X)$ will be maximal.

Lemma 1 (Leftover Hash Lemma for Conditional Min-Entropy [ILL89])**.** *Let X, E be a joint distribution. If $H_\infty(X|E) \geq k$, and $m = k - 2\log(1/\epsilon)$, then*

$$(H(X), H, E) \approx_{\epsilon/2} (U_m, U_d, E),$$

where m is the output length of a universal hash function H, and d is the length of the description of H.

4.1 Sequential Functions

We here recall the definitions of sequential functions due to [JMRR21].

Definition 3 (Sequential Functions)**.** *A **selective sequential function (SSF)** $F = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval})$ is defined as the following tuple of algorithms:*

$\mathsf{Setup}(1^\lambda \boxed{, k}) \to \mathsf{pp}$**:** *On input the security parameter 1^λ $\boxed{\text{, and } k \in 2^{o(\lambda)}}$, the setup algorithm returns the public parameters pp. By convention, the public parameters encode an input domain X and an output domain Y.*

$\mathsf{Gen}(\mathsf{pp} \boxed{, k}) \to x$**:** *On input the public parameters pp $\boxed{\text{, and } k \in 2^{o(\lambda)}}$, the instance generation algorithm samples a random input $x \leftarrow X$.*

$\mathsf{Eval}(\mathsf{pp}, x, k) \to y$**:** *On input the public parameters* pp*, an input* $x \in X$*, and* $k \in 2^{o(\lambda)}$*, the evaluation algorithm returns an output* $y \in Y$*.*

An SSF is an ***Adaptive Sequential Function (ASF)*** *if* Setup *is independent of* k*. An ASF is a* ***Dynamic Sequential Function (DSF)*** *if* Gen *is independent of* k*. For SSFs, ASFs, and DSFs,* Gen *and* Setup *are required to be PPT in the security parameter. For SSFs and ASFs,* Gen *is allowed to also have a polylog dependency on* k*, and for SSFs,* Setup *is also allowed to have a polylog dependency on* k*.*

An SF F *satisfies* $(t_C(\lambda), t_A(\lambda))$*-sequentiality for machine models* $(\mathcal{M}_C, \mathcal{M}_A)$ *if the following hold:*

1. *There exists an algorithm in the computational model* $\mathcal{M}_C$ *such that for all* k *and for all* x *that can be output by* Gen*, it computes* Eval *in at most time* $k \cdot t_C(\lambda)$*.*
2. *For all* $\lambda \in \mathbb{N}$ *and for all tuples of PPT machines* $(\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$*, such that* $\mathcal{A}_2$ *runs in time strictly less than* $k \cdot t_A(\lambda)$ *in the computational model* $\mathcal{M}_A$*, there exists a negligible function* $negl$ *such that:*

   (a) *If* F *is a selective sequential function:*

$$\mathsf{Pr}\left[\, y = y' \;\middle|\; \begin{array}{c} (k, \tau_0) \leftarrow \mathcal{A}_0(1^\lambda),\ \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, k), \\ \tau_1 \leftarrow \mathcal{A}_1(\mathsf{pp}, k, \tau_0),\ x \leftarrow \mathsf{Gen}(\mathsf{pp}, k), \\ y' \leftarrow \mathcal{A}_2(\mathsf{pp}, x, k, \tau),\ y \leftarrow \mathsf{Eval}(\mathsf{pp}, x, k) \end{array} \right] = negl(\lambda)$$

   (b) *If* F *is an adaptive sequential function:*

$$\mathsf{Pr}\left[\, y = y' \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda),\ (k, \tau) \leftarrow \mathcal{A}_1(\mathsf{pp}), \\ x \leftarrow \mathsf{Gen}(\mathsf{pp}, k), \\ y' \leftarrow \mathcal{A}_2(\mathsf{pp}, x, k, \tau),\ y \leftarrow \mathsf{Eval}(\mathsf{pp}, x, k) \end{array} \right] = negl(\lambda)$$

   (c) *If* F *is a dynamic sequential function:*

$$\mathsf{Pr}\left[\, y = y' \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda),\ (k, \tau) \leftarrow \mathcal{A}_1(\mathsf{pp}), \\ x \leftarrow \mathsf{Gen}(\mathsf{pp}) \\ y' \leftarrow \mathcal{A}_2(\mathsf{pp}, x, k, \tau),\ y \leftarrow \mathsf{Eval}(\mathsf{pp}, x, k) \end{array} \right] = negl(\lambda)$$

We next define iterative sequential functions.

Definition 4. *An Iterative Sequential Function (ISF) is a Sequential Function such that the* Eval *function is iterative: there exists a function* Round *such that* $\mathsf{Eval}(\mathsf{pp}, x, k) = (\mathsf{Round}(\mathsf{pp}, \cdot, k))^{(k)}(x)$ *where '*$\cdot$*' represents some fixed-length, polynomially-sized bit string that is only dependent on the output of the previous call to* Round*. We have Selective, Adaptive and Dynamic Iterative Sequential Functions defined in the same way as Sequential Functions. In addition, we say that a DISF is a Continuous ISF (CISF) if* Round *is also independent of* k*.*

We will use CISFs considerably in this paper, so we provide some intuition hree. It may seem like every ISF is a CISF, but this is not the case. An example of an ISF that is not a CISF can easily be seen by modifying the FHE-based construction of [JMRR21] to not use FHE bootstrapping every round. If that construction used bootstrapping, say, every other round, then the Round function needs to take k as an input because it needs to do different things in even and odd rounds. A simpler (if more artificial) construction that is an ISF but not a CISF could be a sequential function based on raising an element of a group of unknown order to, say, the 3rd power in odd rounds and the 5th power in even rounds.

Alternative Definitions of Sequential Functions. We note that there are potentially other, alternative ways of defining sequential functions. One obvious way might be, for an ISF, allowing the adversary to choose k after seeing the input value x. For the applications we consider, typically k is fixed in advance, but for other applications, this might be a useful extension of the definition. We note that all of our results for ISFs still hold for this change to the definition with at most minor changes, including our main separation result.

4.2 Verifiable Delay Functions

We next recall the definition of Verifiable Delay Functions (VDFs). We use the definition from [JMRR21], which is adapted from [BBBF18, DGMV19]. Specifically, we define sequentiality *twice*: once for their (T, ϵ) notion of sequentiality, and once for our (T_C, T_A) definition of sequentiality. This new definition helps us more accurately take into account differences in computational models (and thus hardware). The two definitions are trivially equivalent if the same hardware model is used for both honest and adversarial evaluators.

Definition 5 (Verifiable Delay Function)**.** *A VDF* $V = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is defined as the following tuple of algorithms:*

$\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$**:** *On input the security parameter* 1^λ*, the setup algorithm returns the public parameters* pp*. By convention, the public parameters encode an input domain* X *and an output domain* Y*.*

$\mathsf{Gen}(\mathsf{pp}) \to x$**:** *On input the public parameters* pp*, the instance generation algorithm samples a random input* $x \leftarrow X$*.*

$\mathsf{Eval}(\mathsf{pp}, x, T) \to (y, \pi)$**:** *On input the public parameters* pp*, an input* $x \in X$*, and a time parameter* $T \in 2^{o(\lambda)}$*, the evaluation algorithm returns an output* $y \in Y$ *together with a proof* π*. The evaluation algorithm may use random coins to compute* π*, but not for computing* y*.*

$\mathsf{Vf}(\mathsf{pp}, x, y, \pi, T) \to \{0, 1\}$**:** *On input the public parameter* pp*, an input* $x \in X$*, an output* $y \in Y$*, a proof* π*, and a time parameter* T*, the verification algorithm outputs a bit* $\{0, 1\}$*.*

Efficiency. We require that Setup and Gen run in time $poly(\lambda)$, and Vf runs within $poly(\log(T), \lambda)$. We require Eval to run in exact parallel time T with at most $poly(\log(T), \lambda)$ processors.

Definition 6 (Completeness)**.** *A VDF* $V = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is complete if for all* $\lambda \in \mathbb{N}$ *and all* $T \in \mathbb{N}$*, the following holds:*

$$\Pr\left[\mathsf{Vf}(\mathsf{pp}, x, y, \pi, T) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ x \leftarrow \mathsf{Gen}(\mathsf{pp}) \\ (y, \pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x, T) \end{array}\right] = 1$$

Definition 7 (Soundness)**.** *A VDF* $V = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is sound if for all* $\lambda \in \mathbb{N}$ *and for all PPT machines* $\mathcal{A}$*, there exists a negligible function* $negl$ *such that:*

$$\Pr\left[\mathsf{Vf}(\mathsf{pp}, x, y', \pi', T) = 1 \text{ and } y \neq y' \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (T, x, y', \pi') \leftarrow \mathcal{A}_1(\mathsf{pp}) \\ (y, \pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x, T) \end{array}\right] = negl(\lambda)$$

We can define sequentiality for VDFs:

Definition 8 ((T,ϵ)-Sequentiality)**.** *A VDF $V = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ is (T,ϵ)-sequential if for all $\lambda \in \mathbb{N}$ and for all pairs of PPT machines $(\mathcal{A}_1, \mathcal{A}_2)$, such that the parallel running time of $\mathcal{A}_2$ (with $poly(T,\lambda)$ processors) is less than $(1-\epsilon)\cdot T$, there exists a negligible function $negl$ such that:*

$$\mathsf{Pr}\left[\, y = y' \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda),\ (T,\tau) \leftarrow \mathcal{A}_1(\mathsf{pp}),\ x \leftarrow \mathsf{Gen}(\mathsf{pp}) \\ y' \leftarrow \mathcal{A}_2(\mathsf{pp}, x, T, \tau),\ (y,\pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x, T) \end{array} \right] = negl(\lambda)$$

Definition 9 ((T_C, T_A)-Sequentiality)**.** *A VDF $V = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ is (T_C, T_A)-sequential if for all $\lambda \in \mathbb{N}$ and for all pairs of PPT machines $(\mathcal{A}_1, \mathcal{A}_2)$, such that the running time of $\mathcal{A}_2$ on computational model $\mathcal{M}_A$ is less than T_A, there exists a negligible function $negl$ such that:*

$$\mathsf{Pr}\left[\, y = y' \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda),\ (T_C,\tau) \leftarrow \mathcal{A}_1(\mathsf{pp}),\ x \leftarrow \mathsf{Gen}(\mathsf{pp}) \\ y' \leftarrow \mathcal{A}_2(\mathsf{pp}, x, T_C, \tau),\ (y,\pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x, T_C) \end{array} \right] = negl(\lambda)$$

4.3 Indistinguishability Obfuscation

Lastly, we recall the definition of indistinguishability obfuscation.

Definition 10 (Indistinguishability Obfuscation [BGI+01])**.** *An* indistinguiability obfuscator *$i\mathcal{O}$ for a circuit class $\{\mathcal{C}_\lambda\}$ is a PPT uniform algorithm satisfying the following conditions:*

- **Functionality***: For any $C \in \mathcal{C}_\lambda$, then with probability 1 over the choice of $C' \leftarrow i\mathcal{O}(1^\lambda, C)$, $C'(x) = C(x)$ for all inputs x.*
- **Security***: For all pairs of PPT adversaries (S, D), if there exists a negligible function α such that*

  $$\Pr[\forall x, C_0(x) = C_1(x) : (C_0, C_1, \sigma) \leftarrow S(\lambda)] > 1 - \alpha(\lambda)$$

  then there exists a negligible function β such that

  $$\left|\Pr[D(\sigma, i\mathcal{O}(\lambda, C_0)) = 1] - \Pr[D(\sigma, i\mathcal{O}(\lambda, C_1)) = 1]\right| < \beta(\lambda)$$

When $\mathcal{C}_\lambda$ is the class of all polynomial-size circuits, we simply call $i\mathcal{O}$ an indistinguishability obfuscator. There are several known ways to construct indistinguishability obfuscation:

- Garg et al. [GGH+13] build the first candidate obfuscation from cryptographic multilinear maps.
- Provably from novel strong circularity assumptions [BDGM20, GP21, WW20].
- Provably from "standard" assumptions [JLS21]: (sub-exponentially secure) LWE, LPN over fields, bilinear maps, and constant-locality PRGs.

For the sake of this paper, we only need a very weak form of $i\mathcal{O}$, namely $i\mathcal{O}$ for null circuits. It has the same functionality as a normal $i\mathcal{O}$, and can obfuscate circuits with non-null outputs, but its security only needs to hold for null programs that output $\perp$ on all inputs. Notice that it is equivalent to witness encryption [GGSW13].

Definition 11 (Security of $i\mathcal{O}$ for Null Circuits [WZ17, GKW17])**.** *For all pairs of PPT adversaries (S, D), if there exists a negligible function α such that*

$$\Pr[\forall x, C_0(x) = C_1(x) = \bot : (C_0, C_1, \sigma) \leftarrow S(\lambda)] > 1 - \alpha(\lambda)$$

then there exists a negligible function β such that

$$\left| \Pr[D(\sigma, i\mathcal{O}(\lambda, C_0)) = 1] - \Pr[D(\sigma, i\mathcal{O}(\lambda, C_1)) = 1] \right| < \beta(\lambda)$$

5 An Oracle Separation for One-Way Functions and Sequential Functions

In this section, we show that there exists an oracle that implies a sequential function but does not imply a time-bounded one-way function. Specifically, for a prime-order cyclic group $\mathbb{G}$ with order $p = \Theta(2^\lambda)$, a random generator g for the group, and a randomly sampled group element $r \in \mathbb{G}$, we consider the following function:

$$f^{\mathcal{O}_{\mathsf{Mult}}}(r, T) = r \cdot g^T$$

where we compute $r \cdot g^T$ in the group using a group multiplication oracle $\mathcal{O}_{\mathsf{Mult}}$ which computes the "multiply by g" group operation with the generator g hard-coded. To compute this result, one just need to call the $\mathcal{O}_{\mathsf{Mult}}$ oracle iteratively for T times, starting with the input r. While this would be one-way in any number of generic group models (e.g. [Sho97]), we also provide an oracle $\mathcal{O}_{\mathsf{Inv}}$ that takes as input circuits with standard binary gates, $\mathcal{O}_{\mathsf{Mult}}$ gates, and other $\mathcal{O}_{\mathsf{Inv}}$ gates that allows us to invert any kind of function or circuit we might build using these tools. As we explained in the overview, the challenge (and what we spend most of our time doing) is proving that providing access to the $\mathcal{O}_{\mathsf{Inv}}$ oracle does not break the inherent sequentiality of $\mathcal{O}_{\mathsf{Mult}}$.

Our construction borrows heavily from generic group techniques: we imagine that instead of given group elements directly, we are only given *labels* of group elements, similar to that of Shoup's version [Sho97] of the generic group model. For extensive intuition on our proof, please go back to section 2.

5.1 Sampling a PSPACE Solution

One of the core technical tools that the proof relies on is a uniform generator to sample a random accepting instance (i.e. solution) for a PSPACE-relation, similar to the generator for an NP-witness due to Bellare, Goldreich, and Petrank [BGP00]. They show how to construct a uniform generator for an NP-relation using an NP-oracle.

Lemma 2 ([BGP00])**.** *Let R be an NP-relation. Then there is a uniform generator for R which is implementable in probabilistic, polynomial time with an NP-oracle.*

We essentially want a PSPACE version of the above lemma, and it turns out that we can construct such generators quite directly (in fact, our lemma is much easier to prove than the one above).

Lemma 3. *Let R be a* PSPACE*-relation. Then there is a uniform generator for R which is implementable in probabilistic, polynomial time with an* PSPACE*-oracle.*

Proof. Let the input to R be n bits, the generator works in these following two steps.

First, we determine the total number of solutions m for the relation R. To do so, we construct the following PSPACE statement for $k = 0, 1, 2, \ldots, 2^n$: *there exists at least k solutions for R.* And then we can use the PSPACE-oracle to determine if the statement is true or not. Naively, we can iterate k from 0 up to 2^n to obtain the number of solutions with $O(2^n)$ calls to the PSPACE oracle. Or, we can perform a binary search by starting with $k = 2^{n-1}$, and recurse in the half where m lies in. This gives us m in $O(n)$ calls to the PSPACE oracle.

Then using m, we sample a uniform $i \in [m]$. The generator will output the i-th solution in the following way. Fix an ordering of all the inputs, say just an increasing order according to the bit values. We construct the following PSPACE statement: *the first bit of the i-th solution for R is* 0. We use the PSPACE oracle to solve for the statement and hence is able to determine the first bit of the i-th solution. We repeat this n times to get all the n bits of the i-th solution, which we output. □

5.2 Definitions for the Proof

In this subsection, we define the other necessary concepts for the proof. In particular, we introduce notation which will allow us to much more concisely state arguments in our proof. Let λ be the security parameter. In the proof, we will be working with the cyclic group $\mathbb{G}$ with prime order $p = \Theta(2^\lambda)$ and a generator g sampled uniformly at random.

Labeling of Group Elements. Similar to that of Shoup's version [Sho97] of the generic group model, we imagine access to the group elements through the *labels* of group elements, following the formalization by Zhandry [Zha22]. The labels of group elements will be n-bit strings with $2^n \gg p$ where $n = poly(\lambda)$. The label for a group element g^x with $x \in \mathbb{Z}_p$ is determined by the labeling function $L : \mathbb{Z}_p \to \{0,1\}^n$ modeled as a random function, and is denoted by $L(x)$. For instance, the group element g^7 will have label $L(7)$. Notice that x here is the exponent, and the group element is g^x.

Additionally, we let $L^{(x)}$ denote the distribution of the label for g^x, and $\ell^{(x)} \sim L^{(x)}$ be the actual sampled label. Similarly, for the ease of syntax, we let $x^{(\ell)}$ denote the exponent in g^x whose label is ℓ. For instance, if ℓ is the label for the group element g^7, we will have $x^{(\ell)} = 7$. We have $x^{(\ell)} = \perp$ if ℓ is not a valid label, i.e. there is no such x that g^x has the label ℓ. So technically, we have $x^{(\ell^{(x)})} = x$ and $\ell^{(x^{(\ell)})} = \ell$.

In some cases in our proof, we will require $\ell^{(x)}$ be to sampled from an appropriate distribution to account for possible constraints posted to the label distribution. For instance, through some carefully crafted queries, an adversary might be able to produce a constraint that the first bit of the label for g^x is 0. In that case, $\ell^{(x)}$ will no longer be a uniform n-bit string,. Instead, it will be sampled from the distribution $\ell^{(x)} \sim 0||U_{n-1}$, where U_{n-1} is a uniform distribution of a $(n-1)$-bit string.

Construction of the Sequential Function. The original sequential function definition due to [BBBF18] allows for sequentiality T up to $2^{o(\lambda)}$, but in order for the simulated random oracle to record the queries, here we only allow up to a polynomial number of queries. This requires us to use a fine-grained definition of sequential function such as from [JMRR21] that allows for only polynomial evaluations and not a definition of sequential function like that of [BBBF18] that allows

even honest users to compute the function a superpolynomial number of times. We note that these definitions are mostly equivalent in practice because in [BBBF18], the authors assume that both adversaries and honest users can perform superpolynomial sequential work. Please see Section 4 for the full definitions.

Towards constructing our sequential function, we sample a random generator $g \in \mathbb{G}$. For a desired time bound $T = poly(\lambda) \ll p$, our sequential function requires the adversary to query T times in sequence a "multiply by g" oracle $\mathcal{O}_{\mathsf{Mult}}$ which we define below, along with other oracles used in the proof. To model the sequential time required for different computations, the game will proceed in rounds, and the oracle queries will take some corresponding number of rounds to execute before they are answered. Roughly speaking, the more complicated an oracle query is, the more rounds it will need to wait before being answered. We note that, like a random oracle, our oracles are not efficiently implementable (they would require exponential space). We specify the oracles below:

- $\mathcal{O}_{\mathsf{Mult}}(\alpha_1) \to \alpha_2$: takes as input a label α_1. If $x^{(\alpha_1)} = \perp$, i.e. α_1 is not a valid label of a group element, output $\perp$. Otherwise, output $\alpha_2 = \ell^{(x^{(\alpha_1)}+1)}$, which is the label for the group element $g^{x^{(\alpha_1)}+1} = g^{x^{(\alpha_1)}} \cdot g$, where $g^{x^{(\alpha_1)}}$ is the corresponding group element for the label α_1. If the query is received in round i, the response will be provided at round $i + 1$.

- $\mathcal{O}_{\mathsf{MultInv}}(\alpha_1) \to \alpha_2$: takes as input a label α_1. If $x^{(\alpha_1)} = \perp$, i.e. α_1 is not a valid label of a group element, output $\perp$. Otherwise, output $\alpha_2 = \ell^{(x^{(\alpha_1)}-1)}$, which is the label for the group element $g^{x^{(\alpha_1)}-1} = g^{x^{(\alpha_1)}} \cdot g^{-1}$. If the query is received in round i, the response will be provided at round $i + 1$. *We note that this oracle is implied by $\mathcal{O}_{\mathsf{Inv}}$ but we list it here anyway because it plays an integral role in our proof.*

- $\mathcal{O}_{\mathsf{Inv}}(C, y) \to z/\perp$: takes as input a circuit C with potentially polynomially many input and output wires which can either represent bits or labels, a desired output y, and outputs a uniformly random solution z s.t. $C(z) = y$, and $\perp$ in the case if no such z exists. The circuit C can contain binary gates, "multiply by g" and "multiply by g^{-1}" gates, and inversion gates which may recursively use $\mathcal{O}_{\mathsf{Inv}}$ to invert some circuit C' on some output y'. Let d be the size of the circuit C, which we will define below (roughly speaking, it is the number of gates in the circuit, counted recursively), then for a query received in round i, the response will be sent at round $i + d$.

Definition 12. *The size of a circuit C containing binary gates, group multiplication gates and inversion gates is defined recursively. First, we assign costs to each gate. A binary gate has cost 0, a "multiply by g" or "multiply by g^{-1}" gate has cost 1, and an inversion gate for a circuit C has cost equal to the size of C. Then the size of the circuit is defined to be the summation of the costs of all the gates in the circuit.*

Intuitively, this definition implies for any power i, it takes at minimum i rounds to compute g^i or g^{-i} assuming that the inversion oracle $\mathcal{O}_{\mathsf{Inv}}$ isn't too useful for speeding up the computation. This intuition does turn out to be true, and we formalize and prove this later in our proof.

