Decomposable Obfuscation: A Framework for Building Applications of Obfuscation From Polynomial Hardness

Qipeng Liu Mark Zhandry

Princeton University
{qipengl, mzhandry}@princeton.edu

Abstract

There is some evidence that indistinguishability obfuscation (iO) requires either exponentially many assumptions or (sub)exponentially hard assumptions, and indeed, all known ways of building obfuscation suffer one of these two limitations. As such, any application built from iO suffers from these limitations as well. However, for most applications, such limitations do not appear to be inherent to the application, just the approach using iO. Indeed, several recent works have shown how to base applications of iO instead on functional encryption (FE), which can in turn be based on the polynomial hardness of just a few assumptions. However, these constructions are quite complicated and recycle a lot of similar techniques.

In this work, we unify the results of previous works in the form of a weakened notion of obfuscation, called *Decomposable Obfuscation*. We show (1) how to build decomposable obfuscation from functional encryption, and (2) how to build a variety of applications from decomposable obfuscation, including all of the applications already known from FE. The construction in (1) hides most of the difficult techniques in the prior work, whereas the constructions in (2) are much closer to the comparatively simple constructions from iO. As such, decomposable obfuscation represents a convenient new platform for obtaining more applications from polynomial hardness.

1 Introduction

Program obfuscation has recently emerged as a powerful cryptographic concept. An obfuscator is a compiler for programs, taking an input program, and scrambling it into an equivalent output program, but with all internal implementation details obscured. Indistinguishability obfuscation (iO) is the generally-accepted notion of security for an obfuscator, which says that the obfuscations of equivalent programs are computationally indistinguishable.

In the last few years since the first candidate indistinguishability obfuscator of Garg, Gentry, Halevi, Raykova, Sahai, and Waters [GGH+13b], obfuscation has been used to solve many new amazing tasks such as deniable encryption [SW14], multiparty non-interactive

key agreement [BZ14], polynomially-many hardcore bits for any one-way function [BST14], and much more. Obfuscation has also been shown to imply most traditional cryptographic primitives[1] such as public key encryption [SW14], zero knowledge [BP15], trapdoor permutations [BPW16], and even fully homomorphic encryption [CLTV15]. This makes obfuscation a "central hub" in cryptography, capable of solving almost any cryptographic task, be it classical or cutting edge. Even more, obfuscation has been shown to have important connections to other areas of computer science theory, from demonstrating the hardness of finding Nash equilibrium [BPR15] to the hardness of certain tasks in differential privacy [BZ14, BZ16].

The power of obfuscation in part comes from the power of the underlying tools, but its power also lies in the *abstraction*, by hiding away the complicated implementation details underneath a relatively easy to use interface. In this work, we aim to build a similarly powerful abstraction that avoids some of the limitations of iO.

1.1 The Sub-exponential Barrier In Obfuscation

Indistinguishability obfuscation (iO), as an assumption, has different flavor than most assumptions in cryptography. Most cryptographic assumptions look like

> "Distribution A is computationally indistinguishable from distribution B," or
> "Given a sample a from distribution A, it is computationally infeasible
> to compute a value b such that a, b satisfy some given relation."

Such assumptions are often referred to as falsifiable [Nao03], or more generally as complexity assumptions [GT16]. In contrast, iO has the form

> "*For every pair of circuits C_0, C_1 that are functionally equivalent,*
> $\mathsf{iO}(C_0)$ is computationally indisitnguishable from $\mathsf{iO}(C_1)$."

In other words, for each pair of equivalent circuits C_0, C_1, there is an instance of a complexity assumption: that $\mathsf{iO}(C_0)$ is indisintugishable from $\mathsf{iO}(C_1)$. iO then is really a *collection* of exponentially-many assumptions made simultaneously, one per pair of equivalent circuits. iO is violated if *a single* assumption in the collection is false. This is a serious issue, as the security of many obfuscators relies on new assumptions that essentially match the schemes. To gain confidence in the security of the schemes, it would seem like we need to investigate the iO assumption for every possible pair of circuits, which is clearly infeasible.

Progress has been made toward remedying this issue. Indeed, Gentry, Lewko, Sahai, and Waters [GLSW15] show how to build obfuscation from a single assumption — multilinear subgroup elimination — on multilinear maps. Unfortunately, the security reduction loses a factor exponential in the number of input bits to the program. As such, in order for the reduction to be meaningful, the multilinear subgroup elimination problem must actually be *sub-exponentially* hard. Similarly, Bitansky and Vaikuntanathan [BV15] and Ananth and Jain [AJ15] demonstrate how to construct iO from a tool called functional encryption (FE). In turn, functional encryption can be based on simple assumptions on multilinear

[1] with additional mild assumptions such as the existence of one-way functions

maps [GGHZ16]. However, while the construction of functional encryption can be based on the polynomial hardness of just a couple multilinear map assumptions, the construction of iO from FE incurs an exponential loss. This means the FE scheme, and hence the underlying assumptions on multilinear maps, still need to be *sub-exponentially* secure.

All current techniques for building iO suffer one of these two limitations: either security is based on an exponential number of assumptions, or the reduction incurs an exponential loss. Unfortunately, this means every application of iO also suffers from the same limitations. As iO is the only known instantiation of many new cryptographic applications, an important research direction is to devise new instantiations that avoid this exponential loss.

1.2 Breaking the Sub-exponential Barrier

A recent line of works starting with Garg, Pandey, Srinivasan [GPS16] and continued by [GPSZ16, GS16] have shown how to break the sub-exponential barrier for certain applications. Specifically, these works show how to base certain applications on functional encryption, where the loss of the reduction is just polynomial. Using [GGHZ16], this results in basing the applications on the polynomial hardness of a few multilinear map assumptions. The idea behind these works is to compose the FE-to-iO conversion of [BV15, AJ15] with the iO-to-Application conversion to get an FE-to-Application construction. While this construction requires an exponential loss (due to the FE-to-iO conversion), by specializing the conversion to the particular application and tweaking things appropriately, the reduction can be accomplished with a polynomial loss. Applications treated in this way include: the hardness of computing Nash equilibria, trapdoor permutations, universal samplers, multiparty non-interactive key exchange, and multi-key functional encryption[2].

While the above works represent important progress, the downside is that, in order to break the sub-exponential barrier, they also break the convenient obfuscation abstraction. Both the FE-to-iO and iO-to-Application conversions are non-trivial, and the FE-to-iO conversion is moreover non-black box. Add to that the extra modifications to make the combined FE-to-Application conversion be polynomial, and the resulting constructions and analyses become reasonably cumbersome. This makes translating the techniques to new applications rather tedious — not to mention potentially repetitive given the common FE-to-iO core — and understanding the limits of this approach almost impossible.

1.3 A New Abstraction: Decomposable Obfuscation

In this work, we define a new notion of obfuscation, called *Decomposable Obfuscation*, or dO, that addresses the limitations above. This notion abstracts away many of the common techniques in [GPS16, GPSZ16, GS16]; we use those techniques to build dO from the *polynomial* hardness of functional encryption. Then we can show that dO can be used to build the various applications. With our new notion in hand, the dO-to-Application

[2]The kind of functional encryption that is used as a starting point only allows for a single secret key query

constructions begin looking much more like the original iO-to-Application constructions, with easily identifiable modifications that are necessary to prove security using our weaker notion.

1.3.1 The Idea

Functional Encryption (FE). As in the works of [GPS16, GPSZ16, GS16], we will focus on obtaining our results from the starting point of polynomially-secure functional encryption. Functional encryption is similar to regular public key encryption, except now the secret key holder can produce function keys corresponding to arbitrary functions. Given a function key for a function f and a ciphertext encrypting m, one can learn $f(m)$. Security requires that even given the function key for f, encryptions of m_0 and m_1 are indistinguishable, so long as $f(m_0) = f(m_1)$[3].

The FE-to-iO Conversion. The FE-to-iO conversions of [BV15, AJ15] can be thought of very roughly as follows. To obfuscate a circuit C, we generate the keys for an FE scheme, and encrypt the description of C under the FE scheme's public key, obtaining c. We also produce function keys fk_i for particular functions f_i that we will describe next. The obfuscated program consists of c and the fk_i.

To evaluate the program on input x, we first use fk_1 and c to learn $f_1(C)$. $f_1(C)$ is defined to produce two ciphertexts c_0, c_1, encrypting $(C, 0)$ and $(C, 1)$, respectively. We keep c_{x_1}, discarding the other ciphertext. Now, we actually define fk_1 to encrypt $(C, 0)$ and $(C, 1)$ using the functional encryption scheme itself — therefore, we can continue applying function keys to the resulting plaintexts. We use fk_2 and c_{x_1} to learn $f_2(C, x_1)$. $f_2(C, b)$ is defined to produce two ciphertexts c_{b0}, c_{b1}, encrypting $(C, b0)$ and $(C, b1)$. Again, these ciphertexts will be encrypted using the functional encryption scheme. We will repeat this process until we obtain the encryption c_x of (C, x). Finally, we apply the last function key for the function f_{n+1}, which is the universal circuit evaluating $C(x)$.

This procedure implicitly defines a complete binary tree of all strings of length at most 2^n, where a string x is the parent of the two string $x||0$ and $x||1$. At each node $y \in \{0,1\}^{\leq n}$, consider running the evaluation above for the first $|y|$ steps, obtaining a ciphertext c_y encrypting (C, y). We then assign the circuit C to the node y, according to the circuit that is encrypted in c_y. The root is *explicitly* assigned C by handing out the ciphertext c since we explicitly encrypt C to obtain c. All subsequent nodes are *implicitly* assigned C as c_y is derived from c during evaluation time. Put another way, by explicitly assigning a circuit C to a node (in this case, the root) we implicitly assign the same circuit C to all of its descendants. The exception is the leaves: if we were to assign a circuit C to a leaf x, we instead assign the output $C(x)$. In this way, the leaves contain the truth table for C.

Now, we start from an obfuscation of C_0 (assigning C_0 to the root of the tree) and we wish to change the obfuscation to an obfuscation of C_1 (assigning C_1 to the root). We cannot do this directly, but the functional encryption scheme does allow us to do the following: un-assign

[3]The two encryptions would clearly be distinguishable if $f(m_0) \neq f(m_1)$ just by decrypting using the secret function key. Thus, this is the best one can hope for with an indistinguishability-type definition

a circuit C from any internal node y[4], and instead *explicitly* assign C to the two children of that node. This is accomplished by changing c_y to encrypt $(\perp, x)$, explicitly constructing the ciphertexts $c_{y||0}$ and $c_{y||1}$, and embedding $c_{y||0}, c_{y||1}$ in the function key $\mathsf{fk}_{|y|}$ in a particular way. If the children are leaves, explicitly assign the outputs of C on those leaves. Note that this process does not change the values assigned to the leaves; as such, the functionality of the tree remains unchanged, so this change cannot be detected by functionality alone. The security of functional encryption shows that, in fact, the change is undetectable to any polynomial-time adversary.

The security reduction works by performing a depth-first traversal of the binary tree. When processing a node y on the way down the tree, we un-assign C_0 from y and instead explicitly assign C_0 to the children of y. When we get to a leaf, notice that by functional equivalence, we actually simultaneously have the output of C_0 and C_1 assigned. Therefore, when processing a node y on our way up the tree from the leaves, we can perform the above process in reverse but for C_1 instead of C_0. We can un-assign C_1 from the children of y, and then explicitly assign C_1 to y. In this way, when the search is complete, we explicitly assign C_1 to the root, which implicitly assigns C_1 to all nodes in the tree. At this point, we are obfuscating C_1. By performing a depth-first search, we ensure that the number of explicitly assigned nodes never exceeds $n + 1$, which is crucial for the efficiency of the obfuscator, as we pay for explicit assignments (since they correspond to explicit ciphertexts embedded in the function keys) but not implicit ones (since they are computed on the fly). Note that while the obfuscator itself is polynomial, the number of steps in the proof is exponential: we need to un-assign and re-assign every internal node in the tree, which are exponential in number. This is the source of the exponential loss.

Shortcutting the conversion process. The key insight in the works of [GPS16, GPSZ16, GS16] is to modify the constructions in a way so that it is possible to re-assign certain internal nodes in a single step, without having to re-assign all of its descendants first. By doing this it is possible to shortcut our way across an exponential number of steps using just a few steps.

In these prior works, the process is different for each application. In this work, we generalize the conditions needed for and the process of shortcutting in a very natural way. To see how shortcutting might work, we introduce a slightly different version of the above assignment setting. Like before, every node can be assigned a circuit. However, now the circuit assigned to a node u of length k must work on inputs of length $n - k$; essentially, it is the circuit that is "left over" after reading the first k bits and which operates on the remaining $n - k$ bits.