We formally define our construction of the sequential function as follows.

- $\mathsf{Setup}(1^\lambda)$: To set up the sequential function, we sample the cyclic group $\mathbb{G}$ with prime order $p = \Theta(2^\lambda)$ and labels as n-bit strings with $n = poly(\lambda) > 2 \log ||\mathbb{G}||$. We sample a random

generator $g \in \mathbb{G}$ and define $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{MultInv}}$ with g hard-coded in. The public parameter pp consists of the group description $\mathbb{G}$, the label length n, and access to random oracles $\mathcal{O}_{\mathsf{Mult}}$, $\mathcal{O}_{\mathsf{MultInv}}$, and $\mathcal{O}_{\mathsf{Inv}}$, as defined above.

- $\mathsf{Gen}(\mathsf{pp})$: To generate an input α for the sequential function, sample a random group element $r \in \mathbb{G}$ and the input α is the corresponding label for the group element r. Notice that if we have $r = g^x$, then we have $\alpha = \ell^{(x)}$, where $\ell^{(x)}$ denotes the label for the group element $g^x = r$.

- $\mathsf{Eval}(\mathsf{pp}, \alpha, T)$: To evaluate the sequential function, invoke the random oracle $\mathcal{O}_{\mathsf{Mult}}$ a total of T times starting from α. The output is hence $\beta = {\mathcal{O}_{\mathsf{Mult}}}^{(T)}(\alpha)$, which corresponds to the group element $r \cdot g^T$. Notice that by setting the Round function to be $\mathcal{O}_{\mathsf{Mult}}$, this is in fact a CISF by definition.

In the proof, W.L.O.G., we assume $r = g^0$ is the multiplicative identity, and hence we have $\alpha = \ell^{(0)}$ and the sequential function we compute is simply

$$\mathsf{Eval}(\mathsf{pp}, \alpha, T) = {\mathcal{O}_{\mathsf{Mult}}}^{(T)}(\ell^{(0)}) = \ell^{(T)},$$

which we will argue to be indeed sequential. We emphasize that this does not change any fundamental properties of the proof, but does allow us to ignore r in our description, considerably simplifying the presentation in some places.

5.3 Proof Outline

To show that this gives us a sequential function that is not one-way, we show that given an inversion oracle $\mathcal{O}_{\mathsf{Inv}}$ that inverts any binary circuit with additional "group multiplication" gates as well as "gates" that represent calls to itself, the function is still sequential. But now with the inversion oracle, the function can be easily inverted, and hence is not one-way. Effectively, we wish to show that for any adversary $\mathcal{A}^{\mathcal{O}_{\mathsf{Mult}}, \mathcal{O}_{\mathsf{Inv}}}(\alpha)$ that outputs $\mathcal{O}^T_{\mathsf{Mult}}(\alpha)$, $\mathcal{A}$ must still be sequential (take time T). Notice that similar to the generic group model, we never hand out the group elements directly - all the adversary sees are the labels for them. In the proof, in order to simulate the inversion oracle, we assume the existence of a PSPACE oracle that all parties (including the adversary) have access to.

We note that arguing that no one-way function can exist in this model is simple: because $\mathcal{O}_{\mathsf{Inv}}$ can be called recursively on itself and we have a PSPACE oracle, there is nothing that cannot be inverted in our model. However, arguing that our construction is sequential is substantially more complicated. We prove this using a hybrid argument, which proceeds essentially as follows:

- H_0: we start with the original pair of oracles $(\mathcal{O}_{\mathsf{Mult}}, \mathcal{O}_{\mathsf{Inv}})$. We note that this corresponds to the real game as we have defined it, and also that, as defined, it is not efficiently implementable (it would require exponential space).

- H_1: we move to an efficient simulation $(\mathcal{S}^{\mathcal{O}_{\mathsf{PSPACE}}})$ where $\mathcal{S}^{\mathcal{O}_{\mathsf{PSPACE}}}$ is a simulator with two interfaces and access to a PSPACE oracle, and it simulates both the group multiplication oracle and the inversion oracle by maintaining a list of constraints. This is an efficient (assuming a PSPACE oracle) simulation, and the (relatively straightforward) challenge in moving from H_0 to H_1 is just showing that this simulation is correct.

- H_2: we make a relatively minor change to the simulator. In addition to any queries the adversary makes, for every label β that has been returned to the adversary, we also query

$\mathcal{O}_{\mathsf{Mult}}(\beta)$ and $\mathcal{O}_{\mathsf{MultInv}}(\beta)$ if they have not yet been queried. If the adversary does not make these queries, we just store them and don't return them. Since this would be a valid way for an adversary to query, its indistinguishability from H_1 also follows from the correctness of our simulator.

- H_3: we simulate using the same simulator, but when sampling the labels for group elements g^i where $i \leq T$, we ignore the constraints that follow from queries to $\mathcal{O}_{\mathsf{Inv}}$ that result in $\perp$. We note that, by our description of the previous hybrid, these will be sampled by the simulator in the course of the game.

- H_4: we simulate using the simulator as in H_3, except that when sampling the labels for elements g^i where $i \leq T$, we query the actual group multiplication and the group multiplicative inverse oracle $(\mathcal{O}_{\mathsf{Mult}}, \mathcal{O}_{\mathsf{MultInv}})$. In other words, the values of these labels corresponding to group elements with small exponent are sampled completely independently from the rest of the simulation.

We show that for any efficient adversary, it cannot distinguish between two consecutive hybrids with non-negligible probability. And then we show that in H_4, no adversary can output $\mathcal{O}^T_{\mathsf{Mult}}(\alpha)$ in sequential time less than T. This last step is rather straightforward since only the $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{MultInv}}$ oracles are queried; we never query the $\mathcal{O}_{\mathsf{Inv}}$ oracle in the simulation. The bulk of the proof is in proving the indistinguishability of H_2 and H_3, as well as H_3 and H_4.

5.4 Hybrid Definitions

We prove the following theorem through a sequence of hybrids.

Theorem 1. *Assuming the existence of a* PSPACE *oracle, then there exists an oracle $\mathcal{O}$ and a function $f^{\mathcal{O}}$ with oracle access to $\mathcal{O}$ such that $f^{\mathcal{O}}$ is a sequential function, but not a one-way function.*

We begin the proof by defining a shortcutting experiment/game for the adversary, where the adversary wins by computing $\ell^{(T)}$ in strictly less than T rounds. To show the adversary's advantage in winning this game is negligible, we modify the experiment through a sequence of hybrid experiments, so that in the final hybrid, it is easy to argue about the adversary's negligible winning probability.

Shortcutting Game.

The adversary $\mathcal{A}$ plays the game $\mathsf{Shortcut}^{\mathcal{O}_{\mathsf{Mult}},\mathcal{O}_{\mathsf{Inv}}}_{\mathcal{A}}(\alpha, T)$, for a label $\alpha \in \{0,1\}^m$ that corresponds to the group identity and a sufficiently large exponent $T = poly(\lambda)$. It is allowed $q = poly(\lambda)$ parallel oracle access to $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{Inv}}$, i.e. it can submit up to q number of parallel queries to the oracles each round. The rule of the game is rather simple: the adversary wins the game if it outputs $\ell^{(T)}$, the label that corresponds to the group element g^T, in time less than T, and loses otherwise. Put more formally, consider the following game:

Shortcutting Game $\mathsf{Shortcut}^{\mathcal{O}_{\mathsf{Mult}},\mathcal{O}_{\mathsf{Inv}}}_{\mathcal{A}}(\alpha, T, q)$:

- For round $i = 1, 2, \ldots, T-1$:
  - At the beginning of the round, $\mathcal{A}$ receives responses from $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{Inv}}$ for queries submitted in previous rounds.

– $\mathcal{A}$ submits up to q number of parallel queries to $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{Inv}}$. Recall that $\mathcal{A}$ will receive responses from $\mathcal{O}_{\mathsf{Mult}}$ in round $i+1$, and those from $\mathcal{O}_{\mathsf{Inv}}$ in round $i+d$ where d is the size of the circuit C queried.

- At any time, $\mathcal{A}$ can terminate the game by outputting a label β. It wins the game if $\beta = \ell^{(T)}$, and loses otherwise.
- After $T-1$ rounds have expired, the adversary automatically loses the game.

Notice that we wish to show that for all adversaries, the probability of winning the shortcutting game is negligible.

Hybrid 0.

In hybrid 0, the adversary plays the original shortcutting game $\mathsf{Shortcut}_{\mathcal{A}}^{\mathcal{O}_{\mathsf{Mult}},\mathcal{O}_{\mathsf{Inv}}}(\alpha, T, q)$.

Hybrid 1.

In Hybrid 1, we will simulate $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{Inv}}$ for the adversary using a simulator $\mathcal{S}^{\mathcal{O}_{\mathsf{PSPACE}}}$. The simulator has two interfaces (and hence can answer queries to both oracles) and operates in the following manner:

- $\mathcal{S}$ maintains a list of constraints. The initial constraint is that α is the label for the multiplicative identity. We denote this constraint as $\mathcal{INIT}$. For every query that is responded to with a non-$\perp$ output, we add appropriate LABEL constraints which takes the form of a pair of labels (α_1, α_2), meaning that $x^{(\alpha_1)} = x^{(\alpha_2)} + 1$, i.e. the group element that corresponds to α_1 is equal to the generator g multiplied by the group element that corresponds to α_2. We denote the entire set of LABEL constraints as $\mathcal{LABEL}$. For queries that have a $\perp$ response, we add NEVER constraints, indicating that a circuit is not satisfiable, which we specify in a moment. Similarly, we denote the set of NEVER constraints as $\mathcal{NEVER}$.
- To answer a query for $\mathcal{O}_{\mathsf{Mult}}(\alpha_1)$, construct the following two PSPACE-relations:
  – R_1: There is an assignment of labels to all group elements in $\mathbb{G}$ that satisfies all the constraints (i.e. $\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER}$) and that α_1 is a valid label.
  – R_2: There is an assignment of labels to all group elements in $\mathbb{G}$ that satisfies all the constraints.

  Let a be the number of solutions for the PSPACE-relation R_1, and b be the number of solutions for R_2. Then:

  – With probability $1 - a/b$, output $\alpha_2 = \perp$, and record a NEVER constraint that $\mathcal{O}_{\mathsf{Mult}}(\alpha_1) = \perp$, meaning that α_1 should *never* be a valid label.
  – With probability a/b, output α_2 that is computed in the following way. First, use a uniform generator for R_1 based on Lemma 2 to sample a satisfying solution for R_1. Using the assignment of labels in this solution, first look up the group element with exponent $x^{(\alpha_1)}$ that corresponds to the label α_1, and then find the label α_2 that corresponds to the group element with exponent $x^{(\alpha_1)} + 1$. Notice that we essentially have $\alpha_2 = \ell^{(x^{(\alpha_1)}+1)}$. Record (α_1, α_2) in the list of LABEL constraints.

At a first glance, the response to multiplication queries may seem unnecessarily complex. Why do we need to do more than just check if an input label is valid and just use the existing LABEL constraints naturally? However, we emphasize that asking the oracle to multiply invalid labels creates NEVER constraints, and there can be seemingly complicated interplay between NEVER constraints and asking the multiplication oracle to multiply things that are not confirmed to be valid labels. So we unfortunately need a rather complicated description here.

- To answer a query for $\mathcal{O}_{\mathsf{Inv}}(C, y)$, construct the following two PSPACE-relations:
  - R_1: There is an assignment of labels to all group elements in $\mathbb{G}$ that satisfies all the constraints and that there exists an input z s.t. $C(z) = y$.
  - R_2: There is an assignment of labels to all group elements in $\mathbb{G}$ that satisfies all the constraints.

  Notice that here, perhaps a bit counter-intuitively, we are still counting the number of possible label assignments for the group elements, instead of possible assignments of wire values. This is because the distribution of whether a circuit is satisfiable should also be dependent on the previous label assignments, not just the circuit itself.

  Let a be the number of solutions for the PSPACE-relation R_1, and b be the number of solutions for R_2. Then:

  - With probability $1-a/b$, output $z = \perp$, and record a NEVER constraint that $\mathcal{O}_{\mathsf{Inv}}(C, y) = \perp$, meaning that the inversion query on (C, y) should *never* be satisfiable.
  - With probability a/b, output z that is computed in the following way. First, use a uniform generator for R_1 based on Lemma 2 to sample a satisfying solution for R_1. Then use the PSPACE oracle again to generate a solution for the following PSPACE-relation: under the assignment of labels in the solution for R_1, there exists an assignment of values to input and output wires in C such that the output of the circuit is y. Using this solution, simply output z as the value in the input wire to the circuit. To maintain the list of constraints, run the circuit C with z as input. For each "multiply by g" gate in C with input wire α_1 and output wire α_2, record (α_1, α_2) in the list of constraints. For each "multiply by g^{-1}" gate with input wire α_1 and output wire α_2, record (α_2, α_1) in the list of LABEL constraints.

Hybrid 2.

In Hybrid 2, we simulate $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{Inv}}$ using the same simulator as in H_1, with a slight twist. Recall that the simulator $\mathcal{S}$ maintains a list of constraints consisting of $\mathcal{INIT}, \mathcal{LABEL}$ and $\mathcal{NEVER}$, and these constraints are added at the beginning of the experiment ($\mathcal{INIT}$) or when answering an oracle query ($\mathcal{LABEL}$ and $\mathcal{NEVER}$). In hybrid 2, we will "predict" which constraints will be added later, and we sample them proactively at the beginning of each round. Notice that we are only sampling these constraints behind the scenes in the simulator, the interaction with the adversary remains the same as in hybrid 2 in that we don't send any additional things to the adversary. With this "proactive sampling" technique, we will add some LABEL constraints at the beginning of each round, which we denote as $\mathcal{AUTO}^*$ and $\mathcal{AUTO}$, and we will explain the difference between them in a moment. Whenever we sample a label, it will now be conditioned on $\mathcal{INIT}, \mathcal{LABEL}, \mathcal{NEVER}, \mathcal{AUTO}^*$ and $\mathcal{AUTO}$.

With proactive sampling, for any label α_1 that the adversary has, we "predict" that since the adversary already has α_1, it might query $\mathcal{O}_{\mathsf{Mult}}(\alpha_1)$ or $\mathcal{O}_{\mathsf{MultInv}}(\alpha_1)$ in the future, so we proactively sample the constraints we would get if the adversary were to make these two queries, which essentially requires us sampling the labels $\ell^{(x^{(\alpha_1)}+1)}$ and $\ell^{(x^{(\alpha_1)}-1)}$.

Put more concretely, at the beginning of round i, for any label α_1 that appears in $\mathcal{INIT}, \mathcal{LABEL}$, $\mathcal{AUTO}^*$, or $\mathcal{AUTO}$ constraints, we first check whether $\ell^{(x^{(\alpha_1)}+1)}$ and $\ell^{(x^{(\alpha_1)}-1)}$ haven been sampled yet. If they are already sampled, we make sure to not resample them, as that will cause a collision in the label-to-group-element mapping. If either of them hasn't been sampled yet, say $\ell^{(x^{(\alpha_1)}+1)}$ for instance, we sample it using the PSPACE oracle, conditioned on $\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. Notice that this label can be though of as sampled uniformly (conditioned on all existing constraints) as $\alpha_2 = \ell^{(x^{(\alpha_1)}+1)} \sim L^{(x^{(\alpha_1)}+1)} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$.

If $x^{(\alpha_1)} = i - 1$, this means $\alpha_1 = \ell^{(i-1)}$, and $\alpha_2 = \ell^{(i)}$, so they are the labels one would get if one were to follow the honest execution trace by iteratively querying $\mathcal{O}_{\mathsf{Mult}}$ starting with the initial input α. For these α_1's, we add the label constraint (α_1, α_2) to the set of $\mathcal{AUTO}^*$ constraints. For all other α_1's, we add the label constraint (α_1, α_2) to the set of $\mathcal{AUTO}$ constraints.

It works similarly for $\ell^{(x^{(\alpha_1)}-1)}$. We sample $\alpha_3 = \ell^{(x^{(\alpha_1)}-1)} \sim L^{(x^{(\alpha_1)}-1)} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$, and if $x^{(\alpha_1)} = -i + 1$, we add the constraint (α_3, α_1) to $\mathcal{AUTO}^*$; otherwise, we add it to $\mathcal{AUTO}$.

The rest of the simulator works exactly as in H_1. Notice that the constraints in $\mathcal{AUTO}^*$ and $\mathcal{AUTO}$ are sampled in the same way, so for now they are just different categorizations depending on which group elements the labels correspond to. Namely, if the labels in the constraints correspond to $g^{-i}, g^{-i+1}, \ldots, g^{-1}, g^0, g^1, \ldots, g^i$, i.e. the group elements in the honest execution trace, they are put in $\mathcal{AUTO}^*$ [1]; all other constraints go to $\mathcal{AUTO}$. In the future hybrids, we will further modify how the constraints in $\mathcal{AUTO}^*$ are sampled, which the main reason why we separate them out.

Hybrid 3

In Hybrid 3 we modify the way the $\mathcal{AUTO}^*$ constraints in H_2 are sampled.

Recall that in hybrid 2, for the $\mathcal{AUTO}^*$ constraints, we sample $\alpha_2 = \ell^{(i)}$ and $\alpha_3 = \ell^{(-i)}$ at round i, and we sample them uniformly as $\ell^{(i)} \sim L^{(i)} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$ and $\ell^{(-i)} \sim L^{(-i)} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. In H_3, we will sample them as $\ell^{(i)} \sim L^{(i)} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$ and $\ell^{(-i)} \sim L^{(-i)} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. Notice the difference is that we sample them no longer conditioned on the NEVER constraints.

This is the only change we make in this hybrid. For constraints in $\mathcal{AUTO}$, they are still sampled conditioned on $\mathcal{NEVER}$ and the rest of the simulator works exactly as in H_2.

Hybrid 4

In Hybrid 4 we further modify the way these $\mathcal{AUTO}^*$ constraints in H_3 are sampled. Specifically, in H_3, at round i, we sample the labels $\ell^{(i)}$ and $\ell^{(-i)}$ from appropriate label distributions, conditioned on $\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. Now, we will generate these constraints by querying the actual $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{MultInv}}$ oracles. Specifically, in H_4, we maintain the $\mathcal{AUTO}^*$ constraints in the following way.

[1] Another way to think about the $\mathcal{AUTO}^*$ constraints is that they are what one will get if running proactive sampling starting with just the $\mathcal{INIT}$ constraint.

- In round $i-1$ for $i = 1, 2, 3, \ldots, T$, query $\ell^{(i)} \leftarrow \mathcal{O}_{\mathsf{Mult}}(\ell^{(i-1)})$ and $\ell^{(-i)} \leftarrow \mathcal{O}_{\mathsf{MultInv}}(\ell^{(-i+1)})$.
- At the beginning of round i, add the LABEL constraints $(\ell^{(i-1)}, \ell^{(i)})$, $(\ell^{(-i)}, \ell^{(-i+1)})$ to the set of $\mathcal{AUTO}^*$.

The rest of the simulator works exactly as in H_3. Notice that in this hybrid, everything is simulated except for the queries to $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{MultInv}}$.

5.5 Proof of Hybrid Arguments

Lemma 4. *H_0 and H_1 are distributed in a statistically identical way.*

Proof. This lemma just boils down to the correctness of our initial simulation in H_1. By inspecting the construction of the simulator in Hybrid 1, we can see that from the adversary's point of view, the behavior of Mult and Inv queries are exactly the same in both hybrids. Specifically, the probability of answering $\perp$ or non-$\perp$ are computed correctly according to the overall number of possible label assignments as in the PSPACE relations R_1 and R_2, and if non-$\perp$ answers are provided, we sample them correctly using a uniform PSPACE-relation generator, which we show can be constructed using lemma 3.

Essentially, in H_1, when we sample the labels when answering a query, we sample them from the distributions $\{L^{(i)}\}_{i\in[p]}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER}$. Notice that this gives a uniform distribution conditioned on all previous constraints, and therefore query results. And by the definition of the oracles, in H_0, the labels are also sampled uniformly conditioned on prior results. Hence H_0 and H_1 are statistically indistinguishable. □

Lemma 5. *H_1 and H_2 are distributed in a statistically identical way.*

Proof. Notice that the simulator in H_2 behaves exactly like the simulator in H_1 except for the extra $\mathcal{AUTO}^*$ and $\mathcal{AUTO}$ constraints. In H_1, the label for some group element $g^{(x)}$ is sampled from the distribution $L^{(x)}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER}$, while in H_2, it is sampled from the distribution $L^{(x)}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. We wish to argue that these two distributions are identical.

Here we utilize the correctness of the PSPACE oracle that is used to sample the labels. In H_2, we are simply sending additional queries to the PSPACE oracle, but each subsequent query will be conditioned on all of the previous queries. We can think about this process as sampling an injective mapping from group elements to labels. In H_1, we directly sample the mapping for group element g^x, while in H_2, we sample some other mappings first, and then sample the mapping for g^x *conditioned on the previously sampled mappings*. By the chain rule of conditional probability, these two sampling methods yield the same distribution as desired.

Alternatively, we can think about H_2 as another instance of the simulation with a different set of oracle queries (which now includes those proactive sampling queries). This is just another valid sequence of queries to make, and given the correctness of the simulation in H_1, the resulting distributions should still be uniform, and hence indistinguishable from H_1.

□

Lemma 6. *No adversary can distinguish between H_2 and H_3 with non-negligible probability.*

Proof. The difference between H_2 and H_3 is how the $\mathcal{AUTO}^*$ constraints are sampled. In H_2, they are sampled from $L^{(i)}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. In H_3, they are sampled from $L^{(i)}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. To show that H_2 and H_3 are indistinguishable for the adversary, we will simply argue that with all but negligible probability, we have

$$\{L^{(i)}\}_{i\in[-T,T]}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO} \approx_\epsilon \{L^{(i)}\}_{i\in[-T,T]}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO},$$

where $\approx_\epsilon$ denotes the two distribution are statistically close. Due to the complexity of the argument, we defer the proof of this statement to section 5.6.