If we explicitly assign a circuit C_y to a node y, its children are implicitly assigned the *partial evaluations* of C_y on 0 and 1. That is, the circuit $C_{y||b}$ assigned to $y||b$ is $C_y(b, \cdot)$. We will actually use $C_y(b, \cdot)$ to denote the circuit obtained by hard-coding b as the first input bit, and then simplifying using simple rules: (1) any unary gate with a constant input wire is deleted and replaced with an appropriate constant input wire, (2) any binary gate with a constant input is replaced with just a unary gate (a passthrough or a NOT) or a hardwired

[4]By assigning $\perp$ instead, which does *not* propagate down the tree

output according to the usual rules, (3) any gate whose output wire is unused is deleted, and (4) this process is repeated until there are no gates with hardwired inputs and no unused gates. An important observation is that our rules guarantee that circuits assigned to leaves are always constants, corresponding to the output of the circuit at that point.

Now when we obfuscate by assigning C to the root, the internal nodes are implicitly assigned the simplified partial evaluations of C on the prefix corresponding to that node: node y is assigned $C(y, \cdot)$ (simplified). The move we are allowed to make is now to un-assign C from a node where C was explicit, and instead explicitly assign the simplified circuits $C(0, \cdot)$ and $C(1, \cdot)$ to its children. We call the partial evaluations $C(0, \cdot)$ and $C(1, \cdot)$ *fragments* of C, and we call this process of un-assigning the parent and assigning the fragments to the children *decomposing* the node to its children fragments. The reverse of decomposing is *merging*.

This simple transformation to the binary tree rules allows for, in some instances, the necessary shortcutting to avoid an exponential loss. When transforming C_0 to C_1, the crucial observation is that if any fragment $C_0(x, \cdot)$ is equal to $C_1(x, \cdot)$ *as circuits* (after simplification), it suffices to stop when we explicitly assign a circuit to x; we do not need to continue all the way down to the leaves. Indeed, once we explicitly assign the fragment $C_0(y, \cdot)$ to a node y, y already happens to be assigned the fragment $C_1(y, \cdot)$ as well, and all of its descendants are therefore implicitly assigned the corresponding partial evaluations of C_1 as well. By not traversing all the way to the leaves, we cut out potentially exponentially many steps. For certain circuit pairs, it may therefore be possible to transform C_0 to C_1 in only polynomially-many steps.

Our New Obfuscation Notion. Our new obfuscation notion stems naturally from the above discussion. Consider two circuits C_0, C_1 of the same size, and consider assigning C_0 to the root of the binary tree. Suppose there is a set S of tree nodes of size τ that (1) exactly cover all of the leaves[5], and (2) for every nodes $x \in S$, the (simplified) fragments $C_0(y, \cdot)$ and $C_1(y, \cdot)$ are identical *as circuits*. Then we say the circuits C_0, C_1 are τ-decomposing equivalent. Our new obfuscation notion, called *decomposable obfuscation*, is parameterized by τ and says, roughly, that the obfuscations of two τ-decomposing equivalent circuits must be indistinguishable.

1.4 Our Results

Our results are as follows:

- We show how to use (compact, single key) functional encryption to attain our notion of dO. The construction is similar to the FE-to-iO conversion, with the key difference that each step simplifies the circuit as must as possible; this implements the new tree rules we need for shortcutting.

[5]In the sense that for each leaf, the path from root to leaf contains exactly one element in S

The number of steps in the process of converting C_0 to C_1, and hence the loss in the security reduction is proportional to τ. However, we show that by performing the decompose/merge steps in the right order, we can make sure the number of explicitly assigned nodes is always at most $n+1$, independent of τ. This means the obfuscator itself does not depend on τ, and therefore τ can be taken to be an arbitrary polynomial or even exponential and the obfuscator will still be efficient. If we restrict τ to a polynomial, we obtain dO from polynomially secure FE. Our results also naturally generalize to larger τ: we obtain dO for quasipolynomial τ from quasipolynomially secure FE, and we obtain dO for exponential τ from (sub)exponentially secure FE.

- We note that by setting τ to be 2^n, τ-decomposing equivalence corresponds to standard functional equivalence, since we can take the set S of nodes to consist of all leaf nodes. Then dO coincides with the usual notion of indistinguishability obfuscation, giving us iO from sub-exponential FE. This re-derives the results of [BV15, AJ15]. In our reduction, the loss is $O(2^n)$.

- We then show how to obtain several applications of obfuscation from dO with *polynomial* τ. Thus, for all these applications, we obtain the application from the polynomial hardness of FE, re-deriving several known results. In these applications, there is a single input, or perhaps several inputs, for which the computation must be changed from using the original circuit to using a hard-coded value. This is easily captured by decomposing equivalence: by decomposing each node from the root to the leaf for a particular input x, the result is that that the program's output on x is hard-coded into the obfuscation. Applications include:

  - Proving the hardness of finding Nash equilibria (Section 5.5; Nash hardness from FE was originally shown in [GPS16])
  - Trapdoor Permutations (Section 5.6; originally shown in [GPSZ16])
  - Universal Samplers (Section 5.3; originally shown in [GPSZ16])
  - Short Signatures (Section 5.2; not previously known from functional encryption, though known from obfuscation [SW14])
  - Multi-key functional encryption (Section 5.4; originally shown in [GS16])

  We note that Nash, universal samplers, and short signatures only require (polynomially hard) dO and one-way functions. In contrast, trapdoor permutations and multi-key functional encryption both additionally require public key encryption. If basing the application on public key functional encryption, this assumption is redundant. However, unlike the the case for full-fledged iO, we do not know how to obtain public key functional encryption from just polynomially hard dO and one-way functions (more on this below). We do show that a weaker multi-key *secret key* functional encryption scheme does follow from dO and one-way functions.

Thus, we unify the techniques underlying many of the applications of FE — namely iO, Nash, trapdoor permutations, universal samplers, short signatures, and multi-key FE —

under a single concept, dO. The constructions and proofs starting from dO are much simpler than the original proofs using functional encryption, due to the convenient dO abstraction hiding many of the common details. We hope that dO will also serve as a starting point for further constructions based on polynomially-hard assumptions.

1.5 Discussion

A natural question to ask is: what are the limits of these techniques? Could they be used to give full iO from polynomially-hard assumptions? Or at least all known applications from polynomial hardness? Here, we discuss several difficulties that arise.

Difficulties in breaking the sub-exponential barrier. First, exponential loss may be inherent to constructing iO. Indeed, the following informal argument is adapted from Garg et al. [GGSW13]. Suppose we can prove iO from a single fixed assumption. This means that for every pair of equivalent circuits C_0, C_1, we prove under this assumption that $\mathsf{iO}(C_0)$ is indistinguishable from $\mathsf{iO}(C_1)$. Fix two circuits C_0, C_1, and consider the proof for those circuits. If C_0 is equivalent to C_1, then the proof succeeds. However, if C_0 is *not* equivalent to C_1, then the proof *must* fail: let x be a point such that $C_0(x) \neq C_1(x)$. Then a simple adversary with x hard-coded can distinguish $\mathsf{iO}(C_0)$ from $\mathsf{iO}(C_1)$ simply by running the obfuscated program on x.

This intuitively means that the proof must some how decide whether C_0 and C_1 are equivalent. Since the proof consists of an *efficient* algorithm R reducing breaking the assumption to distinguishing $\mathsf{iO}(C_0)$ from $\mathsf{iO}(C_1)$, it seems that R must be efficiently deciding circuit equivalence. Assuming $P \neq NP$, such a reduction should not exist.[6]

The reductions from iO to functional encryption/simple multilinear map assumptions avoid this argument by not being efficient. Indeed, the reductions traverse the entire tree of 2^n nodes as described above. In essence, the proof in each step just needs to check a local condition such as $C_0(x) = C_1(x)$ for some particular x — which can be done efficiently — as opposed to checking equivalence for all inputs.

While this argument is far from a proof of impossibility, it does represent an significant inherent difficulty in building full-fledged iO from polynomial hardness. We believe that overcoming this barrier, or showing that it is insurmountable, is an important and fascinating open question. For example, imagine translating the arguments above to iO for computational models with unbounded input lengths such as Turing machines. In this case, equivalence is not only inefficient, but *undecidable*. As such, the above arguments demonstrate a barrier to basing Turing machine obfuscation on a finite number of even (sub)exponentially hard

[6] One may wonder whether the same arguments apply to the seemingly similar setting of zero knowledge, where zero knowledge must hold for true instances, but soundness must hold for false instances. The crucial difference is that soundness does not prevent the zero knowledge simulator from working on false instances. Therefore, a reduction from a hard problem to zero knowledge does not need to determine whether the instance is in the language. In contrast, for iO, the security property must apply to equivalent circuits, but correctness implies that it *cannot* apply to inequivalent circuits.

assumptions. An important open question is whether it is possible to build iO from Turing machines from iO for circuits; we believe achieving this goal will likely require techniques that can also be used to overcome the sub-exponential barrier.

For the remainder of the discussion, we will assume that building iO from polynomial hardness is beyond reach without significant breakthroughs.

Avoiding the Barrier. We observe that poly-decomposing equivalence is an NP relation: the polynomial-sized set of nodes where the fragments are identical provides a witness that two circuits are equivalent: it is straightforward to check that a collection of nodes covers all of the leaves and that the fragments at those nodes are identical. In contrast, general circuit equivalence is *co-*NP-complete, and therefore unlikely to be in NP unless the polynomial hierarchy collapses. This distinction is exactly what allows us to avoid the sub-exponential barrier.

Our security reduction has access to the witness for equivalence, which guides how the reduction operates. The reduction can use the witness to trivially verify that the two circuits are equivalent; if the witness is not supplied or is invalid, the reduction does not run. The sub-exponential barrier therefore no longer applies in this setting.

More generally, the sub-exponential barrier will not apply to circuit pairs for which there is a witness proving equivalence; in other words, languages of circuit pairs in $NP \cap co\text{-}NP$ [7]. Any languages outside $NP \cap co\text{-}NP$ are likely to run into the same sub-exponential barrier as full iO since witnesses for equivalence do not exist, and meanwhile there remains some hope that languages inside might be obfuscatable without a sub-exponential loss by feeding the witness to the reduction.

In fact, almost all applications of obfuscation we are aware of can be modified so that the pairs of circuits in question have a witness proving equivalence. For example, consider obtaining public key encryption from one-way functions using obfuscation [SW14]. The secret key is the seed s for a PRG, and the public key is the corresponding output x. A ciphertext encrypting message m is an obfuscation of the program $P_{x,m}$, which takes as input a seed s' and checks that $\mathsf{PRG}(s') = x$. If the check fails, it aborts and outputs 0. Otherwise if the check passes, it outputs m. To decrypt using s, simply evaluate the obfuscated program on s.

In the security proof, iO is used for the following two programs: $P_{x,m}$ where x *is a truly random element in the co-domain of* PRG, and Z, the trivial program that always outputs 0. We note that since PRG is expanding, with high probability x will not have a pre-image, and therefore $P_{x,m}$ will also output 0 everywhere. Therefore, $P_{x,m}$ and Z are (with high probability) functionally equivalent.

For general PRGs, there is no witness for equivalence of these two programs. However, by choosing the right PRG, we can remedy this. Let P be a one-way permutation, and let h be a hardcore bit for P. Now let $\mathsf{PRG}(s) = (P(s), h(s))$. Instead of choosing x randomly, we choose x as $P(s), 1 \oplus h(s)$ for a random seed s[8]. This guarantees that x has no pre-image

[7] Circuit equivalence is trivially in *co-*NP; a point on which the two circuits differ is a witness that they are not equivalent

[8] This is no longer a random element in the codomain of the PRG, but it suffices for the security proof

under PRG. Moreover, s serves as a witness that x has no pre-image. Therefore, the programs $P_{x,m}$ and Z have a witness for equivalence.

Limits of the dO approach. Unfortunately, decomposable obfuscation is not strong enough to prove security in many settings. In fact, we demonstrate (Section 4) that τ-decomposing equivalence can be decided in time proportional to τ, meaning poly-decomposing equivalence is actually in P. However, for example, the equivalence of programs $P_{x,m}$ and Z above cannot possibly be in P — otherwise we could break the PRG: on input x, check if $P_{x,m}$ is equivalent to Z. A random output will yield equivalence with probability $1/2$, whereas a PRG sample will never yield equivalence circuits. In other words, $P_{x,m}$ and Z are provably *not* poly-decomposing equivalent, despite being functionally equivalent programs.

One can also imagine generalizing dO to encompass more general paths through the binary tree of prefixes. For example, one could decompose the circuit into fragments, partially merge some of the fragments back together, decompose again, etc. We show that this seemingly more general *path* decomposing equivalence is in fact equivalent to (standard) decomposing equivalence. Therefore, this path dO is equivalent to (standard) dO, and only works for pairs of circuits that can be easily verified as equivalent.