Assuming the correctness of the statement, it should be easy to see that H_2 and H_3 are indistinguishable, as the only difference between the hybrids is the distribution of $\mathcal{AUTO}^*$ constraints. If an adversary is able to distinguish between H_2 and H_3 with non-negligible probability, then it essentially distinguishes the two distributions with non-negligible probability, which contradicts with our assumption of the statement being correct. □

Lemma 7. *No adversary that can distinguish between H_3 and H_4 with non-negligible probability.*

Proof. The only difference between H_3 and H_4 is the way the $\mathcal{AUTO}^*$ constraints are generated. In H_3, the labels are randomly sampled conditioned on $\mathcal{INIT}, \mathcal{LABEL}, \mathcal{AUTO}^*$ and $\mathcal{AUTO}$, while in H_4, the labels are obtained by directly querying the multiplication and multiplicative inverse oracle. Notice that the actual oracle samples the labels directly from the distribution $L^{(x)}$, without conditioning on $\mathcal{INIT}, \mathcal{LABEL}$, and $\mathcal{AUTO}$. It does sample the labels conditioned on $\mathcal{AUTO}^*$ itself though, as the oracle is consistent and won't give back the same label for different group elements or different labels for the same group element. So here, we wish to show that, with all but negligible probability,

$$\{L^{(i)}\}_{i\in[-T,T]}|\mathcal{AUTO}^* \approx_\epsilon \{L^{(i)}\}_{i\in[-T,T]}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}.$$

We defer the proof of this statement to section 5.6. □

Theorem 1. *Assuming the existence of a* PSPACE *oracle, then there exists an oracle $\mathcal{O}$ and a function $f^{\mathcal{O}}$ with oracle access to $\mathcal{O}$ such that $f^{\mathcal{O}}$ is a sequential function, but not a one-way function.*

Proof. The lemmas above show a sequence of a constant number of hybrid experiments where no adversary can distinguish one from the next with non-negligible probability. Notice that the first hybrid H_0 corresponds to the adversary playing the original shortcutting game, and the last hybrid H_4 corresponds to the adversary playing the shortcutting game, but interacting with simulated oracles.

In H_4, the simulator simulates both $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{Inv}}$ oracles correctly, but only requires access to $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{MultInv}}$. In other words, if an adversary can win the game in H_4, then it might as well simulate $\mathcal{O}_{\mathsf{Inv}}$ for itself using the simulator in H_4, and still win the game. But then, the only information that the adversary can gather about powers of g are through querying the $\mathcal{O}_{\mathsf{Mult}}$ and $\mathcal{O}_{\mathsf{MultInv}}$ oracles. Notice that multiplying by g in the group has a cycle length $p > T$, so using these two oracles, $\ell^{(T)}$ can only be computed as early as in round T, causing the adversary to lose the

game. Therefore, we have shown an oracle $\mathcal{O}_{\mathsf{Mult}}$ that gives a sequential function $f^{\mathcal{O}_{\mathsf{Mult}}}(\alpha) = \ell^{(T)}$ that is not one-way.

□

5.6 Proof of Lemma 6 and Lemma 7

In this subsection, we present the missing parts of the proof for Lemma 6 and Lemma 7. For these two lemmas, one important proof technique that we will utilize is to analyze the distribution of $X^{(\ell)}$ for some given label ℓ. Recall that $x^{(\ell)}$ denotes the group element that corresponds to the label ℓ, and here, we will use the upper case $X^{(\ell)}$ to denote the distribution from which $x^{(\ell)}$ is sampled.

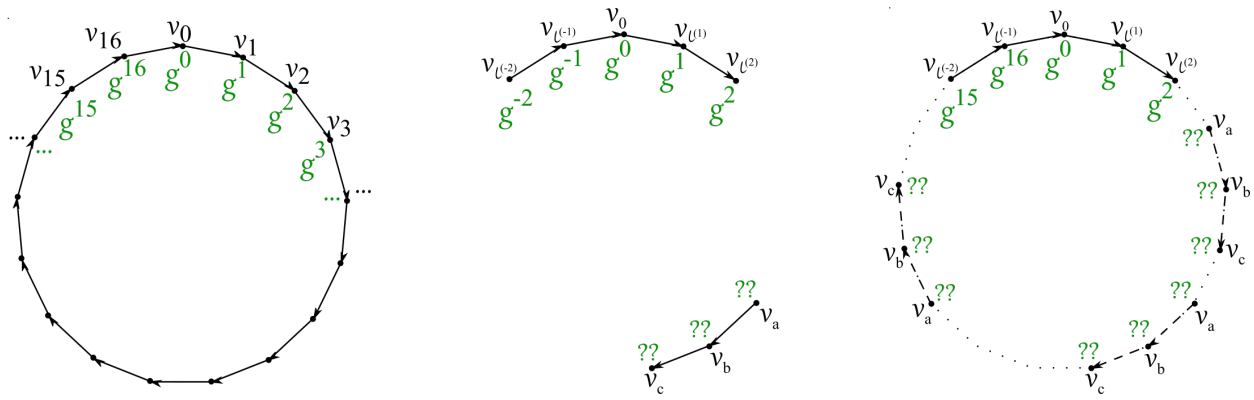
One way to visualize the "group element" distribution $X^{(\ell)}$ is through a *constraint graph*, which we will describe in a moment. But first, let's imagine the following *ideal graph* for the given group $\mathbb{G}$ and random oracle $\mathcal{O}_{\mathsf{Mult}}$. The graph has p nodes where $p = \Theta(2^\lambda)$ is the order of the group, and each node corresponds to a label of a group element in $\mathbb{G}$. The edges are added by repeatedly querying $\mathcal{O}_{\mathsf{Mult}}$ starting from the label for the group identity g^0. For each query $\alpha_2 = \mathcal{O}_{\mathsf{Mult}}(\alpha_1)$, a directed edge is added from the node for label α_1 to the node for label α_2. Repeat this process until we get back at the label $\ell^{(0)}$ for g^0. Notice that by the property of the group multiplication, the ideal graph has the shape of a *cycle* with length p.

Now, we describe how to organize the constraints $\mathcal{INIT}, \mathcal{LABEL}, \mathcal{AUTO}^*$ and $\mathcal{AUTO}$ into a *constraint graph*, which is a subgraph of the ideal graph. First, for the constraint $\mathcal{INIT}$ (i.e. the label for group identity is α), we add a node v_0 that corresponds to the label α. Then, for the $\mathcal{LABEL}, \mathcal{AUTO}^*$ and $\mathcal{AUTO}$ constraints of the form (α_1, α_2), we add nodes $v_{\alpha_1}, v_{\alpha_2}$, and a directed edge $(v_{\alpha_1}, v_{\alpha_2})$. Notice that since the constraint graph is a subgraph of a cycle, the connected components of the constraint graph corresponds to *arcs* on the cycle. We denote an arc as $(u_1, u_2, \ldots, u_k)$ for nodes $u_1, \ldots, u_k$ and edges $(u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k)$. Now imagine overlaying these arcs onto the ideal graph. Notice that the *location* of an arc in the ideal graph depends on the discrete log of its nodes labels w.r.t. g, which is precisely the distribution $X^{(\ell)}$ for the labels in the constraints that form the arc. Since we know that v_0 corresponds to $1 = g^0$, the location of the arc containing v_0 is known and fixed. Let us call it the "good arc". However, for any "bad arc" that does not contain v_0, where should we overlay it in the ideal graph? It turns out that, ignoring NEVER constraints, its location is uniformly distributed along the entire cycle, minus the good arc portion containing v_0. Translating what this means for the distributions $X^{(\ell)}$ for all the labels ℓ in the bad arc, they should be uniform distributions in $[0, p]$, conditioned on the bad arc not intersecting with the good arc. For an illustration of the ideal graph, the constraint graph and how they overlay, see Figure 2.

We note that this constraint graph ignores NEVER constraints, but we will show that this is not so important in our analysis. We first show a useful lemma explaining that NEVER constraints don't actually reduce the number of possible assignments on the constraint graph all that much.

Lemma 8. *Any NEVER constraint (i.e. a response from the oracle $\mathcal{O}_{\mathsf{Inv}}$ that returns $\perp$) reduces the number of possible label assignments by no more than a $\frac{1}{poly(\lambda)}$ multiplicative factor with all but negligible probability, i.e. let $\{L^{(i)}\}_{i\in[p]}$ be the distributions of label assignments for all group element conditioned on all prior constraints, and* never *be an added NEVER constraint. We have that with overwhelming probability,*

$$\frac{\left|\mathsf{supp}(\{L^{(i)}\}_{i\in[p]}|\mathsf{never})\right|}{\left|\mathsf{supp}(\{L^{(i)}\}_{i\in[p]})\right|} \geq \frac{1}{poly(\lambda)}.$$



(a) Ideal Graph (b) Constraint Graph (c) Constraint Graph overlaid onto Ideal Graph

Figure 2: Illustration of an ideal graph, a constraint graph, and the constraint graph overlaid onto the ideal graph for a group with order $p = 17$. The green texts denote the group element to which the labels at each node correspond. Notice that when overlaying the constraint graph, only the location of the "good arc" is fixed and is centered around v_0 since the discrete logs of its group elements are known. The other arc with unknown group elements is equally likely to be anywhere else on the cycle.

Proof. Consider an inversion query on circuit C and output y. Let n be the number of label assignments that satisfy all the pre-existing constraints, and m be the number of label assignments that satisfy all the pre-existing constraints *and* that the circuit C is satisfiable for output y. Notice that the probability of outputting $\perp$ on this query is $1 - m/n$. So if the adversary wants to obtain a stronger NEVER constraint by reducing a large number of possible label assignments, it would want to query (C, y) such that m is large, but doing so would cause the probability of actually getting the NEVER constraint to go down. In order words, the more powerful a NEVER constraint is, the less likely it is going to be added. Notice that by adding a NEVER constraint, we are essentially ruling out the subset of m label assignments that have satisfying inputs from all n possible label assignments. But this happens with only probability $1 - m/n$. So on expectation, every time we make an inversion query in the hope of getting a NEVER constraint, we rule out $m(1 - m/n) \leq n/4$ out of the total n possible label assignments.

To put this more concretely, in order to reduce the n number of possible label assignments down to $n/\mathit{superpoly}(\lambda)$, we would set $m = (1 - 1/\mathit{superpoly}(\lambda))n$. But in this case, we will only output a NEVER constraint with probability $1 - m/n = 1/\mathit{superpoly}(\lambda)$ which is negligible. Therefore, any NEVER constraint can only reduce the number of possible label assignments by no more than a $\frac{1}{\mathit{poly}(\lambda)}$ multiplicative factor with overwhelming probability. □

This above lemma says that with overwhelming probability, a NEVER constraint can only reduce the number of *all* label assignments by a $\mathit{poly}(\lambda)$ factor, i.e. give at most $O(\log \lambda)$ bits of information on all the label distributions. But perhaps a sophisticated adversary can concentrate this information on a specific arc of its choice, and therefore change the label distributions on that arc in a non-negligible way. In the following lemma, we show that this is also not possible. Recall that I denotes the *mutual information* between two random variables. We have:

Lemma 9. *Let $a, b \in [p]$ s.t. $a < b$, $\{L^{(i)}\}_{i\in[a,b]}$ be the distributions of label assignments for the arc $(v_{\ell^{(a)}}, v_{\ell^{(a+1)}}, \ldots, v_{\ell^{(b)}})$ of length $b-a = poly(\lambda)$, $\mathcal{C}$ be the collection of prior non-NEVER constraints with size $|\mathcal{C}| = poly(\lambda)$ (includes everything in $\mathcal{INIT}, \mathcal{LABEL}, \mathcal{AUTO}^*, \mathcal{AUTO}$), and* never *be an added NEVER constraint. We have that with overwhelming probability,*

$$I\left(\mathsf{never}; \{L^{(i)}\}_{i\in[a,b]} \middle| \mathcal{C}\right) = negl(\lambda).$$

Proof. First, by Lemma 8, with overwhelming probability,

$$\left|\mathsf{supp}\left(\{L^{(i)}\}_{i\in[p]} \middle| \mathcal{C} \wedge \mathsf{never}\right)\right| \geq \frac{1}{poly(\lambda)} \cdot \left|\mathsf{supp}\left(\{L^{(i)}\}_{i\in[p]} \middle| \mathcal{C}\right)\right|.$$

Notice that this implies the mutual information between never and all the label distributions $\{L^{(i)}\}_{i\in[p]}$ conditioned on all the prior constraints is bounded by

$$I\left(\mathsf{never}; \{L^{(i)}\}_{i\in[p]} \middle| \mathcal{C}\right) \leq -\log \frac{1}{poly(\lambda)} = O(\log \lambda).$$

Since we have an exponential number of labels, this means that the distribution of each individual label is very unlikely to be changed significantly by a NEVER constraint. But Lemma 8 only gives us this for the label distributions for all group elements $\{L^{(i)}\}_{i\in[p]}$, not specifically for the label distributions $\{L^{(i)}\}_{i\in[a,b]}$ that we needed. For instance, although the adversary may only reduce the total number of possible label assignments by a small polynomial factor, but it might target particularly the labels in the arc $[a, b]$ and cause the label distributions there to change significantly. We argue that this is not possible.

To impact the label distributions in the arc $[a, b]$, the NEVER constraint never would need to change the distribution of labels near existing arcs by depending on one of the existing arcs. Otherwise, if never is independent from all existing arcs, i.e. constraints, we will have that the mutual information between never and each unsampled label distribution conditioned on $\mathcal{C}$ is the same. Specifically, let $\mathcal{L}_\mathcal{C}$ denote the set of labels that appear in $\mathcal{C}$. If we assume that never is independent from all labels in $\mathcal{L}_\mathcal{C}$, then for all $i \in [p]$ s.t. $\ell^{(i)} \notin \mathcal{L}_\mathcal{C}$, $I(\mathsf{never}; L^{(i)}|\mathcal{C})$ is the same. Therefore, we have

$$\begin{aligned}
I\left(\mathsf{never}; \{L^{(i)}\}_{i\in[a,b],\ell^{(i)}\notin\mathcal{L}_\mathcal{C}} \middle| \mathcal{C}\right) &= I\left(\mathsf{never}; \{L^{(i)}\}_{i\in[a,b]} \middle| \mathcal{C}\right) \\
&\leq \frac{b-a}{p - |\mathcal{L}_\mathcal{C}|} O(\log \lambda) \\
&= \frac{poly(\lambda)}{2^\lambda - poly(\lambda)} O(\log \lambda) \\
&= O(2^{-\lambda} poly(\lambda) \log \lambda) = negl(\lambda),
\end{aligned}$$

meaning its impact on the label distributions between $[a, b]$ will be negligible.

Now we consider the case where the circuit in never actually depends on some other existing constraints/arcs. Let d be the size of the circuit in the never constraint. Notice that a circuit of size d can only impact the distributions of labels at most distance d away from the constraints that the circuit is dependent on. Let's denote the set of these label distributions as $\mathcal{L}_d$. But such a query will not be answered until d rounds later, and recall that in each round, we automatically extend all

the known labels with $\mathcal{AUTO}^*$ and $\mathcal{AUTO}$. With this proactive sampling, whatever constraints the circuit depends on when the query was submitted d rounds ago, their corresponding arcs are extended in both directions by d, and this matches exactly $\mathcal{L}_d$, the collection of label distributions that the never constraint might be able to impact. Therefore, for all the label distributions in $\mathcal{L}_d$ that the never constraint might affect, they have already been fully sampled and hence have zero entropy. Therefore, trivially we have

$$I(\mathsf{never}; \mathcal{L}_d | \mathcal{C}) \leq H(\mathcal{L}_d | \mathcal{C}) = 0.$$

Another intuitive way to think about this is that since the labels in $\mathcal{L}_d$ are sampled *after* the adversary submits the inversion query, the circuit in the never constraint must be independent from these labels and therefore the never constraint cannot possibly give any information about these labels.

Now that we know never constraint cannot give any information on arcs around existing constraints, we fall back to the same scenario as before: the mutual information between never and each label distribution $I(\mathsf{never}; L^{(i)} | \mathcal{C})$ is the same for all $i \in [p]$ s.t. $\ell^{(i)} \notin \mathcal{L}_{\mathcal{C}}$, and we have already shown that in this case

$$I\left(\mathsf{never}; \{L^{(i)}\}_{i\in[a,b]} \middle| \mathcal{C}\right) = O(2^{-\lambda} poly(\lambda) \log \lambda) = negl(\lambda).$$

Bringing these two parts together, we have shown that in either case, with overwhelming probability, the mutual information between never and $\{L^{(i)}\}_{i\in[a,b]}$ conditioned on $\mathcal{C}$ is negligible as desired. □

With this stronger lemma in hand, we will proceed to prove the missing part for Lemma 6.

Lemma 10. *With all but negligible probability,*

$$\{L^{(i)}\}_{i\in[-T,T]} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{NEVER} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO} \approx_\epsilon \{L^{(i)}\}_{i\in[-T,T]} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO},$$

where $\approx_\epsilon$ denotes the two distribution are statistically close.

Proof. We prove this argument by induction on the NEVER constraints. We first consider when the first NEVER constraint is added. Let's say it is in round k and we denote never as the random variable for the first NEVER constraint, we will show that with overwhelming probability

$$\{L^{(i)}\}_{i\in[-T,T]} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO} \approx_\epsilon \{L^{(i)}\}_{i\in[-T,T]} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathsf{never} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}.$$

Notice that $\{L^{(i)}\}_{i\in[-T,T]} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$ corresponds to the possible label assignments for $g^{-T}, \ldots, g^T$ under all previous constraints (notice that $\mathcal{INIT}, \mathcal{LABEL}, \mathcal{AUTO}^*, \mathcal{AUTO}$ are precisely all the constraints so far), and $\{L^{(i)}\}_{i\in[-T,T]} | \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathsf{never} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$ corresponds to the possible label assignments for $g^{-T}, \ldots, g^T$ after we add the NEVER constraint never.

We now apply Lemma 9 with $a = -T, b = T, \mathcal{C} = \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. Notice that as required, $b - a = 2T = poly(\lambda)$ and the total number of constraints in $\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$ is indeed bounded by $poly(\lambda)$ since there are only $poly(\lambda)$ number rounds and in each round we can generate at most $poly(\lambda)$ number of constraints. By Lemma 9, with overwhelming probability,

$$I\left(\mathsf{never}; \{L^{(i)}\}_{i\in[-T,T]} \middle| \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}\right) = O(2^{-\lambda} poly(\lambda)\log\lambda) = negl(\lambda).$$

Therefore we have, with overwhelming probability,

$$\begin{aligned}
&\left|\mathsf{supp}\left(\{L^{(i)}\}_{i\in[-T,T]} \middle| \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathsf{never} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}\right)\right| \\
\geq& 2^{-O(2^{-\lambda} poly(\lambda)\log\lambda)} \cdot \left|\mathsf{supp}\left(\{L^{(i)}\}_{i\in[-T,T]} \middle| \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}\right)\right| \\
\geq& (1 - negl(\lambda)) \cdot \left|\mathsf{supp}\left(\{L^{(i)}\}_{i\in[-T,T]} \middle| \mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}\right)\right|.
\end{aligned}$$

Recall that these two distributions are uniformly random on their supports, therefore their statistical distance is given by

$$\begin{aligned}
\Delta(\mathcal{L}_{\mathsf{never}}, \mathcal{L}) &= \frac{1}{2} \sum_{\{\ell^{(i)}\in\{0,1\}^n\}_{i\in[-T,T]}} \left|\Pr\left[\mathcal{L}_{\mathsf{never}} = \{\ell^{(i)}\}_i\right] - \Pr\left[\mathcal{L} = \{\ell^{(i)}\}_i\right]\right| \\
&= \frac{1}{2}\left(\sum_{\{\ell^{(i)}\}_i \in \mathsf{supp}(\mathcal{L}) \cap \mathsf{supp}(\mathcal{L}_{\mathsf{never}})} \left|\Pr\left[\mathcal{L}_{\mathsf{never}} = \{\ell^{(i)}\}_i\right] - \Pr\left[\mathcal{L} = \{\ell^{(i)}\}_i\right]\right|\right. \\
&\quad \left. + \sum_{\{\ell^{(i)}\}_i \in \mathsf{supp}(\mathcal{L}) \setminus \mathsf{supp}(\mathcal{L}_{\mathsf{never}})} \left|\Pr\left[\mathcal{L}_{\mathsf{never}} = \{\ell^{(i)}\}_i\right] - \Pr\left[\mathcal{L} = \{\ell^{(i)}\}_i\right]\right|\right) \\
&= \frac{1}{2}\left(\sum_{\{\ell^{(i)}\}_i \in \mathsf{supp}(\mathcal{L}) \cap \mathsf{supp}(\mathcal{L}_{\mathsf{never}})} \left|\left|\frac{1}{\mathsf{supp}(\mathcal{L}_{\mathsf{never}})}\right| - \left|\frac{1}{\mathsf{supp}(\mathcal{L})}\right|\right|\right. \\
&\quad \left. + \sum_{\{\ell^{(i)}\}_i \in \mathsf{supp}(\mathcal{L}) \setminus \mathsf{supp}(\mathcal{L}_{\mathsf{never}})} \left|\frac{1}{\mathsf{supp}(\mathcal{L})}\right|\right) \\
&\leq \frac{1}{2}\left(2^{-O(2^{-\lambda} poly(\lambda)\log\lambda)} \cdot 2^{2\lambda T} \cdot \left(2^{-2\lambda T} \cdot 2^{-O(2^{-\lambda} poly(\lambda)\log\lambda)} - 2^{-2\lambda T}\right)\right. \\
&\quad \left. + (1 - 2^{-O(2^{-\lambda} poly(\lambda)\log\lambda)}) \cdot 2^{2\lambda T} \cdot 2^{-2\lambda T}\right) \\
&\leq \frac{1}{2}\left(2^{-O(2^{-\lambda} poly(\lambda)\log\lambda)} \cdot (2^{-O(2^{-\lambda} poly(\lambda)\log\lambda)} - 1) + (1 - 2^{-O(2^{-\lambda} poly(\lambda)\log\lambda)})\right) \\
&= \frac{1}{2}\left(1 - 2^{-O(2^{-\lambda} poly(\lambda)\log\lambda)}\right)^2 \\
&= negl^2(\lambda),
\end{aligned}$$

where we use $\mathcal{L}_{\mathsf{never}}$ to denote the distribution $\{L^{(i)}\}_{i\in[-T,T]}|\mathcal{INIT}\wedge\mathcal{LABEL}\wedge\mathsf{never}\wedge\mathcal{AUTO}^*\wedge\mathcal{AUTO}$ and $\mathcal{L}$ to denote $\{L^{(i)}\}_{i\in[-T,T]}|\mathcal{INIT}\wedge\mathcal{LABEL}\wedge\mathcal{AUTO}^*\wedge\mathcal{AUTO}$. As we can see, with overwhelming probability of $(1-\mathit{superpoly}(\lambda))$, the statistical distance between these two distributions are indeed negligible as desired.

Notice that here $\mathcal{L}_{\mathsf{never}}$ and $\mathcal{L}$ correspond to label *distributions*, not particular sampled labels. For particular sampled labels, the statement could be false. For instance, if the never constraint ruled out a particular label, and the sampled label happened to be this particular label, then we do not have the statistical closeness for the sampled label as it would never be sampled if conditioned on the never constraint. But if we take the *distribution* of the label over the randomness of the sampling procedure, such instances happens with only negligible probability, and these two label *distributions* are indeed statistically close.

To extend the above argument to multiple NEVER constraints, we apply the inductive argument. Given the $\mathcal{INIT}, \mathcal{LABEL}, \mathcal{AUTO}^*$ and $\mathcal{AUTO}$ constraints and a list of NEVER constraints, for each added NEVER constraint, by the same argument above, with overwhelming probability the label distributions are statistically close whether conditioned on the NEVER constraint or not. Notice that this can be repeated up to a polynomial times (and hence allowing up to polynomial number of NEVER constraints). The number of NEVER constraints is indeed bounded by $\mathit{poly}(\lambda)$, as there are at most $T = \mathit{poly}(\lambda)$ rounds in the game and each round the adversary can submit at most $q = \mathit{poly}(\lambda)$ number of queries. Therefore, by union bound and triangular inequality of statistical distance, with an overwhelming probability of $(1 - \mathit{poly}(\lambda)\cdot\mathit{superpoly}(\lambda))$, the statistical distance between $\{L^{(i)}\}_{i\in[-T,T]}|\mathcal{INIT}\wedge\mathcal{LABEL}\wedge\mathcal{NEVER}\wedge\mathcal{AUTO}^*\wedge\mathcal{AUTO}$ and $\{L^{(i)}\}_{i\in[-T,T]}|\mathcal{INIT}\wedge\mathcal{LABEL}\wedge\mathcal{AUTO}^*\wedge\mathcal{AUTO}$ is bounded by $\mathit{poly}(\lambda)\mathit{negl}^2(\lambda) = \mathit{negl}(\lambda)$. Therefore, even with a polynomial number of NEVER constraints, with overwhelming probability, these two label distributions are indeed statistically close as desired. □

To finish the proof for Lemma 7, we present the following three lemmas on the location and the length of the arcs in the constraint graph.

Lemma 11. *If an arc generated in our simulation does not contain the node v_0, i.e., it is a "bad arc", then its location in the ideal graph is statistically close to the uniform distribution along the entire cycle, conditioned on not intersecting with the good arc. Specifically, at round i, notice that given $\mathcal{INIT}$ and $\mathcal{AUTO}^*$, the good arc takes the form $(\ell^{(-i)}, \ell^{(-i+1)}, \ldots, \ell^{(i-1)}, \ell^{(i)})$. Let α be the label at the center of the bad arc, and let d be the length of the bad arc, we have*

$$X^{(\alpha)} \approx_\epsilon U\left[i+\frac{d}{2}+1, p-i-\frac{d}{2}-1\right],$$

where $U[i+\frac{d}{2}+1, p-i-\frac{d}{2}-1]$ denotes a uniform distribution with support $[i+\frac{d}{2}+1, i+\frac{d}{2}+2, \ldots, p-i-\frac{d}{2}-1]$, i.e. the uniform distribution along the entire cycle conditioned on not intersecting with the good arc.

Proof. In the absence of NEVER constraints, this lemma follows from the observation that the only direct label-to-discrete-log mapping in the constraints comes from the $\mathcal{INIT}$ constraint, which maps a label to the group identity g^0 with discrete log of 0. All the other LABEL constraints only give out relative discrete logs. For example, a LABEL constraint (α_1, α_2) only tells us the relation $x^{(\alpha_2)} = x^{(\alpha_1)} + 1$, but without revealing what $x^{(\alpha_1)}$ and $x^{(\alpha_2)}$ really are. Therefore, in order to know concretely the value of $x^{(\alpha_1)}$, it must be chained, through a sequence of LABEL constraints, back

to the $\mathcal{INIT}$ constraint. And this directly implies that this constraint (α_1, α_2) is in the good arc. As a result, for any bad arc, i.e. any arc that does not contain node v_0, its corresponding group elements have unknown discrete logs and are equally likely to be any group element without an assigned label (since these group elements have the same label distribution). Therefore, its location on the cycle follows a uniform distribution, conditioned on that it does not intersect with the good arc, i.e. $X^{(\alpha)} \approx_\epsilon U[i + \frac{d}{2} + 1, p - i - \frac{d}{2} - 1]$. Notice that in order to not intersect with the good arc at round i, the center of the bad arc must be at least distance $\frac{d}{2} + 1$ away from the endpoints i and $-i$ of the good arc.

However, what happens if NEVER constraints exist? Recall that α is the label at the center of the bad arc, and let $x^{(\alpha)}$ be the discrete log of its corresponding group element, and never be an added NEVER constraint. We now apply Lemma 9 with $a = x^{(\alpha)} - \frac{d}{2}, b = x^{(\alpha)} + \frac{d}{2}$ and $\mathcal{C}$ be the collection of prior constraints. With overwhelming probability, we have that

$$I\left(\mathsf{never}; \{L^{(i)}\}_{i \in \left[x^{(\alpha)} - \frac{d}{2}, x^{(\alpha)} + \frac{d}{2}\right]} \middle| \mathcal{C}\right) = negl(\lambda).$$

This essentially says that with overwhelming probability, the mutual information between the never constraint and any specific arc centered at α is negligible. Therefore, the adversary cannot "concentrate" the mutual information between the never constraint and group element distributions $X^{(\alpha)}$ for $\alpha \in \{0,1\}^n$ towards any particular location on the arc, meaning the never constraint can only affect the group element distribution $X^{(\alpha)}$ negligibly. We know that without the never constraint, $X^{(\alpha)}$ follows the uniform distribution $U[i + \frac{d}{2} + 1, p - i - \frac{d}{2} - 1]$, and that with overwhelming probability, the never constraint's impact on $X^{(\alpha)}$ is negligible. Therefore, with overwhelming probability, we have $X^{(\alpha)}$ is statistically close to $U[i + \frac{d}{2} + 1, p - i - \frac{d}{2} - 1]$.

To extend the above argument to multiple NEVER constraints, we apply an inductive argument. Given the constraint graph and a list of NEVER constraints, the impact of a new NEVER constraint on the group element distributions is also negligible by the same reasoning above. Notice that this can be repeated up to a polynomial times (and hence allowing up to polynomial number of NEVER constraints). With overwhelming probability, the total amount of mutual information these NEVER constraints can give on the group element distributions is only $poly(\lambda) \cdot \log \lambda$ bits, but the information is distributed evenly among 2^λ number of group element distributions, meaning the the mutual information between the NEVER constraint and any specific group element distribution $X^{(\alpha)}$ is only $2^{-\lambda} poly(\lambda) \log \lambda = negl(\lambda)$. Hence, even with a polynomial number of NEVER constraints, their impact on each group element distribution is negligible, so we can fall back to our argument at the beginning of this proof where we simply ignore the NEVER constraints. □

With Lemma 11, we can think about the constraint graph for any collection of $\mathcal{INIT}, \mathcal{LABEL}$, $\mathcal{NEVER}, \mathcal{AUTO}^*$ and $\mathcal{AUTO}$ constraints as a good arc, together with some other arcs randomly located along the entire cycle.

Now consider the collection of constraints and their corresponding constraint graph at the end of round i, we will argue that it is (almost) impossible to have an arc of length at least $2i + 1$. We say it is (almost) impossible because of possible arc intersections: imagine we have two separate arcs that are less than $2d$ distance apart, then after d rounds, if we extend both arcs in each round, they would now intersect and form a single arc that could longer than the bound we're trying to prove. As a special case, this also captures where the adversary just luckily guesses the label for a group element, as it would correspond to the adversary obtaining a new arc of length 1 that happens to intersect with some existing arc. In the following lemma, we prove the bound nevertheless assuming

such intersections does not happen, and in the next lemma, we bound the probability of such arc intersections to be negligible.

Lemma 12. *Let $\mathcal{INIT}, \mathcal{LABEL}, \mathcal{NEVER}, \mathcal{AUTO}^*$ and $\mathcal{AUTO}$ be the collection of constraints at the end of round i, and G_i be the corresponding constraint graph, and $(v_{\ell(-a)}, \ldots, v_{\ell(-1)}, v_0, v_{\ell(1)}, \ldots, v_{\ell(b)})$ be some arc in G_i (W.L.O.G. we assume the good arc). Then assuming that the arc does not intersect with another arc, with all but negligible probability in the security parameter λ, we have $a, b \leq i-1$.*

Proof. Again, let's first argue the simpler case without NEVER constraints. Assume W.L.O.G. towards contradiction that we have $b >= i$ and $a = i-1$. Consider the nodes $v_0, ..., v_{\ell(b)}$. In order to produce the labels for these nodes, $\ell^{(0)}, \ell^{(1)}, \ldots, \ell^{(i-1)}$, one must evaluate $\mathcal{O}_{\mathsf{Mult}}$ for each of $\ell^{(0)}, \ell^{(1)}, \ldots, \ell^{(i-2)}$, either directly via an $\mathcal{O}_{\mathsf{Mult}}$ query, or somewhere in the circuit for an $\mathcal{O}_{\mathsf{Inv}}$ query. Notice that for $\mathcal{O}_{\mathsf{Inv}}$ queries, it can either extend existing arcs or generate new arcs, but in both cases, as we will show in a moment, the length that is extended or generated is still bounded.

First of all, by the sequentiality nature of the $\mathcal{O}_{\mathsf{Mult}}$ and the fact that they take 1 round to finish, by the end of round i, using $\mathcal{O}_{\mathsf{Mult}}$ queries alone, we can only reach as far as $\ell^{(i-1)}$, but not $\ell^{(i)}$. What about the $\mathcal{O}_{\mathsf{Inv}}$ queries? We prove this via strong induction. As inductive hypothesis, we assume that we only have labels up to $\ell^{(i'-1)}$ in round i' for all $i' < i$. Now we show that in round i, we also only have labels up to $\ell^{(i-1)}$. Let's consider an $\mathcal{O}_{\mathsf{Inv}}$ query submitted back in round $j < i$. In order for it to finish before the end of round i, its size (the number of gates, counting recursively) must be bounded by $i-j$. This indicates that this $\mathcal{O}_{\mathsf{Inv}}$ query can produce at most $i-j$ LABEL constraints, since we only add one LABEL constraint for each $\mathcal{O}_{\mathsf{Mult}}$ gate in the circuit. But this query was submitted back in round j, so by the inductive hypothesis, it can only depend on the labels up to $\ell^{(j-1)}$. Extending these labels by length $i-j$ can only get us to as far as $\ell^{(i-1)}$, but not $\ell^{(i)}$. Notice that this argument readily extends to circuits with recursive calls to the $\mathcal{O}_{\mathsf{Inv}}$ oracle, following the recursive definition of the "size" of the circuit.

Now let's consider what happens with NEVER constraints included. Our argument is of a very similar style as the one without the NEVER constraints. We prove the full statement using strong induction on the round i. Let P_i be the proposition that at the end of round i, with overwhelming probability, the arc only reaches $v_{\ell(-a)}$ and $v_{\ell(b)}$ where $a, b \leq i-1$. The base case P_1 is trivial since at the end of the first round, we haven't heard back from any queries yet, and the good arc only contains v_0 corresponding to the $\mathcal{INIT}$ constraint.

Next, we show the inductive step that if $P_1, P_2, \ldots, P_{i-1}$ are all true, then P_i holds true as well. To see this, we consider the query responses we receive at round i. Notice that a query response will produce either LABEL constraints or a NEVER constraint that gets added to the constraint list.

Notice that for any constraints created for a circuit of size d, this means the query takes time d to complete, so to receive the response at round i, the query must have been submitted at round $i-d$. For instance, if a NEVER constraint is added for a circuit of size d, this also means the query was submitted in round $i-d$. Notice that by P_{i-d}, at the time the query was submitted, it can only depend on $(v_{\ell(-i+d+1)}, \ldots, v_0, \ldots v_{\ell(i-d-1)})$ from the good arc (and potentially other uniformly distributed arcs). Specifically, it cannot depend on the nodes $v_{\ell(-i)}, \ldots, v_{\ell(-i+d)}$ and $v_{\ell(i-d)}, \ldots, v_{\ell(i)}$, since these nodes have not been determined yet at the time the query was submitted. Applying Lemma 9 with $a = i-d, b = i$ and $\mathcal{C}$ be the collection of prior constraints. With overwhelming probability, we have that

$$I\left(\mathsf{never}; \{L^{(i)}\}_{i \in [i-d,i]} \middle| \mathcal{C}\right) = negl(\lambda),$$

meaning the never constraint only gives negligible information towards predicting these labels, and similarly for $\{L^{(i)}\}_{i\in[-i,-i+d]}$ as well. Then we use a union bound similar as in Lemma 10 and Lemma 11, and have that even for a $poly(\lambda)$ number of NEVER constraints, they give negligible mutual information for the label distributions $\{L^{(i)}\}_{i\in[-i,-i+d]}$ and $\{L^{(i)}\}_{i\in[i-d,i]}$.

By proposition P_{i-d}, the good arc at round $i-d$ reaches at most $v_{\ell(-i+d+1)}$ and $v_{\ell(i-d-1)}$. Then, with our above argument handling just the LABEL constraints and ignoring the NEVER constraints, with overwhelming probability, the two sides of the arc can grow by at most d, and therefore reaching at most $v_{\ell(-i+1)}$ and $v_{\ell(i-1)}$ as desired.

Therefore, with both LABEL and NEVER constraints, any arc in G_i at round i cannot reach nodes beyond $v_{\ell(-i+1)}$ and $v_{\ell(i-1)}$ with non-negligible probability. The rest follows by induction. □

Next, we argue that the probability of having such arc intersections is also negligible.

Lemma 13. *Let G_i be the constraint graph at round i. The probability that there exists two arcs in G_i that in a later round $j \le T$ "merge" together into a single arc is negligible.*

Proof. First, we notice that there are only $poly(\lambda)$ number arcs in G_i and that all the arcs are $poly(\lambda)$ in length. This is because we can only generate $poly(\lambda)$ number of constraints each round, and there are a total of $T = poly(\lambda)$ rounds. Each arc corresponds to at least one constraint, and the arc length corresponds to the number of chained LABEL constraints, so both the number of arcs and the arc lengths are bounded by $poly(\lambda)$.

Now let us calculate the probability of two specific arcs having an intersection. Recall that an arc can only extend its either side by at most $d = j - i \le T = poly(\lambda)$. Therefore, for two arcs to have an intersection, their endpoints must be within $2d$ distance of each other. Fixing one arc, then the other arc only has $4d$ possible locations in order to intersect. (Note that in the case of one arc being the good arc centered around v_0, we always fix the good arc and consider the bad arc, which is guaranteed to be randomly located by Lemma 11). Hence, the probability of any two given arcs having an intersection is $4d/p = O(poly(\lambda)2^{-\lambda})$.

Then, by union bounding over all possible pairs of arcs ($(poly(\lambda))^2$ number of these), the probability of having any intersection is bounded by $O((poly(\lambda))^2 \cdot poly(\lambda)2^{-\lambda})$, which is negligible as desired. □

With the above three lemmas, we can now finish the proof for Lemma 7.

Lemma 14. *With all but negligible probability,*

$$\{L^{(i)}\}_{i\in[-T,T]}|\mathcal{AUTO}^* \approx_\epsilon \{L^{(i)}\}_{i\in[-T,T]}|\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO},$$

where $\approx_\epsilon$ denotes the two distribution are statistically close.

Proof. The difference between these two distributions is that the latter is additionally conditioned on the set of constraints $\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}$. What will cause these two distributions to differ? If without these constraints, the sampled label will present some form of *collision* with one or more of these constraints. Let's say we are at the beginning of round k, and for some $i \in [-T, T]$, we sample $\ell^{(i)} \sim L^{(i)}|\mathcal{AUTO}^*$. There are two types of possible collisions:

1. **Group element collision:** the label $\ell^{(i)}$ corresponds to the same group element that a different label ℓ' in $\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$ corresponds to. Notice that $\ell^{(i)}$ corresponds to the group element g^i. If we have $-k+1 \le i \le k-1$, notice that $\ell^{(i)}$

will be fully determined by the $\mathcal{AUTO}^*$ constraints in both distributions, as the labels $\ell^{(-k+1)}, \ell^{(-k+2)}, \ldots, \ell^{(k-1)}$ are already sampled and put in the $\mathcal{AUTO}^*$ constraints. In this case, for the group element g^i, both distributions should give the same label, and hence we don't have a group element collision. On the other hand, if $i \geq k$ or $i \leq -k$, consider the constraint graph constructed formed by the constraints $\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. If in the constraints there is a label ℓ' for the group element g^i, then there must be some node that corresponds to g^i in the constraint graph. By Lemma 11, with overwhelming probability, that node must be in the good arc (otherwise, the bad arc will be uniformly distributed along the entire cycle of length $\Theta(2^\lambda)$, and the probability of it containing the node for g^i is negligible). This means there exists an arc $(v_0, v_{\ell^{(1)}}, \ldots, v_{\ell^{(i)}})$ in the constraint graph. However, by combining Lemma 12 and Lemma 13, with overwhelming probability that no arc intersection happens, one can only extend the good arc to at most $v_{\ell^{(k-1)}}$, but not $v_{\ell^{(i)}}$, where $i > k-1$. Therefore, the probability of having a group element collision is negligible.

2. **Label collision:** the label $\ell^{(i)}$ is the same as some label ℓ' in $\mathcal{INIT} \wedge \mathcal{LABEL} \wedge \mathcal{AUTO}^* \wedge \mathcal{AUTO}$. Notice that we can submit at most q queries per round, and by round k we can submit at most qk number of queries. For the queries responded before round k, their circuit size must be bounded by k, meaning that each such query can produce at most k constraints. Also, notice that each constraint can introduce at most 2 labels. Therefore, there are at most $2q \cdot k^2$ number of labels in the constraints, where $q, T = poly(\lambda)$, but there are a total of 2^λ possible labels. By union bound, the probability of having such a label collision is bounded by $2qT^2 2^{-\lambda} = poly(\lambda) 2^{-\lambda}$, which is negligible.

Bringing these two parts together, the probability of having either a group element or a label collision is negligible. Therefore, with overwhelming probability, the two distributions only differ negligibly, making them statistically close as desired. □

5.7 Intuition on the Oracles in the Proof

At a first glance, it might seem that the oracles used in our proof are extremely non-standard: for instance, we allow our inversion oracles to take as input circuits that call the oracles themselves, we bound the number of queries an oracle can make in a certain amount of time, and we impose delays on response times to certain queries. For readers that may only be familiar with traditional lower bounds like [IR89] or [BM09], this may cause some initial discomfort. However, these techniques are not novel and we could circumvent the latter two with some extra formalism if we desired.

Letting an Oracle Call Circuit Contain Its Own Routines. It may seem unusual to let a generic primitive call its own subroutines (in other words, let an oracle call itself). However, this technique is not without precedent: for instance, in [GMM17], the authors create an *extended black-box framework* that does exactly this. Their goals–to model primitives like functional encryption and fully homomorphic encryption, which have numerous non-black box constructions and proofs–are very different from ours, but they show that this technique is useful and not inherently incompatible with separation results. While we do not have to use all of the machinery that they do in their extended black-box framework, we refer interested readers to their work for more details.

Bounding the Number of Times an Oracle Can Be Called in a Time Period. In traditional separation results, oracles can be called in an arbitrary manner at arbitrary times, and

(typically) only the total amount of oracle queries is considered in the lower bound. However, in order to abstractly model things like proof of work systems in the blockchain [GKL15], it is necessary to consider oracles that can only be called a certain amount of times within a certain time period; this concept is in fact integral to proof-of-work consensus algorithms being secure. This sort of technique has been used far too many times for us to keep track in the literature in some form; some notable examples include [GKW+16, BGK+18, BMTZ17].

We note that it would be simple to bound the number of times an oracle can be called in a particular time period without requiring this of the oracle itself by defining a challenger (or more complicated security game, like in the UC model [Can01]) that regulates how many times an oracle could be called in a particular round (or time period). This way, no such restrictions would have to be placed on the oracle itself. We could add this to our main separation result, and it would not change any of our results.

Oracles with Delayed Response Times. We note that [BBBF18] introduced *random delay oracles*, which they define to be essentially just random oracles with delayed response times, and use them to help prove properties of VDFs. This technique has also been used in other places as well [Fis18]. We note that this sort of oracle could also be avoided with extra formalism: a challenger (or environment) could enforce the delays rather than the oracle itself.

6 Sequential Functions vs. PSPACE

In this section, we explore the relationship between sequential functions and PSPACE. Specifically, we show that Continuous Iterative Sequential Functions (CISFs) are PSPACE-complete, while the more general Dynamic Sequential Functions (DSFs) are *not* in PSPACE in the random oracle model.

6.1 (Worst-Case) CISFs are PSPACE-Complete

First, recall the syntax of a CISF [JMRR21] from Definition 4:

Definition 13 (Continuous Iterative Sequential Functions)**.** *A* ***continuous iterative sequential function (CISF)*** $F = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Round})$ *is defined as the following tuple of algorithms:*

$\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$**:** *On input the security parameter* 1^λ*, the setup algorithm returns the public parameters* pp*. By convention, the public parameters encode an input domain* X *and an output domain* Y*.*

$\mathsf{Gen}(\mathsf{pp}) \to x$**:** *On input the public parameters* pp*, the instance generation algorithm samples a random input* $x \leftarrow X$*.*

$\mathsf{Eval}(\mathsf{pp}, x, k) = (\mathsf{Round}(\mathsf{pp}, \cdot))^{(k)}(x) \to y$**:** *On input the public parameters* pp*, an input* $x \in X$*, and* $k \in 2^{o(\lambda)}$*, the evaluation algorithm runs the* Round *function iteratively, and eventually returns an output* $y \in Y$*.*

An CISF F *satisfies* $(t_C(\lambda), t_A(\lambda))$*-sequentiality as defined in Definition 3.*

Notice that CISFs are by nature a computational problem. To show that CISFs are PSPACE-complete, we would need a corresponding decisional problem. Concretely, we consider the following decisional variant of the CISF problem, which we will refer to as the Decisional CISF (DCISF) problem. This is just the (almost) immediate extension of a computational problem to a decision problem.