Unsurprisingly then, all the applications we obtain using poly-decomposable obfuscation obfuscate circuits for which it is easy to verify equivalence. This presents some interesting limitations relative to iO:

- All known ways of getting public key encryption from iO and one-way functions suffer from a similar problem as the example above, and cannot to our knowledge be based on poly-dO. In other words, unlike iO, dO might not serve as a bridge between Minicrypt and Cryptomania. Some of our applications — namely multi-key functional encryption and trapdoor permutations — imply public key encryption; for these applications, we actually have to use public key encryption as an additional ingredient. Note that if we are instantiating dO from functional encryption, we get public key encryption for free. However, if we are interested in placing dO itself in the complexity landscape, the apparent inability to give public key encryption is an interesting barrier.

  More generally, a fascinating question is whether any notion of obfuscation that works only for efficiently-recognizable equivalent circuits can imply public key encryption, assuming additionally just one-way functions.

- While iO itself does not imply one-way functions[9], iO can be used in conjunction with a worst-case complexity assumption, roughly $NP \not\subseteq BPP$, to obtain one-way functions [KMN+14]. The proof works by using a hypothetical inverter to solve the circuit equivalence problem; assuming the circuit equivalence problem is hard, they reach a contradiction. The solver works exactly because iO holds for the equivalent circuits.

[9] If $P = NP$, one-way functions do not exist but circuit minimization can be used to obfuscate

This strategy simply does not work in the context of dO. Indeed, dO only applies to circuits for which equivalence is easily decidable anyway, meaning no contradiction is reached. In order to obtain any results analogous to [KMN+14] for restricted obfuscation notions, the notion must always work for at least some collection of circuit pairs for which circuit equivalence is hard to decide. Put another way, dO could potentially exist in Pessiland.

- More generally, dO appears to roughly capture the most general form of the techniques in [GPS16, GPSZ16, GS16], and therefore it appears that these techniques will not extend to the case of non-efficiently checkable equivalence. Many constructions using obfuscation fall in this category of non-checkable equivalence: deniable encryption and non-interactive zero knowledge [SW14], secure function evaluation with optimal communication complexity [HW15], adaptively secure universal samples [HJK+16], and more.

We therefore leave some interesting open questions:

- Build iO for a class of circuit pairs for which equivalence is not checkable in polynomial time, but for which security can be based on the polynomial hardness of just a few assumptions.
- Modify the constructions in deniable encryption/NIZK/function evaluation/etc so that obfuscation is only ever applied on program pairs for which equivalence can be easily verified — ideally, the circuits would be decomposing equivalent.
- Prove that for some applications, obfuscation *must* be applied to circuit pairs with non-efficiently checkable equivalence.

2 Preliminaries: Definitions and Notations

In this paper, we use κ to denote the security parameter. We denote a polynomial by $\mathsf{poly}(\cdot)$. We say a function $f(\cdot) : \mathbb{N} \to \mathbb{R}^+$ is negligible if for all constant $c > 0$, $f(n) < \frac{1}{n^c}$ for all large enough n. Wel use $\mathrm{negl}(\cdot)$ to denote a negligible function. When we refer to a probabilistic algorithm $\mathcal{A}$, sometimes we need to specify the random string r feed to $\mathcal{A}$ on input x as $\mathcal{A}(x; r)$. For a finite set S, we use $x \xleftarrow{R} S$ to denote uniform sampling of x from the set S. If we ignore r for a probabilistic algorithm $\mathcal{A}$, then the randomness is drawn uniformly at random; i.e. $\mathcal{A}(x)$ denotes $\mathcal{A}(x; r)$ where r is a uniformly random string. We use $\mathcal{A}$ to denote a sequence of non-uniform adversaries $\{\mathcal{A}_\kappa\}$ and we say $\mathcal{A}$ is a poly sized adversary if for every κ, $\mathcal{A}_\kappa$ is of size at most $\mathsf{poly}(\kappa)$. We denote $[\kappa]$ as the set $\{1, 2, \cdots, k\}$. A binary string x is represented as $x_1 x_2 \cdots x_\ell$ where ℓ is the length of that binary string. For a binary string x, $x_{[i]}$ denotes the i-bit prefix of x which is $x_1 x_2 \cdots x_i$ and $x_{[i \cdots j]}$ denotes the substring $x_i x_{i+1} \cdots x_j$. For two strings x, y, $x||y$ is the concatenation of x and y.

One-Way Functions

A one-way function is a function that is easy to compute but hard to invert. Here is the formal definition of a one-way function:

Definition 2.1. *A one-way function* $f : \{0,1\}^* \to \{0,1\}^*$ *is a deterministic algorithm satisfying the following properties:*

- ***Efficiently Computable :*** *For any* $\kappa \in \mathbb{Z}^+$*, any* $x \in \{0,1\}^\kappa$*,* $f(x)$ *is polynomial time computable;*
- ***Hard to Invert :*** *For any poly sized adversary* $\mathcal{A}$*, there exists a negligible function* negl *such that for any* $\kappa \in \mathbb{Z}^+$*,*

$$\Pr_{x \xleftarrow{R} \{0,1\}^\kappa}[f(\mathcal{A}(f(x))) = f(x)] < \mathsf{negl}(\kappa)$$

Pseudo Random Generators

A pseudo random generator is a deterministic algorithm that maps a short truly random string to a longer pseudo random string. Assuming the existence of one-way functions, there exists PRG. Here is the formal definition of PRG.

Definition 2.2. *A pseudo random generator* PRG *with a expansion factor* $\ell(\cdot)$ *is a polynomial deterministic algorithm such that*

- ***Length Expansion :*** *For any* $\kappa \in \mathbb{Z}^+$ *and* $x \in \{0,1\}^\kappa$*,* $|\mathsf{PRG}(x)| = \ell(\kappa)$ *where* $\ell(\cdot)$ *is a polynomial*
- ***Pseudo Randomness :*** *For any poly sized adversary* $\mathcal{A}$*, there exists a negligible function* negl *such that for any* κ*,*

$$\left| \Pr_{y \xleftarrow{R} \{0,1\}^{\ell(\kappa)}}[\mathcal{A}(y) = 1] - \Pr_{x \xleftarrow{R} \{0,1\}^\kappa}[\mathcal{A}(\mathsf{PRG}(x)) = 1] \right| \leq \mathsf{negl}(\kappa)$$

Symmetric Key Encryption

Now we define a symmetric key encryption scheme.

Definition 2.3. *A symmetric key encryption scheme* SKE *consists a tuple of algorithms* $\mathsf{SKE.KeyGen}$*,* $\mathsf{SKE.Enc}$*,* $\mathsf{SKE.Dec}$ *where*

- $\mathsf{SKE.KeyGen}(1^\kappa)$ *is a probabilistic polynomial time algorithm that takes a security parameter* κ*, outputs a secret key* sk*;*
- $\mathsf{SKE.Enc}(\mathsf{sk}, m)$ *is a polynomial time algorithm that takes a secret key* sk *and a message* $m \in \{0,1\}^*$*, outputs a ciphertext* c*;*

- $\mathsf{SKE.Dec}(\mathsf{sk}, c)$ *is a polynomial time algorithm that takes a secret key* sk *and a ciphertext* $c \in \{0,1\}^*$*, outputs a message* m'*;*
- ***Correctness :*** *a symmetric key encryption is said to be correct if for all* κ *and all message* $m \in \{0,1\}^*$*,*

$$\Pr\left[\mathsf{SKE.Dec}(\mathsf{sk}, c) = m \,|\, \mathsf{sk} \leftarrow \mathsf{SKE.KeyGen}(1^\kappa)\,;\, c \leftarrow \mathsf{SKE.Enc}(\mathsf{sk}, m)\right] = 1$$

Consider the following security game $\mathsf{Game}_{\kappa,\mathcal{A},b}$:

- The adversary $\mathcal{A}$ generates two messages m_0, m_1 ($|m_0| = |m_1|$) and sends them to the challenger;
- The challenger runs $\mathsf{SKE.KeyGen}(1^\kappa)$ to get a secret key sk, and sends the ciphertext $c = \mathsf{SKE.Enc}(\mathsf{sk}, m_b)$ to $\mathcal{A}$;
- $\mathcal{A}$ gets c and outputs b'; b' is the output of this game.

A symmetric key encryption scheme SKE is said to be secure if for every poly sized adversary $\mathcal{A}$, there exists a negligible function negl such that for every κ

$$|\Pr[\mathsf{Game}_{\kappa,\mathcal{A},0} = 1] - \Pr[\mathsf{Game}_{\kappa,\mathcal{A},1} = 1]| \leq \mathsf{negl}(\kappa)$$

With a pseudo random generator, one can construct a symmetric key encryption scheme from it.

Public Key Encryption

Now we give the definition of a public key encryption scheme.

Definition 2.4. *A public key encryption scheme* PKE *consists a tuple of algorithms* $\mathsf{PKE.KeyGen}$*,* $\mathsf{PKE.Enc}$*,* $\mathsf{PKE.Dec}$ *where*

- $\mathsf{PKE.KeyGen}(1^\kappa)$ *is a probabilistic polynomial time algorithm that takes a security parameter* κ*, outputs a key pair: a secret key* sk *and a public key* pk*;*
- $\mathsf{PKE.Enc}(\mathsf{pk}, m)$ *is a probabilistic polynomial time algorithm that takes a public key* pk *and a message* $m \in \{0,1\}^*$*, outputs a ciphertext* c*;*
- $\mathsf{PKE.Dec}(\mathsf{sk}, c)$ *is a polynomial time algorithm that takes a secret key* sk *and a ciphertext* $c \in \{0,1\}^*$*, outputs a message* m'*;*
- ***Correctness :*** *a public key encryption is said to be correct if for all* κ *and all message* $m \in \{0,1\}^*$*,*

$$\Pr\left[\mathsf{PKE.Dec}(\mathsf{sk}, c) = m \,|\, (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\kappa)\,;\, c \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, m)\right] = 1$$

To define the selective security of a public key encryption scheme, we need the following security game. Consider the security game $\mathsf{Game}_{\kappa,\mathcal{A},b}$:

- The adversary $\mathcal{A}$ generates two messages m_0, m_1 ($|m_0| = |m_1|$) and sends them to the challenger;
- The challenger runs $\mathsf{SKE.KeyGen}(1^\kappa)$ to get a key pair $(\mathsf{pk}, \mathsf{sk})$, and sends the ciphertext $c = \mathsf{SKE.Enc}(\mathsf{sk}, m_b)$ and pk to $\mathcal{A}$;
- $\mathcal{A}$ then outputs b'; b' is the output of this game.

A public key encryption scheme PKE is said to be secure if for every poly sized adversary $\mathcal{A}$, there exists a negligible function negl such that for every κ,

$$|\Pr[\mathsf{Game}_{\kappa,\mathcal{A},0} = 1] - \Pr[\mathsf{Game}_{\kappa,\mathcal{A},1} = 1]| \leq \mathsf{negl}(\kappa)$$

Indistinguishability Obfuscation

Here is the definition of indistinguishability obfuscators [BGI+01, GGH13a].

Definition 2.5. *A PPT algorithm* iO *is an indistinguishability obfuscator for a circuit family* $\mathcal{C}$ *if the following hold,*

- ***Preserving functionalities :*** *For all* κ *and all circuit* $C \in \mathcal{C}$*, for all input* x*, we have* $\Pr[\mathsf{iO}(1^\kappa, C)(x) = C(x)] = 1$*;*
- ***Indistinguishability :*** *For all* $C_0, C_1 \in \mathcal{C}$ *where* C_0 *and* C_1 *have the same functionalities (in other words, for all input* $x \in \{0,1\}^n$*,* $C_0(x) = C_1(x)$*) and* $|C_0| = |C_1|$*, for any poly sized adversary* $\mathcal{A}$*, there exists a negligible function* negl *such that for all* κ*,*

$$|\Pr[\mathcal{A}(\mathsf{iO}(1^\kappa, C_0)) = 1] - \Pr[\mathcal{A}(\mathsf{iO}(1^\kappa, C_1)) = 1]| \leq \mathsf{negl}(\kappa)$$

Functional Encryption

Now we recall the definition of functional encryption schemes [BSW11, O'N10].