Furthermore, note that CISFs are defined in a cryptographic way: the definition of security is clearly an average-case notion. However, PSPACE is defined with respect to the worst case. So, in addition to defining a decisional version of a CISF, we also need to make sure our definition is "worst-case".

Definition 14 (Decisional CISF Problem)**.** *A Decisional CISF (DCISF) problem consists of an instantiation of a CISF* $F = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Round})$ *with public parameter* pp *that is a valid output of the* Setup *procedure, a valid CISF input* $x \in X$*, a value* $y \in Y$ *in the CISF output space, and* $k \in 2^{o(\lambda)}$*. If* $\mathsf{Eval}(\mathsf{pp}, x, k) = y$*, then the answer to the problem is true. Otherwise, the answer to the problem is false.*

Note that our definition here just requires that the public parameters pp and input x are valid outputs of their respective sampling algorithms, and not that they have actually been randomly sampled.

Lemma 15. *The DCISF problem is in* PSPACE*.*

Proof. To show that the DCISF problem is in PSPACE, we argue that all CISFs can be computed using only polynomial space. If this is true, then we can just compute the CISF in PSPACE and then immediately check the output.

It turns out that this is almost immediate due to the definition of a CISF. Notice that a CISF runs Eval by iteratively executing the Round function, which takes as input $x \in X$ of $poly(\lambda)$ size. Furthermore, each iteration of the Round function is only of $poly(\lambda)$ time. Thus, by assuming a constant rate of writing to memory, each Round function can only use up to $poly(\lambda)$ amount of space. And hence, the overall Eval can also be executed under polynomial space. Using this, we can trivially solve the DCISF problem - simply run the CISF faithfully using polynomial space and then compare the output y' with y. Therefore, the DCISF problem is in PSPACE. □

The more complicated direction involves proving that an algorithm for the DCISF problem can be used to solve any problem in PSPACE. To do this, we will use a reduction from the True Quantified Boolean Formula (TQBF) problem, which is known to be PSPACE-complete [Pap07].

Lemma 16. *The DCISF problem is* PSPACE*-hard.*

Proof. To show that the DCISF problem is PSPACE-hard, we reduce the True Quantified Boolean Formula (TQBF) problem to the DCISF problem using a Cook reduction. Specifically, we show that an oracle that can solve an arbitrary DCISF problem can be used to solve the TQBF problem. Since the TQBF problem is a known PSPACE-complete problem, by the reduction, we would have that DCISF is PSPACE-hard as desired.

To do this, we implement a naïve iterative solving algorithm for the TQBF problem that can be modeled as a CISF.

First, recall a TQBF problem statement

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \phi(x_1, \dots, x_n)$$

where $Q_1, \ldots Q_n \in \{\exists, \forall\}$ are quantifiers, $x_1, \ldots, x_n$ are boolean variables, and ϕ is a boolean formula. The answer to the problem is true if and only if there exists an assignment of $x_1, \ldots, x_n \in \{0, 1\}$ such that the overall boolean formula $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi(x_1, \ldots, x_n)$ is true. Now, consider the following algorithm defined in Algorithm 1.

Algorithm 1 A Naïve Iterative Solver for the TQBF problem

```
i ← 0
s ← Stack.Init()                          ▷ A stack of elements of format (v, d)
while i < 2^n do
    x_1, x_2, ..., x_n ← ToBinary(i)   ▷ Convert i to binary, and assign x_1, ..., x_n to be the binary
representation of i
    v ← φ(x_1, x_2, ..., x_n)
    d ← n                                 ▷ The depth of the tree that we're currently at
    while !s.IsEmpty() and s.Peek().d = d do
        if Q_d is ∀ then
            v ← s.Pop().v ∧ v
        else
            v ← s.Pop().v ∨ v
        end if
        d ← d − 1
    end while
    s.Push((v, d))
    i ← i + 1
end while
Output s.Pop().v
```

First, recall that a stack is a first-in last-out data structure. The elements "push"ed into a stack first are "pop"ed out at last, and "peek" allows one to take a look at the next element to be "pop"ed out without actually "pop"ing it.

How does this algorithm work? Let's start by explaining a simplified version. Suppose we consider a binary tree of depth n, where each leaf node is associated with a particular assignment of variables. Each level of the tree, except for the base layer containing the leaf nodes, is associated with either an AND or an OR gate–we can think of each node in the layer as having the same gate. If layer 0 is the leaf layer and layer $n + 1$ is the root node, then layer i is associated with an AND gate if $Q_i = \forall$ and an OR gate if $Q_i = \exists$. We also associate each layer i with bit x_i.

Evaluating the TQBF then reduces to the following: evaluating the validity of ϕ on all of the leaf nodes, and then computing the circuit implied by the tree. At each layer, we eliminate one bit and one constraint. This is essentially equivalent to the standard folklore algorithm for checking the truthfulness of a QBF.

However, there is one major issue: if we are not careful, our computation of the tree may take exponential space. So we must use a depth first search evaluation of the tree, which only requires linear (in n) space. So, roughly speaking, the above algorithm simulates a depth first search using a stack data structure. It starts at the bottom layer by assigning values to all variables, and then gradually works its way up by combining the value of the boolean formula under the two possible values of the same boolean variable using the corresponding quantifier. Specifically, for example, let $x_1, x_2, \ldots, x_{i-1}$ be

fixed, and for $x_i = 0$, let $a = Q_{i+1}x_{i+1}, \ldots, Q_n x_n \phi(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$, and for $x_i = 1$, let $b = Q_{i+1}x_{i+1}, \ldots, Q_n x_n \phi(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$. If $Q_i = \forall$, let $c = a \wedge b$, otherwise, let $c = a \vee b$. Notice that we successfully get $c = Q_i x_i, \ldots, Q_n x_n \phi(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$. By repeating this step, we can add in all the quantifiers, and eventually get $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi(x_1, \ldots, x_n)$ as the output.

Notice that we can define the Round function for the CISF to be each iteration of the outer while loop as in Algorithm 1, and the input x to consist of the TQBF description, the iterator i, and the stack s. Notice that s has at most n elements and hence the whole input to Round is still of $poly(\lambda)$ size. Therefore, to solve the TQBF problem, we can construct a DCISF problem instance with the above Round function and input, and $y = 1$ as the output. The answer to the TQBF problem is simply the answer to the crafted DCISF problem. □

Bringing lemma 15 and lemma 16 together, we have the following theorem.

Theorem 2. *The DCISF problem is* PSPACE*-complete.*

6.2 Dynamic Sequential Functions are not in PSPACE

Then, we show that while CISFs are PSPACE-complete, slightly more general sequential functions known as dynamic sequential functions (DSFs) are *not* in PSPACE. Specifically, we will come up with a Decisional Dynamic Sequential Function problem that is not solvable in polynomial space. But first, let us recall the syntax of a Dynamic Sequential Function from Definition 3, and define its corresponding decisonal problem similar to how we define a DCISF.

Definition 15. *A **dynamic sequential function (DSF)** $F = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval})$ is defined as the following tuple of algorithms:*

$\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$**:** *On input the security parameter 1^λ , the setup algorithm returns the public parameters pp. By convention, the public parameters encode an input domain X and an output domain Y.*

$\mathsf{Gen}(\mathsf{pp}) \to x$**:** *On input the public parameters pp , the instance generation algorithm samples a random input $x \leftarrow X$.*

$\mathsf{Eval}(\mathsf{pp}, x, k) \to y$**:** *On input the public parameters pp, an input $x \in X$, and $k \in 2^{o(\lambda)}$, the evaluation algorithm returns an output $y \in Y$.*

F needs to satisfy the $(t_C(\lambda), t_A(\lambda))$-sequentiality as defined in Definition 3.

Definition 16 (Decisional Dynamic Sequential Function Problem)**.** *A Decisional Dynamic Sequential Function (DDSF) problem consists of an instantiation of a Dynamic Sequential Function $F = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval})$ with public parameter pp that is a valid output of the Setup procedure, a valid DSF input $x \in X$, a DSF output $y \in Y$, and $k \in 2^{o(\lambda)}$. If $\mathsf{Eval}(\mathsf{pp}, x, k) = y$, then the answer to the problem is true. Otherwise, the answer to the problem is false.*

To construct a DDSF problem that requires super-polynomial space to solve, we utilize a graph with high space complexity. These graphs are often used to argue the space complexity of Memory-Hard Functions (MHFs) through a *pebbling game* [AS15].

Definition 17 (Pebbling a Graph (in Parallel) [AS15])**.** *Given $G = (V, E)$ a DAG with source nodes $S \subseteq V$ and sink nodes $T \subseteq V$, a (legal) complete pebbling of G is a sequence $P = (P_0, \ldots, P_t)$ of subsets of V such that:*

1. *$P_0 \subseteq S$.*
2. *A pebble can be put on a node only if the node's predecessor has a pebble in the previous step.*

$$\forall i \in [t], \forall (x, y) \in E, \forall y \in P_i \backslash P_{i-1} \quad x \in P_{i-1}.$$

3. *At some point every target node is pebbled (not necessarily simultaneously).*

$$\forall y \in T, \exists i \in [t] \quad y \in P_i.$$

Notice that a pebble can be placed on a source node or removed from any node at any time.

The above definition is for a *parallel* pebbling game. One can also consider a *sequential* pebbling game where one can place only one pebble each step[1], i.e. $\forall i \in [t], |P_i \backslash P_{i-1}| \leq 1$. In fact, the specific graph that we consider, the graph by Paul, Tarjan, and Celoni [PTC76], originally considers such a sequential pebbling.

Lemma 17 ([PTC76])**.** *There exists a family of DAGs $\{G_n\}_{n \in \mathbb{N}}$ with in degree of 2, 2^n sources and sinks, and $O(n2^n)$ vertices, and have the property that for all $n \geq 8$, for any legal sequential pebbling of the graph, there exists an interval such that during that interval, there are at least $\Omega(2^n)$ pebbles always on the graph.*

Notice that the above lemma only deals with sequential pebbling, hence only models a machine with sequential computation power. To model a parallel machine, we would need the following strengthening of the Lemma by Alwen, Blocki, and Pietrzak [ABP18], which extends the above lemma to any *parallel* pebbling.

Lemma 18 ([ABP18])**.** *There exists a family of DAGs $\{G_n\}_{n \in \mathbb{N}}$ with in degree of 2, 2^n sources and sinks, and $O(n2^n)$ vertices, and have the property that for all $n \geq 8$, for any legal **parallel** pebbling of the graph, there exists an interval such that during that interval, there are at least $\Omega(2^n)$ pebbles always on the graph.*

Eventually, we will construct a DSF based on this family of graphs, so the function description naturally contains a description of these graphs. We want these graphs to be *compactly representable*.

Lemma 19. *The family of DAGs $\{G_n\}_{n \in \mathbb{N}}$ from [PTC76] can be compactly represented using a constant-size description of the graph and an additional parameter n to tune the graph size.*

Proof. To prove this, we reproduce the construction from [PTC76], and verify that it can be compactly represented. First of all, to construct this family of DAGs G_n, we would need the superconcentrators B_n from [Val75].

Definition 18 (Superconcentrator)**.** *A k-superconcentrator is a DAG with k source nodes and k sink nodes, such that for all $j \in [k]$, if S is any subset of j source nodes, and T is any subset of j sink nodes, there exist j vertex-disjoint paths from S to T.*

[1] One can still place a pebble a source node at any time, but that would take one step, since one cannot place two pebbles simultaneously.

[Val75] shows how to construct such 2^i-superconcentrators with $O(2^i)$ number of edges, which we will reproduce momentarily. Before that, we first see how to construct the desired G_n's assuming the existence of B_n, a 2^n-superconcentrator.

G_n from Superconcentrators. The graphs G_n's are recursively defined. Specifically, we define $G_8 = B_8$. And given G_i for $i = 8, 9, 10, \ldots$, we construct $G_{i+1} = (V_{i+1}, E_{i+1})$ from two copies of G_i and two copies of B_i as follows. Let the two copies of G_i be $G_i^{(1)} = (V_i^{(1)}, E_i^{(1)})$ and $G_i^{(2)} = (V_i^{(2)}, E_i^{(2)})$, and let the two copies of B_i be $B_i^{(1)} = (VB_i^{(1)}, EB_i^{(1)})$ and $B_i^{(2)} = (VB_i^{(2)}, EB_i^{(2)})$. Moreover, let $G_i^{(1)}$ and $G_i^{(2)}$ have source nodes $S_i^{(1)} = \{s_{i,j}^{(1)} : j \in [2^i]\}$ and $S_i^{(2)} = \{s_{i,j}^{(2)} : j \in [2^i]\}$ respectively, and $B_i^{(1)}$ and $B_i^{(2)}$ have source nodes $SB_i^{(1)} = \{sb_{i,j}^{(1)} : j \in [2^i]\}$ and $SB_i^{(2)} = \{sb_{i,j}^{(2)} : j \in [2^i]\}$. Similarly, we define the sink nodes for $G_i^{(1)}$, $G_i^{(2)}$, $B_i^{(1)}$, and $B_i^{(2)}$ as $T_i^{(1)} = \{t_{i,j}^{(1)} : j \in [2^i]\}$, $T_i^{(2)} = \{t_{i,j}^{(2)} : j \in [2^i]\}$, $TB_i^{(1)} = \{tb_{i,j}^{(1)} : j \in [2^i]\}$, $TB_i^{(2)} = \{tb_{i,j}^{(2)} : j \in [2^i]\}$. Additionally, we let $S_{i+1} = \{s_{i+1,j} : j \in [2^{i+1}]\}$ and $T_{i+1} = \{t_{i+1,j} : j \in [2^{i+1}]\}$ be the 2^{i+1} source nodes and sink nodes of G_{i+1}. We have $G_{i+1} = (V_{i+1}, E_{i+1})$ where

$$V_{i+1} = S_{i+1} \cup T_{i+1} \cup V_i^{(1)} \cup V_i^{(2)} \cup VB_i^{(1)} \cup VB_i^{(2)},$$

and

$$\begin{aligned} E_{i+1} =& E_i^{(1)} \cup E_i^{(2)} \cup EB_i^{(1)} \cup EB_i^{(2)} \\ & \cup \{(s_{i+1,j}, t_{i+1,j}) : j \in [2^{i+1}]\} \\ & \cup \{(s_{i+1,j}, sb_{i,j}^{(1)}) : j \in [2^i]\} \\ & \cup \{(s_{i+1,j+2^i}, sb_{i,j}^{(1)}) : j \in [2^i]\} \\ & \cup \{(tb_{i,j}^{(1)}, s_{i,j}^{(1)}) : j \in [2^i]\} \\ & \cup \{(t_{i,j}^{(1)}, s_{i,j}^{(2)}) : j \in [2^i]\} \\ & \cup \{(t_{i,j}^{(2)}, sb_{i,j}^{(2)}) : j \in [2^i]\} \\ & \cup \{(tb_{i,j}^{(2)}, t_{i+1,j}) : j \in [2^i]\} \\ & \cup \{(tb_{i,j}^{(2)}, t_{i+1,j+2^i}) : j \in [2^i]\}. \end{aligned}$$

Essentially, G_{i+1} is obtained by putting 2^{i+1} source nodes, $B_i^{(1)}$, $G_i^{(1)}$, $G_i^{(2)}$, $B_i^{(2)}$ and 2^{i+1} sink nodes in a sequence and adding the following edges:

- One-to-one edges from all the source nodes to all the sink nodes (each source node is connected to one sink node).
- One-to-one edges from the first half of source nodes to the source nodes of $B_i^{(1)}$.
- One-to-one edges from the second half of source nodes to the source nodes of $B_i^{(1)}$.
- One-to-one edges from the sink nodes of $B_i^{(1)}$ to the source nodes of $G_i^{(1)}$.
- One-to-one edges from the sink nodes of $G_i^{(1)}$ to the source nodes of $G_i^{(2)}$.

- One-to-one edges from the sink nodes of $G_i^{(2)}$ to the source nodes of $B_i^{(2)}$.
- One-to-one edges from the sink nodes of $B_i^{(2)}$ to the first half of sink nodes.
- One-to-one edges from the sink nodes of $B_i^{(2)}$ to the second half of sink nodes.

For a graphic illustration of the construction, see figure 3.

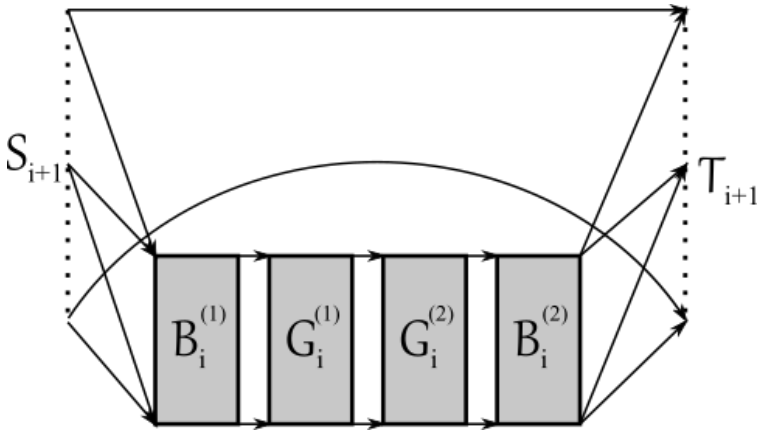


Figure 3: Construction of the graph G_{i+1} from two copies of G_i denoted as $G_i^{(1)}, G_i^{(2)}$ and two copies of 2^i-superconcentrators $B_i^{(1)}, B_i^{(2)}$. The dots represent source and sink nodes of G_{i+1}, and the arrows represent collections of edges between nodes.

Notice that given G_i and B_i, this recursive procedure can be easily executed by a constant-size Turing Machine, and hence the class of graphs G_n has a compact description, conditioned on that B_n's also have compact descriptions. Also, if B_n's have in degree of 2, G_n's have in degree of 2 as well.

Now we verify that such B_n's are indeed compact, by opening up the construction by [Val75].

It turns out that there are two extra levels of recursion: to construct B_n, a 2^n-superconcentrator, we would first construct a 2^n-hyperconcentrator, which we construct from $(2^n, 2^{n-1})$-concentrators. Hyperconcentrators and concentrators are defined as below.

Definition 19 (Hyperconcentrator)**.** *An k-hyperconcentrator is a DAG with k source nodes and k sink nodes, such that for all $j \in [k]$, if S is any subset of j source nodes, and T is the subset of the first j sink nodes, there exist j vertex-disjoint paths from S to T.*

Definition 20. *A (k, ℓ)-concentrator with $k \leq \ell$ is a DAG with k source nodes and ℓ sink nodes such that for any subset T of k sink nodes, there exsit k vertex-disjoint paths from the source nodes S to T.*

[Val75] cites [Pin73] for the existence of such concentrators, but the specific constructions of such concentrators appear to be unclear. Later on in the section, we will present our own construction of the needed concentrators. For now, we first see how to construct superconcentrators from hyperconcentrators, and hyperconcentrators from concentrators.

Superconcentrators from Hyperconcentrators. With these hyperconcentrators, it is easy to get a superconcentrator. To get a superconcentrator, we just need two copies of a hyperconcentrator H_i. We first reverse a copy by reversing all of its edges and get $\overline{H_i}$. Notice the sources of $\overline{H_i}$ used to be the sinks of H_i, and vice versa. Then we simply connect the sinks of H_i to the sources of $\overline{H_i}$ like a one-on-one mapping. The resulting graph is a superconcentrator B_i as needed. See figure 4 for an intuitive illustration. Notice that this construction is also compact and preserves the in degree of 2, assuming the hyperconcentrators have in degree *and out degree* of 2.

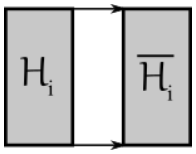


Figure 4: Construction of a 2^i-superconcentrator B_i from two copies of 2^i-hyperconcentrators $H_i, \overline{H_i}$. The source nodes are the source nodes of H_i, and the sink nodes are the sink nodes of $\overline{H_i}$ (originally sources before reversal).

Hyperconcentrators from Concentrators. Now we show assuming $(2^n, 2^{n-1})$-concentrators C_n, how one can construct a 2^n-hyperconcentrator H_n.

The construction is once again recursive. First, notice that H_1 is trivial - a graph with two nodes and a single edge connecting them suffices. Then for $i \geq 1$, let $H_i = (VH_i, EH_i)$ be a 2^i-hyperconcentrator, and $C^{(1)}_{i+1} = (VC^{(1)}_{i+1}, EC^{(1)}_{i+1})$ and $C^{(2)}_{i+1} = (VC^{(2)}_{i+1}, EC^{(2)}_{i+1})$ be two copies of $(2^{i+1}, 2^i)$-concentrators. We show how to construct H_{i+1}, a 2^{i+1}-hyperconcentrator. Let H_i have source nodes $SH_i = \{sh_{i,j} : j \in [2^i]\}$, and $C^{(1)}_{i+1}$ and $C^{(2)}_{i+1}$ have source nodes $SC^{(1)}_{i+1} = \{sc^{(1)}_{i+1,j} : j \in [2^{i+1}]\}$ and $SC^{(2)}_{i+1} = \{sc^{(2)}_{i+1,j} : j \in [2^{i+1}]\}$. Similarly, we define the sink nodes for H_i, $C^{(1)}_{i+1}$ and $C^{(2)}_{i+1}$ as $TH_i = \{th_{i,j} : j \in [2^i]\}$, $TC^{(1)}_{i+1} = \{tc^{(1)}_{i+1,j} : j \in [2^i]\}$, and $TC^{(2)}_{i+1} = \{tc^{(2)}_{i+1,j} : j \in [2^i]\}$. Additionally, we let $SH_{i+1} = \{sh_{i+1,j} : j \in [2^{i+1}]\}$ and $TH_{i+1} = \{th_{i+1,j} : j \in [2^{i+1}]\}$ be the 2^{i+1} source nodes and sink nodes of H_{i+1}. We have $H_{i+1} = (VH_{i+1}, EH_{i+1})$ where

$$VH_{i+1} = SH_{i+1} \cup TH_{i+1} \cup VH_i \cup VC^{(1)}_{i+1} \cup VC^{(2)}_{i+1},$$

and

$$\begin{aligned} E_{i+1} =& EH_i \cup EC^{(1)}_{i+1} \cup EC^{(2)}_{i+1} \\ & \cup \{(sh_{i+1,j}, sc^{(1)}_{i+1,j}) : j \in [2^{i+1}]\} \\ & \cup \{(sh_{i+1,j}, sc^{(2)}_{i+1,j}) : j \in [2^{i+1}]\} \\ & \cup \{(tc^{(1)}_{i+1,j}, th_{i+1,j}) : j \in [2^i]\} \\ & \cup \{(tc^{(1)}_{i+1,j}, sh_{i,j}) : j \in [2^i]\} \\ & \cup \{(tc^{(2)}_{i+1,j}, sh_{i,j}) : j \in [2^i]\} \\ & \cup \{(th_{i,j}, th_{i+1,j}) : j \in [2^i]\} \\ & \cup \{(th_{i,j}, th_{i+1,j+2^i}) : j \in [2^i]\}. \end{aligned}$$

Essentially, we can think about the construction as putting the 2^{i+1} source nodes, the two $(2^{i+1}, 2^i)$-concentrators, the 2^i-hyperconcentrator, and the 2^{i+1} sink nodes into four layers and by adding the following edges:

- One-to-one edges from all the source nodes (of H_{i+1}) to all the source nodes of $C^{(1)}_{i+1}$.
- One-to-one edges from all the source nodes (of H_{i+1}) to all the source nodes of $C^{(2)}_{i+1}$.
- One-to-one edges from all the sink nodes of $C^{(1)}_{i+1}$ to the first half of the sink nodes (of H_{i+1}).
- One-to-one edges from all the sink nodes of $C^{(1)}_{i+1}$ to all the source nodes of H_i.
- One-to-one edges from all the sink nodes of $C^{(2)}_{i+1}$ to all the source nodes of H_i.
- One-to-one edges from all the sink nodes of H_i to the first half of the sink nodes (of H_{i+1}).
- One-to-one edges from all the sink nodes of H_i to the second half of the sink nodes (of H_{i+1}).

See figure 5 for an illustration of the construction.

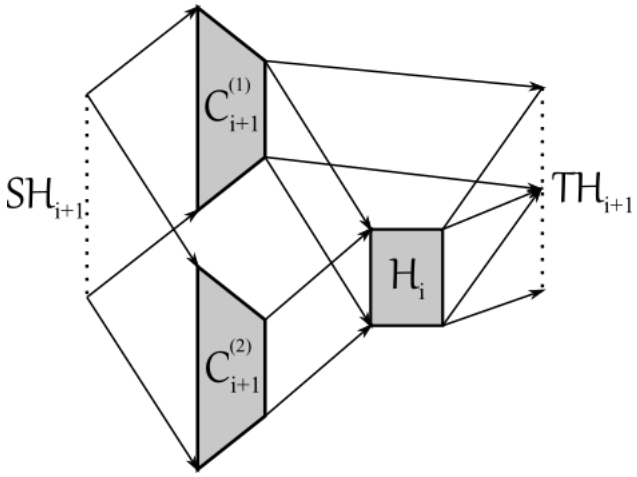


Figure 5: Construction of a 2^{i+1}-hyperconcentrator H_{i+1} from two copies of $(2^{i+1}, 2^i)$-concentrators $C^{(1)}_{i+1}, C^{(2)}_{i+1}$ and a 2^i-hyperconcentrator H_i.

Again, this recursive procedure can be described by a constant size TM, and hence H_n's can be compactly represented conditioned on that such C_n's are compactly representable. Also notice that if C_n has in degree and out degree of 2, then H_n has in degree and out degree of 2 as well.

Constructing Concentrators. Lastly, to complete the argument, we show that the construction of C_n's can be compactly represented. [Pin73] does not present any concrete construction of $(2^n, 2^{n-1})$-concentrators, so here we present our own construction of a $(2^n, 2^{n-1})$-concentrator.

The construction is yet again recursive and in fact pretty simple. First, observe the base case when $n = 1$. We can easily get a $(2, 1)$-concentrator by having two source nodes, one sink node, and an edge from each source to the sink. Now assume that we have $(2^i, 2^{i-1})$ concentrators, we show how to obtain a $(2^{i+1}, 2^i)$ concentrator. Let $C^{(1)}_i = (VC^{(1)}_i, EC^{(1)}_i)$ and $C^{(2)}_i = (VC^{(2)}_i, EC^{(2)}_i)$ be two

copies of $(2^i, 2^{i-1})$-concentrators. Let $C_i^{(1)}$ and $C_i^{(2)}$ have source nodes $SC_i^{(1)} = \{sc_{i,j}^{(1)} : j \in [2^i]\}$ and $SC_i^{(2)} = \{sc_{i,j}^{(2)} : j \in [2^i]\}$, and sink nodes $TC_i^{(1)} = \{tc_{i,j}^{(1)} : j \in [2^{i-1}]\}$ and $TC_i^{(2)} = \{tc_{i,j}^{(2)} : j \in [2^{i-1}]\}$. We construct $C_{i+1} = (VC_{i+1}, EC_{i+1})$ as follows. Let it have sources $SC_{i+1} = \{sc_{i,j} : j \in [2^{i+1}]\}$. Its sinks are simply the union of $TC_i^{(1)}$ and $TC_i^{(2)}$, i.e. $TC_{i+1} = TC_i^{(1)} \cup TC_i^{(2)}$. The vertex and edges are defined as follow:

$$VC_{i+1} = SC_{i+1} \cup VC_i^{(1)} \cup VC_i^{(2)},$$

and

$$\begin{aligned} E_{i+1} =& EC_i^{(1)} \cup EC_i^{(2)} \\ & \cup \{(sc_{i+1,j}, sc_{i,j}^{(1)}) : j \in [2^i]\} \\ & \cup \{(sc_{i+1,j}, sc_{i,j}^{(2)}) : j \in [2^i]\} \\ & \cup \{(sc_{i+1,j+2^i}, sc_{i,j}^{(1)}) : j \in [2^i]\} \\ & \cup \{(sc_{i+1,j+2^i}, sc_{i,j}^{(2)}) : j \in [2^i]\}. \end{aligned}$$

Roughly speaking, we put two $(2^i, 2^{i-1})$ concentrators side by side, and the sink nodes are just the the sink nodes of both concentrators. Each half of the source nodes are connected to the source nodes of each smaller concentrator in a one-to-one fashion. See figure 6 for an illustration of the construction. This construction is compact, and has both in degree and out degree of 2.

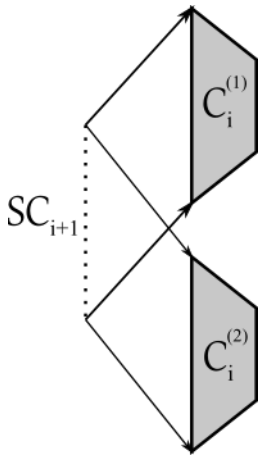


Figure 6: Construction of a $(2^{i+1}, 2^i)$-concentrator C_{i+1} from two copies of $(2^i, 2^{i-1})$-concentrators $C_i^{(1)}, C_i^{(2)}$. The sink nodes of C_{i+1} are the union of the sink nodes of $C_i^{(1)}, C_i^{(2)}$.

It is easy to verify the concentrator property of the construction. We want to show that for any subset of 2^i source nodes, there are 2^i vertex-disjoint paths between the subset of sources and the sink nodes. For any 2^i source nodes, we can always break them into two halves. We would "assign" the first half to go to the source nodes of the first smaller concentrator, and the second half to go to the second one. Notice that for each smaller concentrator, for any set of 2^{i-1} sources, there are 2^{i-1} vertex-disjoint paths between them and the sink nodes. The only remaining thing is to make sure we don't reuse vertices when "assigning" the source nodes. Here is a simple procedure: we first go through the chosen set of 2^i sources, and check if there are pairs of source nodes $sc_{i+1,j}$

and $sc_{i+1,j+2^i}$ that are 2^i indices apart. For all such pairs, we "assign" $sc_{i+1,j}$ to go to $sc^{(1)}_{i,j}$, and $sc_{i+1,j+2^i}$ to go to $sc^{(2)}_{i,j}$. For the rest of the chosen source nodes, we can arbitrarily let $sc_{i+1,j}$ to go to either $sc^{(1)}_{i,j}$ or $sc^{(2)}_{i,j}$, as long as we maintain that the total number of nodes "assigned" to each smaller concentrator is 2^{i-1}. In this way, these "assignments" have no shared vertices, and combined with the previous argument, we have that C_{i+1} fulfills the concentrator property.

Putting these pieces together from bottom up, we first construct $(2^n, 2^{n-1})$-concentrators, which then gives 2^n-hyperconcentrators, leading to 2^n-super--concentrators, and finally yielding the desired G_n. All of these constructions are compact, and G_n has an in degree of 2 as desired. □

Using this family of graphs, we then proceed to prove that DDSF is no in PSPACE by showing a DDSF problem not solvable using only polynomial space. Here we use a technique of Dwork, Naor, and Wee [DNW05] that converts a DAG into a Random-Oracle-based function using graph *labeling*. We use the following summarization of the technique from [BCS16].

Definition 21 (Labeling)**.** *Let $G = (V, E)$ be a DAG with max in-degree δ and a unique sink vertex, $x \in \{0,1\}^\ell$ be a string and $H : \mathbb{Z}_{|V|} \times (\{0,1\}^\ell \cup \{\bot\})^\delta \to \{0,1\}^\ell$ be a function modeled as a random oracle. We define the labeling of G relative to H and x as:*

$$\mathsf{label}_x(v_i) = \begin{cases} H(i, x, \bot, \ldots, \bot) & \text{If } v_i \text{ is a source node} \\ H(i, \mathsf{label}_x(u_1), \mathsf{label}_x(u_2), \ldots, \mathsf{label}_x(u_\delta)) & \text{Otherwise} \end{cases}$$

where $u_1, \ldots, u_\delta$ are the predecessors of v. And if v has less than δ predecessors, $\bot$'s are put in the input to H as placeholders.

We label the graph G using the labeling function by first starting at the source nodes, then proceed to their successors, so on and so forth until we label the unique sink node. To convert the graph G into a function f_G, we define $f_G(x)$ to be the function that outputs the label of the unique sink node under the labeling of G relative to the random oracle function H and input x.

Dwork, Naor, and Wee [DNW05] prove that if G is a graph that is infeasible to pebble using only T steps and S pebbles, then with high probability it is infeasible for an adversary to compute f_G using only $T' \approx T$ random oracle queries and space $S' \approx S\ell$. In the other direction, they show that if there is a pebbling strategy that uses T steps and S pebbles, then this yields an evaluation of f_G using only T queries to the random oracle H, and $S\ell$ memory bits. They consider a sequential pebbling game and sequential queries to the random oracle, but their result readily extends to the case with a parallel pebbling game and where one can make parallel queries to the random oracle.

With this, we prove the following theorem.

Theorem 3. *DDSF is not in* PSPACE*, assuming the random oracle model.*

Proof. We prove this by constructing a DSF in the random oracle model whose decisional variant cannot be solved in polynomial space.

Let λ be the security parameter, we construct the DSF based on the graph G_n from [PTC76] with $n = \sqrt{\lambda}$ using the technique by [DNW05]. However, notice that the graph G_n has 2^n sinks, so the first step is to augment the graph G_n to have a single sink node.

Augmenting the graph G_n. We first augment G_n to G'_n by attaching a top-down binary tree to the sink nodes. Specifically, let the sink nodes of G_n be $t_{0,1}, t_{0,2}, \ldots, t_{0,2^n}$. To get G'_n, we add nodes $T' = \{t_{1,1}, t_{1,2}, \ldots, t_{1,2^{n-1}}; t_{2,1}, \ldots, t_{2,2^{n-2}}; t_{3,1}, \ldots; t_{n-1,1}\}$. For each $t_{i,j}$ added, we also add edges $(t_{i-1,2j-1}, t_{i,j})$ and $(t_{i-1,2j}, t_{i,j})$. Notice that a legal pebbling $P' = (P'_0, \ldots, P'_t)$ of G'_n implies a legal pebbling $P = (P_0, \ldots, P_t)$ of G_n: all we need to do is for all $i \in [t]$, have $P_i = P'_i \backslash T'$, i.e. we remove all the pebbles on the extra nodes added for G'_n. P is a complete pebbling for G_n because in the successful pebbling P' of G'_n, we must have pebbled $t_{0,1}, t_{0,2}, \ldots, t_{0,2^n}$ at some point, so all the sink nodes in G_n are pebbled sometime in P. Also P is legal because any edge in G'_n is either also in G_n, which preserves the validity of a pebble placed by moving along this edge, or it points to a node in T, the pebbles on which are already deleted in P. Therefore, Lemma 18 also holds for G'_n.

Constructing function f_n from G'_n. With G'_n, we use the technique by [DNW05] to construct the function $f_n = f_{G'_n}$. Notice that by Lemma 18, any pebbling of the graph G'_n need at least $\Omega(2^n)$ pebbles, therefore, as shown by [DNW05], for any evaluation of the function f_n, with high probability, there must be some point where it uses approximately $\Omega(2^n \ell)$ memory bits. Consequently, with only polynomial space, one cannot evaluate the function f_n. Since f_n consists of iterated random oracle calls, the decisional problem is as hard as the computational problem. So to check if $y = f_n(x)$ for given function f_n, input x and output y, one would also need $\Omega(2^n \ell) = \Omega(2^{\sqrt{\lambda}} \ell)$ memory, which is super polynomial.

Verifying f_n is sequential. Lastly, it remains to show that f_n is indeed a DSF. To see this, we examine the depth of the graph G'_n from bottom up. The concentrator component C_n has depth $O(n)$, and the hyperconcentrator and superconcentrator components have depth $O(n^2)$. The graph G_n has depth $O(2^n + n^2) = O(2^n)$, and G'_n has depth $O(2^n + n) = O(2^n)$, meaning that to pebble G'_n, even in parallel, one would need at least $O(2^n)$ steps. By [DNW05], with high probability, to compute f_n also requires $O(2^n) = O(2^{\sqrt{\lambda}})$ batches of random oracle queries, giving us the desired sequentiality. We can set k of the DSF to be $k = 2^{\sqrt{\lambda}} = 2^{o(\lambda)}$ as needed.

Putting this together, consider the following DDSF problem. The problem consists of (a class of) DSFs $F = \{f_8, f_9, \ldots\}$, a pp specifying the input and output space, an input x, an output y, and $k \in 2^{o(\lambda)}$. The problem is true if and only if $\mathsf{Eval}(\mathsf{pp}, x, k) = y$. Eval for F works as follows: first, compute the n that gives the desired $k = O(2^n)$, and then compute and output $f_n(x)$ corresponding to the graph G'_n. By the above arguments, F is indeed a DSF, but solving this DDSF problem requires $\Omega(2^{\sqrt{\lambda}})$ super polynomial space.

Therefore, the DDSF problem is not in PSPACE. □

7 Fine-Grained Symmetric Key Cryptography

In this section, we define Symmetric Key Encryption (SKE) against fine-grained time bounded adversaries, which we construct from CISFs. We define fine-grained SKE as a stateful SKE with updating secret keys that proceeds in rounds. Let Alice and Bob have some shared secret. Before round 0, both Alice and Bob run a Setup procedure using their shared secret and get an initial secret key. At each round i, one party, say Alice w.l.o.g., runs Enc to encrypt a message using the

latest secret key, which can be decrypted by Bob using the latest secret key. At each round, both parties also run an Update procedure that outputs an updated secret key. Put formally:

Definition 22 (Fine-Grained SKE)**.** *Let λ, n be security parameters, and $s \in \mathcal{S}$ with $|\mathcal{S}| = 2^{poly(\lambda)}$ a shared secret between two honest parties. An n-time* fine-grained symmetric key encryption *(FGSKE) scheme for key space $\{0,1\}^{\ell_k}$ and message space $\{0,1\}^{\ell_m}$ is a tuple of algorithms $\Pi = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Enc}, \mathsf{Dec})$ defined as follows:*

- $\mathsf{Setup}(1^\lambda, 1^n, s) \to k_1$*: takes as input the security parameters and the shared secret, and outputs an initial secret key k_1.*
- $\mathsf{Update}(k_i) \to k_{i+1}$*: at round $i \in [n]$, takes as input the current secret key k_i and outputs the next secret key k_{i+1}.*
- $\mathsf{Enc}(k_i, m_i) \to \mathsf{ct}_i$*: at round $i \in [n]$, takes as input the current secret key k_i and a message m_i, and outputs a corresponding ciphertext ct_i.*
- $\mathsf{Dec}(k_i, \mathsf{ct}_i) \to m_i'$*: at round $i \in [n]$, takes as input the current secret key k_i and a ciphertext ct_i, and outputs the message m_i' associated with the ciphertext.*

We require *correctness* of the above FGSKE scheme.

Definition 23 (Correctness)**.** *A FGSKE scheme $\Pi = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Enc}, \mathsf{Dec})$ is said to be* correct *if for all $s \in \mathcal{S}$, $i \in [n]$ and $m \in \{0,1\}^{\ell_m}$, we have*

$$\Pr\left[m' = m : \begin{array}{c} k_1 \leftarrow \mathsf{Setup}(1^\lambda, 1^n, s) \\ k_i \leftarrow \mathsf{Update}^{(i-1)}(k_1) \\ m' \leftarrow \mathsf{Dec}(k_i, \mathsf{Enc}(k_i, m)) \end{array}\right] \geq 1 - negl(\lambda).$$

We define security through the following indistinguishability experiment $\mathsf{Dist}^{\mathsf{FGSKE}}_{\mathcal{A},\Pi}(\lambda, n)$:

1. Sample a uniform bit $b \in \{0,1\}$.
2. Sample a secret $s \in \mathcal{S}$ and run $\mathsf{Setup}(1^\lambda, 1^n, s)$ to obtain an initial key k_1.
3. For round $i = 1, 2, \ldots, n$,
   (a) At the beginning of each round, the adversary can choose to terminate the experiment by submitting a pair of challenge messages $m_{i,0}$ and $m_{i,1}$, and receiving $\mathsf{ct}_i \leftarrow \mathsf{Enc}(k_i, m_{i,b})$. Then *before the next round*, the adversary outputs a guess b' for b. If $b' = b$, we say that the adversary succeeds and the experiment outputs 1. Otherwise, the experiment outputs 0.
   (b) If the adversary choose not to terminate the experiment, the adversary submits a query message m_i, and receives $\mathsf{ct}_i \leftarrow \mathsf{Enc}(k_i, m_i)$.
   (c) Run $\mathsf{Update}(k_i)$ to obtain k_{i+1}.

Definition 24 (FGSKE Security)**.** *Let λ, n be security parameters. A FGSKE scheme $\Pi = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Enc}, \mathsf{Dec})$ is said to have n-time fine-grained SKE security if for all PPT adversaries $\mathcal{A}$:*

$$\Pr\left[\mathsf{Dist}^{\mathsf{FGSKE}}_{\mathcal{A},\Pi}(\lambda, n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Notice that the above notion is similar to the standard CPA indistinguishability security notion, except that here we have updating keys and that the adversary needs to output the bit guess b' in a more fine-grained time bound (before the next round starts) than PPT.

We show how to construct such fine-grained SKE from CISFs. To facilitate our security argument, we assume that the CISF is *ideal*, meaning that the Round function takes exactly the same time to evaluate for both the honest parties and the adversaries. Nevertheless, our construction is still secure if the CISF is not ideal, as long as the adversary only has some small advantage in computing the Round function than the honest parties.

Construction 1. *Let λ, n be security parameters. Given CISF $= (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Round})$ an ideal continuous iterative sequential function whose Round function takes exactly 1 round to evaluate and has input and output length $\ell = poly(\lambda)$, we construct a fine-grained SKE scheme $\Pi = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Enc}, \mathsf{Dec})$ for a single message bit as follows. First, to sample the shared secret s, first run CISF.$\mathsf{Setup}(1^\lambda)$ to get pp, and run CISF.$\mathsf{Gen}(\mathsf{pp})$ to get an input x. The shared secret s consists of pp and x, together with the description of the CISF. The rest of the algorithms work as follow:*

- $\mathsf{Setup}(1^\lambda, 1^n, s)$*: Take time n rounds to run $(CISF.\mathsf{Round}(\mathsf{pp}, \cdot))^{(n)}(x)$, and store all the intermediate results. Output $k_1 = (\{y_j\}_{j\in[n]}, 1)$ where $y_j = (CISF.\mathsf{Round}(\mathsf{pp}, \cdot))^{(j)}(x)$.*
- $\mathsf{Update}(k_i)$*: Compute $y_{i+1} = CISF.\mathsf{Round}(\mathsf{pp}, y_i)$, and output $k_{i+1} = (\{y_j\}_{j\in[n+i]}, i+1)$. Notice that this takes exactly one round.*
- $\mathsf{Enc}(k_i, m_i)$*: Sample a uniformly random $r_i \in \{0,1\}^\ell$, and compute $c_i = m_i \oplus \langle r_i, y_{2i-1}\rangle$, where $\oplus$ denotes the XOR operation and $\langle\cdot,\cdot\rangle$ denotes the inner product. Output $\mathsf{ct}_i = (c_i, r_i)$.*
- $\mathsf{Dec}(k_i, \mathsf{ct}_i = (c_i, r_i))$*: Simply output $m'_i = c_i \oplus \langle r_i, y_{2i-1}\rangle$.*

The correctness of the above construction should be easy to verify.

Theorem 4. *If CISF $= (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Round})$ is an ideal continuous iterative sequential function whose Round function takes exactly 1 round to evaluate, then Construction 1 has n-time fine-grained SKE security.*

Proof. Let i be the round that the adversary playing the FGSKE game outputs the bit guess. Recall that in round i, the adversary submits a pair of challenge messages $m_{i,0}$ and $m_{i,1}$, and receives $\mathsf{ct}_i = (c_i, r_i)$ where r_i is a uniform ℓ-bit string, and $c_i = m_{i,b} \oplus \langle r_i, y_{2i-1}\rangle$.