Definition 2.6. *A functional encryption scheme* FE *is a tuple of PPT algorithms* $\mathsf{FE.Gen}$*,* $\mathsf{FE.Enc}$*,* $\mathsf{FE.KeyGen}$ *and* $\mathsf{FE.Dec}$ *such that*

- $\mathsf{FE.Gen}(1^\kappa)$*: takes a security parameter and outputs a pair of keys* $(\mathsf{mpk}, \mathsf{msk})$*;*
- $\mathsf{FE.Enc}(\mathsf{mpk}, m)$*: takes a public key* mpk *and a message* m*, it outputs a ciphertext* c*;*
- $\mathsf{FE.KeyGen}(\mathsf{msk}, f)$*: takes a secret key* msk *and a circuit* f *and outputs a function key* fsk_f*;*
- $\mathsf{FE.Dec}(\mathsf{fsk}_f, c)$*: takes a function key and a ciphertext, it outputs a string* y*.*

A functional encryption scheme FE *should be correct, in other words, for all* κ, n *and all message* $m \in \{0,1\}^n$*, for any circuit* f *defined on* $\{0,1\}^n$*,*

$$\Pr\left[\mathsf{FE.Dec}(\mathsf{fsk}_f, c) = f(m) \;\middle|\; \begin{array}{c}(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Gen}(1^\kappa) \\ c \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, m) \\ \mathsf{fsk}_f \leftarrow \mathsf{FE.KeyGen}(\mathsf{msk}, f)\end{array}\right] = 1$$

Most functional encryption schemes FE *used in our paper will be* ***compact, single-key selective secure****. To define the security, let us first define a security game* $\mathsf{Game}_{\kappa,\mathcal{A},b}$*:*

- *The adversary* $\mathcal{A}$ *first outputs two messages* m_0 *and* m_1 *where* $|m_0| = |m_1|$*;*
- *After receiving messages, the challenger runs* $\mathsf{FE.Gen}(1^\kappa)$ *to generate a key pair* $(\mathsf{mpk}, \mathsf{msk})$ *and computes the ciphertext* $c = \mathsf{FE.Enc}(\mathsf{mpk}, m_b)$ *and sends* (mpk, c) *back to* $\mathcal{A}$*;*
- $\mathcal{A}$ *then submits a function query* f *to the challenger and receives a function key* $\mathsf{fsk}_f = \mathsf{FE.KeyGen}(\mathsf{msk}, f)$ *where* $f(m_0) = f(m_1)$*;*
- *Finally* $\mathcal{A}$ *outputs a bit* b'*;*

We say FE *is single-key selective secure if for any poly sized adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* κ*,*

$$|\Pr[\mathsf{Game}_{\kappa,\mathcal{A},0} = 1] - \Pr[\mathsf{Game}_{\kappa,\mathcal{A},1} = 1]| \leq \mathsf{negl}(\kappa)$$

For a **multi-key selective secure** scheme, we allow an adversary to adaptively make function queries f to the challenger as long as $f(m_0) = f(m_1)$.

A functional encryption scheme is said to be **compact** [AJS15, BV15, AJ15] if for all κ and all $m \in \{0,1\}^*$, the running time (circuit size) of $\mathsf{FE.Enc}(\mathsf{mpk}, m)$ is bounded by $\mathsf{poly}(\kappa, |m|)$.

3 Decomposing Equivalence and dO

In this section, we discuss several basic definitions including decomposing equivalence and dO.

3.1 Partial Evaluations on Circuits

Definition 3.1. *Consider a circuit* C *defined on inputs of length* $n > 0$*, for any bit* $b \in \{0,1\}$*, a* ***partial evaluation*** *of* C *on* ***bit*** b *denoted as* $C(b, \cdot)$ *is a circuit defined on inputs of length* $n-1$*, where we harcode the input bit* x_1 *to* b*, and then simplify. To simplify, while there is a gate that has a hard-coded input, replace it with the appropriate gate or wire in the usual way (e.g.* $\mathsf{AND}(1, b)$ *gets replaced with the pass-through wire* b*, and* $\mathsf{AND}(0, b)$ *gets replaced with the constant* 0*). Then remove all unused wires.*

Also we can define a partial evaluation of a circuit C *on a* ***string*** x *which is repeatedly applying partial evaluations and simplifying bit by bit.*

From now on, whenever we use the expression $C(x, \cdot)$, we always refer to the result of simplifying C after hardcoding the prefix x.

3.2 Circuit Assignments

A binary tree T_n is a tree of depth $n+1$ where the root (whose depth is 1) is labeled ε (an empty string), and for any node that is not a root whose parent is labeled as x, it is labeled $x||0$ if it is a left-child of its parent; it is labeled as $x||1$ if it is a right-child of its parent.

Definition 3.2 (Tree Covering)**.** *We say a set of binary strings $\{x_i\}_{i=1}^{\ell}$ is a **tree covering** for all strings of length n if the following holds: for every string $x \in \{0,1\}^n$, there exists exactly one x_j in the set such that x_j is a prefix of x.*

A tree covering $\{x_i\}_{i=1}^{\ell}$ also can be viewed as a set of nodes in T_n such that for every leaf in the tree, the path from root ε to this leaf will pass exactly one node in the set.

Yet another equivalent formulation is that a tree covering is either (1) a set consisting of the root node of the tree, or (2) the union of two tree coverings for the two subtrees rooted at the children of the root node.

We define a partial order $\preceq$ on nodes in a binary tree. We say that $x \preceq y$ (alternatively, x is **above** y) if x is a prefix of y. We also extend our partial order $\preceq$ to tree coverings. We say a tree covering $TC_0 \preceq TC_1$, or TC_0 is **above** TC_1, if for every node u in TC_1, there exists a node v in TC_0 such that $v \preceq u$ (that is, v is equal to u or an ancestor of u). A tree covering TC_0 is **below** TC_1 if TC_1 is above TC_0. It is straightforward that if $TC_0 \preceq TC_1$, then $|TC_0| \leq |TC_1|$ where $|TC_0| = |TC_1|$ if $TC_0 = TC_1$. We can also extend $\preceq$ to compare tree coverings to nodes. We have $u \preceq TC$ if there is a node $v \in TC$ such that $u \preceq v$. $TC \preceq u$ if there exists a $v \in TC$ such that $v \preceq u$.

Definition 3.3 (Circuit Assignment)**.** *We say $L = \{(x_i, C_{x_i})\}_{i=1}^{\ell}$ is a **circuit assignment** with size ℓ where $\{x_i\}_{i=1}^{\ell}$ is a tree covering for T_n and $\{C_{x_i}\}_{i=1}^{\ell}$ is a set of circuits where C_{x_i} is assigned to the node x_i in the covering.*

We say a circuit assignment is valid if for each C_{x_i}, it is defined on input length $n - |x_i|$.

An evaluation of L on input x is defined as: find the unique x_j which is a prefix of $x = x_j||x_{-j}$ and return $C_{x_j}(x_{-j})$.