We first argue that y_{2i-1} has sufficient min-entropy by the property of the sequential function. Notice that the only possible information that the adversary has about the sequential function is that in the previous rounds $j = 1, 2, \ldots, i-1$, the adversary receives ct_j that is dependent on y_{2j-1}. Since now only $(i-j)$ rounds has passed, by the sequentiality of the CISF, the adversary can compute at most up to $y_{(2j-1)+(i-j)} = y_{i+j-1}$. Notice that $i + j - 1 < 2i - 1$, so the adversary cannot possibly have computed y_{2i-1} by round i. Therefore, by the sequentiality of the CISF, the adversary can only guess y_{2i-1} correctly with negligible probability. Consequently, conditioned on the adversary's view $\mathsf{view}_\mathcal{A}$, we have $H_\infty(y_{2i-1}, r_i|\mathsf{view}_\mathcal{A}) \geq H_\infty(y_{2i-1}|\mathsf{view}_\mathcal{A}) = \Omega(\ell)$.

Once we have that y_{2i-1} has sufficient min-entropy, we can invoke the Leftover Hash Lemma (Lemma 1). Let $H : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}$ compute the inner product. Using the fact that the inner product is a universal hash function and applying Lemma 1, we have

$$(H(y_{2i-1}, r_i), H, \mathsf{view}_\mathcal{A}) \approx_{\epsilon/2} (U_1, U_d, \mathsf{view}_\mathcal{A}),$$

where $1 + 2\log(1/\epsilon) = \Omega(\ell)$. Solving for ϵ yields that $\epsilon = O(2^{-\ell/2})$, i.e. an adversary has advantage at most $O(2^{-\ell/2})$ in distinguishing $\langle y_{2i-1}, r_i \rangle$ and a uniform bit.

Therefore, with overwhelming probability, we can switch $\langle y_{2i-1}, r_i \rangle$ to a uniform bit and the adversary won't be able to detect the switch. But now the ciphertext is just the message XOR'ed with a uniform bit. This is simply One Time Pad (OTP) encryption, and is information theoretically secure. Hence the adversary cannot produce $b' = b$ other than a random guess. □

Fine-grained MACs can be defined analogously, and our construction follows almost immediately from our FGSKE construction. We present the definitions and constructions of fine-grained MACs in section 8.

8 Fine-Grained MAC

We define a fine-grained MAC similarly as to how we defined a fine-grained SKE.

Definition 25 (Fine-Grained MAC)**.** *Let λ, n be security parameters, and $s \in \mathcal{S}$ with $|\mathcal{S}| = 2^{poly(\lambda)}$ a shared secret between two honest parties. An n-time* fine-grained MAC *(FGMAC) scheme for key space $\{0,1\}^{\ell_k}$ and message space $\{0,1\}^{\ell_m}$ is a tuple of algorithms $\Pi = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ defined as follows:*

- $\mathsf{Setup}(1^\lambda, 1^n, s) \to k_1$*: takes as input the security parameters and the shared secret, and outputs an initial secret key k_1.*
- $\mathsf{Update}(k_i) \to k_{i+1}$*: at round $i \in [n]$, takes as input the current secret key k_i and outputs the next secret key k_{i+1}.*
- $\mathsf{Sign}(k_i, m_i) \to \sigma_i$*: at round $i \in [n]$, takes as input the current secret key k_i and a message m_i, and outputs a corresponding MAC tag ct_i.*
- $\mathsf{Verify}(k_i, m_i, \sigma_i) \to 1/0$*: at round $i \in [n]$, takes as input the current secret key k_i, a message m_i, and a tag σ_i, and outputs a bit indicating whether the tag is valid on the message.*

We require *correctness* of the above FGMAC scheme.

Definition 26 (Correctness)**.** *A FGMAC scheme $\Pi = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ is said to be* correct *if for all $s \in \mathcal{S}$, $i \in [n]$ and $m \in \{0,1\}^{\ell_m}$, we have*

$$\Pr\left[\begin{array}{c} k_1 \leftarrow \mathsf{Setup}(1^\lambda, 1^n, s) \\ \mathsf{Verify}(k_i, m, \sigma) = 1 : k_i \leftarrow \mathsf{Update}^{(i-1)}(k_1) \\ \sigma \leftarrow \mathsf{Sign}(k_i, m) \end{array}\right] \geq 1 - negl(\lambda).$$

We define security through the MAC forgery experiment $\mathsf{Forge}^{\mathsf{FGMAC}}_{\mathcal{A},\Pi}(\lambda, n)$:

1. Sample a secret $s \in \mathcal{S}$ and run $\mathsf{Setup}(1^\lambda, 1^n, s)$ to obtain an initial key k_1.
2. For round $i = 1, 2, \ldots, n$,

(a) At the beginning of each round, the adversary can choose to terminate the experiment by submitting a message m and a tag σ. If $\mathsf{Verify}(k_i, m, \sigma) = 1$, we say that the adversary succeeds and the experiment outputs 1. Otherwise, the experiment outputs 0.

(b) If the adversary choose not to terminate the experiment, the adversary submits a query message m_i, and receives $\sigma_i \leftarrow \mathsf{Sign}(k_i, m_i)$.

(c) Run $\mathsf{Update}(k_i)$ to obtain k_{i+1}.

Definition 27 (FGMAC Security)**.** *Let λ, n be security parameters. A FGMAC scheme $\Pi = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ is said to have n-time fine-grained MAC security if for all PPT adversaries $\mathcal{A}$:*

$$\Pr\left[\mathsf{Forge}^{\mathsf{FGMAC}}_{\mathcal{A},\Pi}(\lambda, n) = 1\right] \leq \mathsf{negl}(\lambda).$$

Notice that the above definition is similar to a standard model MAC security definition, except that we deal with an adversary with a fine-grained time bound and that the forgery produced could be on a message that has been queried before. The latter is possible because the keys are updated each round, so even if a message has been MACed before, its old MAC would not verify under the current updated key.

As we sketched in the technical overview, our construction effectively follows from our FGSKE construction. Recall that in our FGSKE construction we run a CISF iteratively to produce many unpredictable bits and then use an inner product as an extractor to extract the randomness which we XOR with the message. Here for the FGMAC construction, we run the CISF iteratively starting from two different inputs and produce two separate chains of unpredictable bits. To MAC a message bit b, the tag is simply the bits from one of the chains, depending on the bit b. To make a successful forgery, the adversary needs to predict the output of a CISF in a bounded time, which is hard due to the CISF security guarantee.

Below we present the construction and security argument in details.

Construction 2. *Let λ, n be security parameters. Given CISF $= (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Round})$ an ideal continuous iterative sequential function whose Round function takes exactly 1 round to evaluate and has input and output length $\ell = poly(\lambda)$, we construct a fine-grained MAC scheme $\Pi = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ for a single message bit as follows. First, to sample the shared secret s, first run CISF.$\mathsf{Setup}(1^\lambda)$ to get pp, and run CISF.$\mathsf{Gen}(\mathsf{pp})$ twice to get two inputs x_0, x_1. The shared secret s consists of pp and x_0, x_1, together with the description of the CISF. The rest of the algorithms work as follow:*

- $\mathsf{Setup}(1^\lambda, 1^n, s)$*: First, run* $(\mathit{CISF}.\mathsf{Round}(\mathsf{pp}, \cdot))^{(n)}(x_0)$ *and* $(\mathit{CISF}.\mathsf{Round}(\mathsf{pp}, \cdot))^{(n)}(x_1)$*. This takes n rounds and store all the intermediate results. Then, output* $k_1 = (\{y_{j,b}\}_{j\in[n],b\in\{0,1\}}, 1)$ *where* $y_{j,b} = (\mathit{CISF}.\mathsf{Round}(\mathsf{pp}, \cdot))^{(j)}(x_b)$*.*
- $\mathsf{Update}(k_i)$*: Compute* $y_{i+1,b} = \mathit{CISF}.\mathsf{Round}(\mathsf{pp}, y_{i,b})$ *for* $b \in \{0, 1\}$*, and output the updated key* $k_{i+1} = (\{y_{j,b}\}_{j\in[n+i],b\in\{0,1\}}, i + 1)$*. Notice that this takes exactly one round.*
- $\mathsf{Sign}(k_i, m_i)$*: Simply output* $\sigma_i = y_{2i-1,m_i}$*.*
- $\mathsf{Verify}(k_i, m_i, \sigma_i)$*: Check if* $\sigma_i = y_{2i-1,m_i}$*. If equal, output* 1*. Otherwise, output* 0*.*

Correctness. Correctness follows immediately due to the fact that CISFs are deterministic.

Notice that this construction is wildly inefficient. To MAC a single message bit, the tag itself is $poly(\lambda)$ bits. But to do better than this seemingly requires a CRHF, which would imply a OWF, which we are not assuming. Thus, this construction is presented more out of theoretical interest in exploring how to construct a MAC from minimal assumptions in the fine-grained setting.

Security. Security is also relatively straightforward to argue, which we do with the following theorem.

Theorem 5. *If CISF* $= (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Round})$ *is an ideal continuous iterative sequential function whose* Round *function takes exactly* 1 *round to evaluate, then Construction 2 has* n*-time fine-grained MAC security.*

Proof. Let i be the round that the adversary playing the FGMAC game outputs their forgery. W.L.O.G., assume the message that the adversary MACs is 0. In order for the forgery to be valid, the adversary needs to produce $y_{2i-1,0}$ in round i.

Notice that the only possible information that the adversary has about the sequential function is that in the previous rounds $j = 1, 2, \ldots, i-1$, the adversary receives σ_j that could be dependent on $y_{2j-1,0}$. Since now only $(i-j)$ rounds has passed, by the sequentiality of the CISF, the adversary can compute at most up to $y_{(2j-1)+(i-j),0} = y_{i+j-1,0}$. Notice that $i+j-1 < 2i-1$, so the adversary cannot possibly have computed $y_{2i-1,0}$ by round i. Therefore, by the sequentiality of the CISF, the adversary can only produce $y_{2i-1,0}$ correctly with negligible probability. Hence, the adversary can win the FGMAC game with only negligible probability. □

9 Fine-Grained Public Key Encryption

In this section, we define public key encryption that is secure against a fine-grained time-bounded adversary. The syntax of the encryption scheme is the same as PKE in the standard model, except that instead of requiring the adversary $\mathcal{A}$ to operate in PPT, here we require the adversary to run in sequential time T.

9.1 Definition

We imagine the following definition for a public key encryption scheme against fine-grained time-bounded adversary. Consider the following indistinguishability experiment for an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$:

PKE Security Experiment $\mathsf{Dist}^{\mathsf{PKE}}_{\mathcal{A},\Pi}(\lambda)$:

1. Run $\mathsf{Gen}(1^\lambda, 1^S)$ to obtain keys $(\mathsf{pk}, \mathsf{sk})$.
2. Sample a uniform bit $b \in \{0, 1\}$.
3. The adversary $\mathcal{A}$ is provided the public key pk.
4. The adversary replies with the challenge query consisting of two messages m_0 and m_1, receives $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, m_b)$.

5. $\mathcal{A}$ outputs a guess b' for b. If $b' = b$, we say that the adversary succeeds and the output of the experiment is 1. Otherwise, the experiment outputs 0.

Definition 28 (PKE Security against fine-grained time-bounded adversary)**.** *For security parameter λ, a public key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is secure against fine-grained time-bounded adversarywith time bound T if for all PPT adversaries $\mathcal{A}$ with running time less than T:*

$$\Pr\left[\mathsf{Dist}^{\mathsf{PKE}}_{\mathcal{A},\Pi}(\lambda) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

9.2 Construction

Here we give a construction of PKE against fine-grained time-bounded adversaryusing Verifiable Delay Functions (VDFs) and indistinguishability Obfuscation (iO) for null circuits. In addition the the completeness, soundness and sequentiality of the VDF, we also require the following properties of the VDF.

Definition 29 (Uniqueness of Proof)**.** *A VDF $V = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ has a unique proof if for all $\lambda \in \mathbb{N}$ and for all PPT machines $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function negl such that:*

$$\Pr\left[\begin{array}{l|l} \mathsf{Vf}(\mathsf{pp}, x, y, \pi', T) = 1 \text{ and } \pi \neq \pi' & \begin{array}{l}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)\\ (T, x) \leftarrow \mathcal{A}_1(\mathsf{pp})\\ (y, \pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x, T)\\ \pi' \leftarrow \mathcal{A}_2(\mathsf{pp}, x, y, T)\end{array}\end{array}\right] = negl(\lambda)$$

Notice that by soundness of a VDF, the output y is unique on a fixed input, and by the uniqueness proof, we have that the proof π is also unique given the output y. Additionally, we require that a real valid proof should be indistinguishable from a random proof if the adversary is bounded by sequential time T.

Definition 30 (Proof Indistinguishability)**.** *A VDF $V = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ has indistinguishable proofs if for all $\lambda \in \mathbb{N}$ and for all PPT machines $\mathcal{A}$ with time bound T, there exists a negligible function negl such that:*

$$\Pr\left[\begin{array}{l|l} \mathcal{A}(\mathsf{pp}, x, \pi, T) = 1 & \begin{array}{l}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)\\ x \leftarrow \mathsf{Gen}(\mathsf{pp})\\ (y, \pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x, T)\end{array}\end{array}\right] -$$

$$\Pr\left[\begin{array}{l|l} \mathcal{A}(\mathsf{pp}, x, \pi, T) = 1 & \begin{array}{l}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)\\ x \leftarrow \mathsf{Gen}(\mathsf{pp})\\ \pi \xleftarrow{\$} \Pi\end{array}\end{array}\right] = negl(\lambda)$$

Construction 3. *Let $i\mathcal{O}$ be an indistinguishability obfuscator for null circuits and $\mathsf{VDF} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ a verifiable delay function with proof indistinguishability and proof uniqueness. We construct our PKE as follows:*

- $\mathsf{Gen}(1^\lambda)$*: Run* $\mathsf{pp} \leftarrow \mathsf{VDF.Setup}(1^\lambda)$*, and then sample the a VDF input* $x \leftarrow \mathsf{VDF.Gen}(\mathsf{pp})$*. Compute the output and the proof* $(y, \pi) \leftarrow \mathsf{VDF.Eval}(\mathsf{pp}, x, T)$*. The public key is* $\mathsf{pk} = (x, \pi)$*, and the private key is* $\mathsf{sk} = y$*.*

- $\mathsf{Enc}(\mathsf{pk} = (x, \pi), m)$*: To encrypt a message, consider the following program*

$$P_{\mathsf{pp},T,x,\pi,m_b}(y') = \begin{cases} m & \text{if } \mathsf{VDF.Vf}(\mathsf{pp}, \mathsf{pk}, y', \pi, T) = 1 \\ \bot & \text{otherwise} \end{cases}.$$

  The ciphertext is simply $i\mathcal{O}(P_{\mathsf{pp},T,\mathsf{pk},\pi,m_b})$*.*

- $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$*: To decrypt a ciphertext, simply evaluate it as an iO program using* $\mathsf{sk} = y$ *as the input, and the output of the iO program is simply the message* m*.*

Correctness is straightforward given the correctness of the VDF and iO. Next we show this construction is secure against fine-grained time-bounded adversary.

9.3 Proof of Security

We organize our proof of security into a sequence of hybrids.

Sequence of Hybrids

- H_0: The original PKE security game, where the challenge bit is fixed to be 0.
- H_1: The same as H_0, except that instead of sampling using the actual proof π, sample a uniformly random π' from the proof space Π. The program $P_{\mathsf{pp},T,x,\pi',m_b}$ now embeds the random proof π' instead of the actual proof π.
- H_2: Switch the program $P_{\mathsf{pp},T,x,\pi',m_b}$ to always output $\bot$.
- H_3: We switch the challenge bit from 0 to 1.
- H_4: Switch the program $P_{\mathsf{pp},T,x,\pi',m_b}$ back to output m_b if $\mathsf{VDF.Vf}(\mathsf{pp}, \mathsf{pk}, y', \pi', T) = 1$.
- H_5: Switch back to using the actual proof π instead of the random proof π'.

Proof of Hybrid Arguments

Lemma 20. *If* VDF *is a verifiable delay function with proof indistinguishability, then no PPT adversary with time bound* T *can distinguish between* H_0 *and* H_1 *(and hence* H_4 *and* H_5*) with non-negligible probability.*

Proof. This step follows directly from the definition of proof indistinguishability. We show that if an adversary $\mathcal{A}$ can distinguish between H_0 and H_1, then we can construct an adversary $\mathcal{A}'$ that breaks the proof indistinguishability property by using $\mathcal{A}$ as a subroutine.

$\mathcal{A}'$ receives as input pp, x, π, T, and works as follows. It first forwards (x, π) to $\mathcal{A}$ as the public key pk, and receives the challenge query m_0 and m_1 from $\mathcal{A}$. $\mathcal{A}'$ then constructs the program $P_{\mathsf{pp},T,x,\pi,m_b}$ and then sends $i\mathcal{O}(P_{\mathsf{pp},T,\mathsf{pk},\pi,m_b})$ back to $\mathcal{A}$. If $\mathcal{A}$ outputs that it is in H_0, $\mathcal{A}'$ outputs that π is a real proof. Otherwise, output that π is a random proof. □

Lemma 21. *If* $i\mathcal{O}$ *is an indistinguishability obfuscator for null circuits and* VDF *is a verifiable delay function with perfect soundness and unique proofs, then no PPT adversary can distinguish between* H_1 *and* H_2 *(and hence* H_3 *and* H_4*) with non-negligible probability.*

Proof. This steps comes from the iO security of the underlying obfuscator. Notice that the only difference between H_1 and H_2 is the program being obfuscated. In H_1, the obfuscated program outputs m_b if $\mathsf{VDF.Vf}(\mathsf{pp}, \mathsf{pk}, y', \pi', T) = 1$, and in H_2, the obfuscated program always outputs $\perp$. If we can show that the program in H_1 always outputs $\perp$, then by the iO security of the obfuscator, no PPT adversary can distinguish them with non-negligible probability.

We need to show that $\mathsf{VDF.Vf}(\mathsf{pp}, \mathsf{pk}, y', \pi', T)$ always fails. Notice that for all input $y' \neq y$, by the soundness of the VDF, $\mathsf{VDF.Vf}(\mathsf{pp}, \mathsf{pk}, y', \pi', T) = 0$. Then for $y' = y$, since in H_1 already have π' being a random proof, then by the uniqueness of the proof, $\mathsf{VDF.Vf}(\mathsf{pp}, \mathsf{pk}, y', \pi', T) = 0$. Therefore, $\mathsf{VDF.Vf}(\mathsf{pp}, \mathsf{pk}, y', \pi', T)$ always fails. □

Lemma 22. *No adversary can distinguish between H_2 and H_3.*

Proof. The only difference between H_2 and H_3 the challenge bit b. But in neither hybrid does the ciphertext depend on the message m_b. Therefore, no adversary can distinguish between these two hybrids. □

Theorem 6. *If* VDF *is a verifiable delay function with perfect soundness, unique proofs, and proof indistinguishability, and $i\mathcal{O}$ is an indistinguishability obfuscator, then Construction 3 is secure against fine-grained time-bounded adversary.*

Proof. The lemmas above show a sequence of a polynomial number of hybrid experiments where no PPT adversary bounded by sequential time T can distinguish one from the next with non-negligible probability. Notice that the first hybrid H_0 corresponds to the PKE security game where $b = 0$, and the last hybrid H_5 corresponds to one where $b = 1$. The security of the indistinguishability game follows. □

10 Key Exchange

In this section, we define an non-interactive key exchange (NIKE) protocol against a fine-grained time-bounded adversary. We show that such a NIKE protocol can be built from what we call a *commutative sequential function.*

Inspiration for This Construction. Suppose we have some group of unknown order G, with some order less than some prime q. We could attempt to construct a fine-grained key exchange between two players Alice and Bob in the following way:

- Fix some public group element $g \in G$.
- Alice and Bob each sample some elements $a, b \in \mathbb{Z}_q$.
- Alice and Bob each compute g^{a^T} and g^{b^T}, respectively.
- Alice sends a to Bob and Bob sends b to Alice.
- Alice and Bob both compute $g^{(ab)^T}$

This protocol seems interesting, because if we assume that repeated squaring is hard, then it might seem like given g, a, and b, an adversary would need time $2T$ to compute $g^{(ab)^T}$–and not time T. Unfortunately this is not the case–this construction is *not* a fine-grained secure key exchange, and there exists attacks by using binary decomposition on the exponents a and b. However, the intuition behind this construction seemed interesting and illuminating for fine-grained key exchange, so we define the general primitive and explain its relationship to key exchange here.

10.1 Commutative Sequential Functions

Commutative sequential functions (CSFs) are a class of sequential function where the function evaluations observe commutativity. In a nutshell, there are four functions f_1, f_2, g_1, g_2 which are individually and compositionally sequential with the additional constraint that, for all inputs x, $f_2(g_1(x)) = g_2(f_1(x))$. We emphasize that this enforces some composability properties which we define in detail below.

Definition 31 (Commutative Sequential Functions)**.** *A **commutative sequential function***

$$F := (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sample}_1, \mathsf{Sample}_2, \mathsf{Eval})$$

is defined as the following tuple of algorithms:

$\mathsf{Setup}(1^\lambda, k) \to \mathsf{pp}$**:** *On input the security parameter 1^λ and $k \in 2^{o(\lambda)}$, the setup algorithm returns the public parameters pp. By convention, the public parameters encode the input domains X, X_f, X_g and output domain Y and the classes of functions $\mathcal{F}_1 : X \to X_f$, $\mathcal{F}_2 : X_f \to Y$, $\mathcal{G}_1 : X \to X_g$, and $\mathcal{G}_2 : X_g \to Y$.*

$\mathsf{Gen}(\mathsf{pp}, k) \to x$**:** *On input the public parameters pp, and $k \in 2^{o(\lambda)}$, the instance generation algorithm samples a random input $x \leftarrow X$.*

$\mathsf{Sample}_1(\mathsf{pp}, k) \to (f_1, f_2)$**:** *On input the public parameters pp , and $k \in 2^{o(\lambda)}$, the function sampling algorithm samples a pair of functions $f_1 \leftarrow \mathcal{F}_1$ and $f_2 \leftarrow \mathcal{F}_2$.*

$\mathsf{Sample}_2(\mathsf{pp}, k) \to (g_1, g_2)$**:** *On input the public parameters pp , and $k \in 2^{o(\lambda)}$, the function sampling algorithm samples a pair of functions $g_1 \leftarrow \mathcal{G}_1$ and $g_2 \leftarrow \mathcal{G}_2$.*

$\mathsf{Eval}(\mathsf{pp}, h, x, k) \to y$**:** *On input the public parameters pp, a function handle $h \in \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{G}_1, \mathcal{G}_2\}$, an input x, and $k \in 2^{o(\lambda)}$, the evaluation algorithm returns an output y.*

In addition to the usual sequential function properties as defined in 3, we require commutativity of the function evaluations.

Definition 32 (Commutativity)**.** *A sequential function $F = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sample}_1, \mathsf{Sample}_2, \mathsf{Eval})$ is commutative if for all security parameters $\lambda \in \mathbb{N}$, the following holds:*

$$\Pr\left[\, y = y' \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda), \\ (f_1, f_2) \leftarrow \mathsf{Sample}_1(\mathsf{pp}, k),\ (g_1, g_2) \leftarrow \mathsf{Sample}_2(\mathsf{pp}, k), \\ x \leftarrow \mathsf{Gen}(\mathsf{pp}) \\ y \leftarrow \mathsf{Eval}(\mathsf{pp}, g_2, \mathsf{Eval}(\mathsf{pp}, f_1, x, k), k), \\ y' \leftarrow \mathsf{Eval}(\mathsf{pp}, f_2, \mathsf{Eval}(\mathsf{pp}, g_1, x, k), k) \end{array} \right] = 1.$$

We also need to define the security of a commutative sequential function. To do this in a fine-grained way, we unfortunately need to add a number of parameters.

Definition 33 (Security)**.** *Let $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma$ be rational numbers. Fix some particular model of computation*[1] *and fix some k, and let α_1 be the time it takes to compute any $f_1 \in \mathcal{F}_1$, let α_2 be the time it takes to compute any $f_2 \in \mathcal{F}_2$, let β_1 be the time it takes to compute any $f_1 \in \mathcal{G}_1$, and let β_2 be the time it takes to compute any $f_1 \in \mathcal{G}_2$, where this time is measured in a particular model of computation and is the time required by an honest user to evaluate the function.*

We say that a commutative sequential function is $(\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma)$-secure if, for some $\gamma > \alpha_2, \beta_2$, no adversary (bounded by the model of computation) can win the following game played between a challenger and the adversary with non-negligible probability:

- *The challenger samples a random input $x \in X$ and sends a description of the public parameters and relevant function families as well as the input x to the adversary.*
- *The adversary has arbitrary polynomial time to preprocess on these parameters.*
- *The challenger samples $f_1 \leftarrow \mathcal{F}_1, f_2 \leftarrow \mathcal{F}_2, g_1 \leftarrow \mathcal{G}_1, g_2 \leftarrow \mathcal{G}_2$.*
- *The challenger sends the tuple (f_2, g_2) to the adversary.*
- *Within time γ, the adversary outputs a value v.*

We say that the adversary wins the commutative sequential function security game if

$$v = f_2\left(g_1\left(x\right)\right) = g_2\left(f_1\left(x\right)\right).$$

We note that the secrets here are the *choices of function from the function family.* For fixed functions, we can just encode these choices as parameters.

10.2 Fine-Grained NIKE

We next describe a NIKE protocol with fine-grained security. The syntax is very similar to a traditional key exchange definition with the exception that we need to carefully keep track of the times of evaluation for all of the functions involved.

Definition 34. *A **fine-grained noninteractive key exchange protocol** (FGNIKE) is a tuple of algorithms $(\mathsf{Setup}, \mathsf{KeyGen}_1, \mathsf{KeyGen}_2, \mathsf{Combine}_1, \mathsf{Combine}_2)$ defined as follows (we allow all of the algorithms to take randomness as input but omit this in our description):*

$\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$**:** *On input the security parameter 1^λ the setup algorithm returns the public parameters pp. By convention, the public parameters encode the domains PP, X_1, X_2, S_1, S_2 and output domain Y.*

$\mathsf{KeyGen}_1(\mathsf{pp}) \to X_1 \times S_1$**:** *on input the public parameters pp, the function outputs a public key share x_1 and a secret s_1.*

$\mathsf{KeyGen}_2(\mathsf{pp}) \to X_2 \times S_2$**:** *on input the public parameters pp, the function outputs a public key share x_2 and a secret s_2.*

[1] The need to fix a particular model of computation for sequential functions is discussed extensively in [JMRR21], and we refer the reader to the discussion there for more details

$\mathsf{Combine}_1(\mathsf{pp}, S_1, X_2) \to Y$**:** *on input the public parameters* pp*, a secret* s_1*, and a public key share* x_2*, the function outputs a key* y*.*

$\mathsf{Combine}_2(\mathsf{pp}, S_2, X_1) \to Y$**:** *on input the public parameters* pp*, a secret* s_2*, and a public key share* x_1*, the function outputs a key* y*.*

Definition 35 (Correctness)**.** *A FGNIKE protocol* $(\mathsf{Setup}, \mathsf{KeyGen}_1, \mathsf{KeyGen}_2, \mathsf{Combine}_1, \mathsf{Combine}_2)$ *is* correct *if for all security parameters* $\lambda \in \mathbb{N}$*, the following holds:*

$$\mathsf{Pr}\left[\begin{array}{c|c} y_1 = y_2 & \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda), \\ (x_1, s_1) \leftarrow \mathsf{KeyGen}_1(\mathsf{pp}),\ (x_2, s_2) \leftarrow \mathsf{KeyGen}_2(\mathsf{pp}), \\ y_1 \leftarrow \mathsf{Combine}_1(\mathsf{pp}, s_1, x_2), \\ y_2 \leftarrow \mathsf{Combine}_2(\mathsf{pp}, s_2, x_1) \end{array} \end{array}\right] = 1.$$

We can now define the security of the FGNIKE protocol. Again, we are forced to use many parameters due to the fine-grained nature of the protocol.

Definition 36 (Security)**.** *Let* $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma$ *be rational numbers. Fix some particular model of computation, and let* α_1 *be the time it takes to compute any* KeyGen_1*, let* α_2 *be the time it takes to compute* KeyGen_2*, let* β_1 *be the time it takes to compute* $\mathsf{Combine}_1$*, and let* β_2 *be the time it takes to compute* $\mathsf{Combine}_2$*, where this time is measured in a particular model of computation and is the time required by an honest user to evaluate the function.*

We say that a FGNIKE is $(\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma)$*-secure if, for some* $\gamma > \alpha_2, \beta_2$*, no adversary (bounded by the model of computation) can win the following game played between a challenger and the adversary with non-negligible probability:*

- *The challenger samples* $\mathsf{pp} \leftarrow \mathsf{Setup}\left(1^\lambda\right)$
- *The adversary has arbitrary polynomial time to preprocess on* pp*.*
- *The challenger samples* $(x_1, s_1) \leftarrow \mathsf{KeyGen}_1\left(\mathsf{pp}\right)$ *and* $(x_2, s_2) \leftarrow \mathsf{KeyGen}_2\left(\mathsf{pp}\right)$*.*
- *The challenger sends the tuple* (x_1, x_2) *to the adversary.*
- *Within time* γ*, the adversary outputs a value* y'*.*

We say that the adversary wins the commutative sequential function security game if

$$y' = \mathsf{Combine}_1\left(\mathsf{pp}, s_1, x_2\right).$$

Note than in a typical key exchange protocol, $\alpha_1, \alpha_2, \beta_1, \beta_2$ would all be functions that are polynomial in the security parameter λ, and γ would be some function that is ideally exponential (but more likely subexponential) in γ. Since γ in our fine-grained applications would be polynomial in λ, it is important that we spell out all of the other parameters, particularly the βs.

10.3 Construction of the KE Protocol

Now we show how to construct a two-party NIKE protocol using a commutative sequential functions. The high-level idea is that each party will sample different function handles (f_1, f_2) and (g_1, g_2), and compute $f_1(x)$ and $g_1(x)$ on some agreed public input x. Then, they will exchange their function

descriptions f_2 and g_2, and eventually computing $f_2(g_1(x)) = g_2(f_1(x))$. For the honest parties, if we had an "ideally" secure sequential function, it would take time T to arrive at the shared key after the communication, but for an eavesdropper, it would take time $2T$, yielding the sequential time gap of T between the honest parties and the adversary.

Construction 4. *Let* $F = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sample}_1, \mathsf{Sample}_2, \mathsf{Eval})$ *be a commutative sequential function, we construct the non-interactive key exchange protocol as follows:*

- *Fix some appropriate security parameter* λ *and some integer* k*.*
- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$*: Run* $F.\mathsf{pp} \leftarrow F.\mathsf{Setup}(1^\lambda, k)$ *and* $x \leftarrow F.\mathsf{Gen}(F.\mathsf{pp}, k)$*. Set* $\mathsf{pp} = (F.\mathsf{pp}, x)$*.*
- $\mathsf{KeyGen}_i(\mathsf{pp}) \to (\mathsf{pv}_i, \mathsf{sv}_i)$*: For user* $i = 1$*, run* $F.\mathsf{Sample}_1(\mathsf{pp}, k)$ *to obtain function handles* (f_1, f_2)*, and compute the secret value* $\mathsf{sv}_1 = F.\mathsf{Eval}(\mathsf{pp}, f_1, x, k)$*. Publish the public value* $\mathsf{pv}_1 = f_2$*. For user* $i = 2$*, run* $F.\mathsf{Sample}_2(\mathsf{pp}, k)$ *to obtain* (g_1, g_2)*, compute* $\mathsf{sv}_2 = F.\mathsf{Eval}(\mathsf{pp}, g_1, x, k)$ *and publish* $\mathsf{pv}_2 = g_2$*.*
- $\mathsf{Combine}_i(\mathsf{pp}, \mathsf{sv}_i, \mathsf{pv}_{3-i}) \to \mathsf{key}$*: For user* $i = 1$*, compute the shared key as* $\mathsf{key} = F.\mathsf{Eval}(\mathsf{pp}, \mathsf{pv}_2, \mathsf{sv}_1, k) = F.\mathsf{Eval}(\mathsf{pp}, g_2, F.\mathsf{Eval}(\mathsf{pp}, f_1, x, k), k)$*. For user* $i = 2$*, compute* $\mathsf{key} = F.\mathsf{Eval}(\mathsf{pp}, \mathsf{pv}_1, \mathsf{sv}_2, k) = F.\mathsf{Eval}(\mathsf{pp}, f_2, F.\mathsf{Eval}(\mathsf{pp}, g_1, x, k), k)$*.*

Correctness of the scheme follows directly from the commutativity of the sequential function.

Security of the scheme follows from the security definition of the commutative sequential function. Notice that the honest parties arrive at the shared key in time (α_2, β_2) after both the $\mathsf{Combine}$ algorithms are run. But in order for the adversary to compute the shared key, it would need to compute either $f_2(g_1(x))$ or $g_2(f_1(x))$, none of which can be started before the pv's are published. Therefore, the adversary need at least γ sequential time to compute the shared key. Hence, at time less than γ after the $\mathsf{Combine}$ algorithms are run, the adversary has no information about the shared key.

10.4 (Lack of) Constructions and Extensions

We unfortunately were not able to develop a construction of a commutative sequential function that wasn't already a "regular" key exchange protocol, and we leave this as an interesting open problem. We had to make quite a lot of strong assumptions to build even fine-grained PKE that potentially did not imply one-way functions, and it is not clear even what sort of primitives would imply fine-grained key exchange. Some kind of VBB obfuscation protocol would most likely suffice, but it is not clear whether we could translate this to an iO construction.

We note that the definitions of both commutative sequential functions and fine-grained key exchange can be extended to involve multiple parties. However, for the sake of brevity (and because we lack concrete constructions), we omit these here.

It would also be nice to show some sort of equivalence between commutative sequential functions and fine-grained NIKE. While they are very similar in spirit, the notation of commutative sequential functions makes this difficult and any such constructions are seemingly very artificial.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[AB16] Joël Alwen and Jeremiah Blocki. Efficiently computing data-independent memory-hard functions. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 241–271. Springer, Heidelberg, August 2016.

[ABBK17] Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. In Chris Umans, editor, *58th FOCS*, pages 192–203. IEEE Computer Society Press, October 2017.

[ABC22] Arasu Arun, Joseph Bonneau, and Jeremy Clark. Short-lived zero-knowledge proofs and signatures. *Cryptology ePrint Archive*, 2022.

[ABP18] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Sustained space complexity. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 99–130. Springer, Heidelberg, April / May 2018.

[ACM22] Thomas Agrikola, Geoffroy Couteau, and Sven Maier. Anonymous whistleblowing over authenticated channels. LNCS, pages 685–714. Springer, Heidelberg, 2022.

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in nc^0. *SIAM Journal on Computing*, 36(4):845–888, 2006.

[AS15] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 595–603. ACM Press, June 2015.

[AW14] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th FOCS*, pages 434–443. IEEE Computer Society Press, October 2014.

[AWY15] Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu. Matching triangles and basing hardness on an extremely popular conjecture. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 41–50. ACM Press, June 2015.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, August 2018.

[BC22] Chris Brzuska and Geoffroy Couteau. On building fine-grained one-way functions from strong average-case hardness. In *EUROCRYPT 2022, Part II*, LNCS, pages 584–613. Springer, Heidelberg, June 2022.

[BCS16] Dan Boneh, Henry Corrigan-Gibbs, and Stuart E. Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 220–248. Springer, Heidelberg, December 2016.

[BD21] Jeffrey Burdges and Luca De Feo. Delay encryption. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 302–326. Springer, Heidelberg, October 2021.

[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, Part II*, volume 11892 of *LNCS*, pages 407–437. Springer, Heidelberg, December 2019.

[BDGM20] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Candidate iO from homomorphic encryption schemes. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 79–109. Springer, Heidelberg, May 2020.

[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, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

[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*, pages 345–356. ACM, January 2016.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 913–930. ACM Press, October 2018.

[BGP00] Mihir Bellare, Oded Goldreich, and Erez Petrank. Uniform generation of np-witnesses using an np-oracle. *Information and Computation*, 163(2):510–526, 2000.

[BM09] Boaz Barak and Mohammad Mahmoody-Ghidary. Merkle puzzles are optimal - an $O(n^2)$-query attack on any key exchange from a random oracle. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 374–390. Springer, Heidelberg, August 2009.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.

[BN00] Dan Boneh and Moni Naor. Timed commitments. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 236–254. Springer, Heidelberg, August 2000.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

[Bri19] Karl Bringmann. Fine-grained complexity theory. In *36th International Symposium on Theoretical Aspects of Computer Science*, page 1, 2019.

[BRSV17] Marshall Ball, Alon Rosen, Manuel Sabin, and Prashant Nalini Vasudevan. Average-case fine-grained hardness. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *49th ACM STOC*, pages 483–496. ACM Press, June 2017.

[BS07] Daniel J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit chinese remainder theorem. *Mathematics of Computation*, 76(257):443–454, 2007.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CG18] Matteo Campanelli and Rosario Gennaro. Fine-grained secure computation. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018, Part II*, volume 11240 of *LNCS*, pages 66–97. Springer, Heidelberg, November 2018.

[CHI+20] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, abhi shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. Cryptology ePrint Archive, Report 2020/374, 2020. https://eprint.iacr.org/2020/374.

[CP18a] Bram Cohen and Krzysztof Pietrzak. The chia network blockchain. *Greenpaper*, 2018.

[CP18b] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 451–467. Springer, Heidelberg, April / May 2018.

[DGMV19] Nico Döttling, Sanjam Garg, Giulio Malavolta, and Prashant Nalini Vasudevan. Tight verifiable delay functions. Cryptology ePrint Archive, Report 2019/659, 2019. https://eprint.iacr.org/2019/659.

[DGMV20] Nico Döttling, Sanjam Garg, Giulio Malavolta, and Prashant Nalini Vasudevan. Tight verifiable delay functions. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 65–84. Springer, Heidelberg, September 2020.

[DMPS19] Luca De Feo, Simon Masson, Christophe Petit, and Antonio Sanso. Verifiable delay functions from supersingular isogenies and pairings. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 248–277. Springer, Heidelberg, December 2019.

[DNW05] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 37–54. Springer, Heidelberg, August 2005.

[DVV16] Akshay Degwekar, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Fine-grained cryptography. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 533–562. Springer, Heidelberg, August 2016.

[eth] https://coinmarketcap.com/currencies/ethereum/.

[EWT21] Shohei Egashira, Yuyu Wang, and Keisuke Tanaka. Fine-grained cryptography revisited. *Journal of Cryptology*, 34(3):23, July 2021.

[Fis18] Ben Fisch. Poreps: Proofs of space on useful data. *Cryptology ePrint Archive*, 2018.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

[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 ACM STOC*, pages 467–476. ACM Press, June 2013.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.

[GKW+16] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 3–16. ACM Press, October 2016.

[GKW17] Rishab Goyal, Venkata Koppula, and Brent Waters. Lockable obfuscation. In Chris Umans, editor, *58th FOCS*, pages 612–621. IEEE Computer Society Press, October 2017.

[GMM17] Sanjam Garg, Mohammad Mahmoody, and Ameer Mohammed. Lower bounds on obfuscation from all-or-nothing encryption primitives. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 661–695. Springer, Heidelberg, August 2017.

[GMPY06] Juan A. Garay, Philip D. MacKenzie, Manoj Prabhakaran, and Ke Yang. Resource fairness and composability of cryptographic protocols. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 404–428. Springer, Heidelberg, March 2006.

[GP21] Romain Gay and Rafael Pass. Indistinguishability obfuscation from circular security. pages 736–749. ACM Press, 2021.

[HR04] Chun-Yuan Hsiao and Leonid Reyzin. Finding collisions on a public road, or do secure hash functions need secret coins? In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 92–105. Springer, Heidelberg, August 2004.

[ILL89] Russell Impagliazzo, Leonid A. Levin, and Michael Luby. Pseudo-random generation from one-way functions (extended abstracts). In *21st ACM STOC*, pages 12–24. ACM Press, May 1989.

[Imp95] Russell Impagliazzo. A personal view of average-case complexity. In *Structure in Complexity Theory Conference*, pages 134–147, 1995.

[IR89] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *21st ACM STOC*, pages 44–61. ACM Press, May 1989.

[JLS21] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. pages 60–73. ACM Press, 2021.

[JMRR21] Samuel Jaques, Hart Montgomery, Razvan Rosie, and Arnab Roy. Time-release cryptography from minimal circuit assumptions. In *International Conference on Cryptology in India*, pages 584–606. Springer, 2021.

[KMT22] Dmitry Khovratovich, Mary Maller, and Pratyush Ranjan Tiwari. Minroot: Candidate sequential function for ethereum vdf. Cryptology ePrint Archive, Paper 2022/1626, 2022. https://eprint.iacr.org/2022/1626.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *CRYPTO 2022, Part IV*, LNCS, pages 359–388. Springer, Heidelberg, August 2022.

[Lin20] Yehuda Lindell. Secure multiparty computation (mpc). Cryptology ePrint Archive, Paper 2020/300, 2020. https://eprint.iacr.org/2020/300.

[LLW19] Rio LaVigne, Andrea Lincoln, and Virginia Vassilevska Williams. Public-key cryptography in the fine-grained setting. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 605–635. Springer, Heidelberg, August 2019.

[LW17] Arjen K Lenstra and Benjamin Wesolowski. Trustworthy public randomness with sloth, unicorn, and trx. *International Journal of Applied Cryptography*, 3(4):330–343, 2017.

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Publicly verifiable proofs of sequential work. In Robert D. Kleinberg, editor, *ITCS 2013*, pages 373–388. ACM, January 2013.

[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*, volume 168 of *LIPIcs*, pages 83:1–83:17. Schloss Dagstuhl, July 2020.

[MT19] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. Homomorphic time-lock puzzles and applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 620–649. Springer, Heidelberg, August 2019.

[Pap07] Christos H. Papadimitriou. *Computational complexity*. Academic Internet Publ., 2007.

[Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, January 2019.

[Pin73] Mark S Pinsker. On the complexity of a concentrator. In *7th International Telegraffic Conference*, volume 4, pages 1–318. Citeseer, 1973.

[PTC76] Wolfgang J Paul, Robert Endre Tarjan, and James R Celoni. Space bounds for a game on graphs. *Mathematical systems theory*, 10(1):239–251, 1976.

[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, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 481–509. Springer, Heidelberg, August 2020.

[RSS20] Lior Rotem, Gil Segev, and Ido Shahaf. Generic-group delay functions require hidden-order groups. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 155–180. Springer, Heidelberg, May 2020.

[RSW96] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[Sha19] Barak Shani. A note on isogeny-based hybrid verifiable delay functions. Cryptology ePrint Archive, Report 2019/205, 2019. https://eprint.iacr.org/2019/205.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.

[Val75] Leslie G Valiant. On non-linear lower bounds in computational complexity. In *Proceedings of the seventh annual ACM symposium on Theory of computing*, pages 45–53, 1975.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 379–407. Springer, Heidelberg, May 2019.

[WP22] Yuyu Wang and Jiaxin Pan. Non-interactive zero-knowledge proofs with fine-grained security. In *EUROCRYPT 2022, Part II*, LNCS, pages 305–335. Springer, Heidelberg, June 2022.

[WPC21] Yuyu Wang, Jiaxin Pan, and Yu Chen. Fine-grained secure attribute-based encryption. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 179–207, Virtual Event, August 2021. Springer, Heidelberg.

[WW10] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51st FOCS*, pages 645–654. IEEE Computer Society Press, October 2010.

[WW20] Hoeteck Wee and Daniel Wichs. Candidate obfuscation via oblivious LWE sampling. Cryptology ePrint Archive, Report 2020/1042, 2020. https://eprint.iacr.org/2020/1042.

[WZ17] Daniel Wichs and Giorgos Zirdelis. Obfuscating compute-and-compare programs under LWE. In Chris Umans, editor, *58th FOCS*, pages 600–611. IEEE Computer Society Press, October 2017.

[Yak18] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

[Zha22] Mark Zhandry. To label, or not to label (in generic groups). In *CRYPTO 2022, Part III*, LNCS, pages 66–96. Springer, Heidelberg, August 2022.