*We call each circuit in the assignment a **fragment**. The **cardinality** of the circuit assignment is the size of the tree covering, and the **circuit size** is the maximum size of any fragment in the assignment.*

A circuit assignment $L = \{(x_i, C_{x_i})\}_{i=1}^{\ell}$ naturally corresponds to a function: on input $y \in \{0,1\}^n$, scan the prefix of y from left to right until we find the smallest i such that $y_{[i]}$ equals to some x_j, output $C_{x_j}(y_{[i+1\cdots n]})$. We will override the notation and write this function as $L(x)$.

We associate a circuit C with the assignment $L_C = \{(\varepsilon, C)\}$ which assigns C to the root of the tree. Notice that L_C and C are equivalent as functions.

Before looking at the equivalence definitions, we need to give several basic operations for circuit assignments. These definitions will simplify our discussions later.

- $\mathsf{Decompose}(L, x)$: takes a circuit assignment L and a string x as parameters. This operation is invalid if x is not in the tree covering. The new circuit assignment has a slightly different tree covering: the new tree covering includes $x||0$ and $x||1$ but not x. It decomposes the fragment C_x into two fragments $C_x(0, \cdot)$ and $C_x(1, \cdot)$ and assigns them to $x||0$ and $x||1$ respectively.

- $\mathsf{CanonicalMerge}(L, x)$: operates on an assignment L where the tree covering includes both children of node x but not x itself. It takes two circuits $C_{x||0}, C_{x||1}$ assigned to the node $x||0$ and $x||1$ and merges them to get the following circuit $C_x(b, y) = (b \wedge C_{x||0}(y)) \vee (\overline{b} \wedge C_{x||1}(y))$ (Here we assume the output length of both circuits is 1. It is stragithforward to extend the definition to circuits with any output length). The new tree covering has x but not $x||0$ or $x||1$.

  One observation is that for any circuit assignment whose tree covering has $x||0$ and $x||1$ but not x and $C_{x||0}, C_{x||1}$ can not be simplified any further, we have the following relation: $\mathsf{Decompose}(\mathsf{CanonicalMerge}(L, x), x) = L$.

- $\mathsf{TargetedMerge}(L, x, C)$ operates on an assignment L where the tree covering includes both children of node x but not x itself. This operation is invalid if either $C(0, \cdot) \neq C_{x||0}$ or $C(1, \cdot) \neq C_{x||1}$ as circuits. It takes the two circuits $C_{x||0}, C_{x||1}$ assigned to the node $x||0$ and $x||1$ and merges them to get $C_x = C$. The new tree covering has x but not $x||0$ or $x||1$.

  We observe that

  - $\mathsf{Decompose}(\mathsf{TargetedMerge}(L, x, C), x) = L$ where $C_{x||0}$ and $C_{x||1}$ in L can not be simplified any further, and all the operations are valid
  - $\mathsf{TargetedMerge}(\mathsf{Decompose}(L, x), x, C) = L$ where C is the fragment at node x in L (as long as the operations are valid).

- $\mathsf{DecomposeTo}(L, x)$: takes a circuit assignment L and a string x as parameters. The operation is valid if $TC \preceq x$, where TC is the tree covering for L. Let u be ancestor of x in TC. Let $p_0 = u, p_1, \ldots, p_t = x$ be the path from the root u to x.

  $\mathsf{DecomposeTo}$ first sets $L_0 = L$, and then runs $L_i \leftarrow \mathsf{Decompose}(L_{i-1}, p_{i-1})$ for $i = 1, \ldots, t$. The output is the new circuit assignment $L' = L_t$. The new tree covering TC' for L' is the minimal TC' that is both below TC and contains x.

  We will also extend $\mathsf{DecomposeTo}$ to operate on circuits in addition to assignments, by first interpreting the circuit as an assignment, and performing $\mathsf{DecomposeTo}$ on the assignment.

- $\mathsf{DecomposeTo}(L, TC)$: It takes a circuit assignment L (if the first parameter is a circuit C, then $L = \{(C, \varepsilon)\}$) and a tree covering TC where TC is below the covering in L. This procedure keeps taking the lexicographically first circuit fragment C_x which x is not in TC and do $\mathsf{Decompose}(L, x)$. Because the covering in L is above TC, the procedure halts when the covering in the new circuit assignment is exactly TC.

- $\mathsf{CanonicalMerge}(L, TC)$: It takes a circuit assignment L and a tree covering TC where TC is below the covering in L. It repeatedly performs $\mathsf{CanonicalMerge}(L, x)$ at different x until the tree covering in the assignment becomes TC. To make the merging truly canonical, we need to specify an order that nodes are merged in. We take the convention that the lowest nodes in the tree are merged first, and between nodes in the same level, the leftmost nodes are merged first.

- $\mathsf{CanonicalMerge}(L)$: it canonically merges all the way to the root. In other words, the procedure keeps taking the lexicographically first circuit fragment pair $C_{x||0}$ and $C_{x||1}$ and doing $\mathsf{CanonicalMerge}(L, x)$ until the tree covering in the circuit assignment is $\{\varepsilon\}$, in other words, it becomes a single circuit.

Note that the functionality of a circuit assignment is preserved under applying any valid operation above.

3.3 Locally, Path, One Shot Decomposing Equivalence

We define several new equivalence notions for circuits based on the decomposing and merging operations defined above. First, we define a local equivalence condition on circuit assignments:

Definition 3.4 (locally decomposing equivalent)**.** *We say two circuit assignments $L_1 = \{(x_i, C_{x_i})\}, L_2 = \{(y_i, C'_{y_i})\}$ are (ℓ, s)-**locally decomposing equivalent** if the following hold:*

- *The circuit sizes of L_1, L_2 are at most s;*

- *The cardinalities of L_1, L_2 are at most ℓ;*

- *L_1 can be obtained from L_2 by applying $\mathsf{Decompose}(L_2, x)$ for some x or by applying $\mathsf{TargetedMerge}(L_2, x, C)$ for some x and C is the fragment assigned in L_1 to the string(node) x;*

Local decomposing equivalence (Local DE) means that we can transform L_1 into L_2 by making just a single local change, namely decomposing a node or merging two nodes. Notice that since decomposing a node does not change functionality, local DE implies that L_1 and L_2 compute equivalent functions. For any ℓ, s, (ℓ, s)-local decomposing equivalence forms a graph, where nodes are circuit assignments and edges denote local decomposing equivalence. Next, we define a notion of *path* decomposing equivalence for circuits (where circuits can be thought as nodes in the graph), which says that two circuits are equivalent if they are connected by a reasonably short path through the graph.

Definition 3.5 (path decomposing equivalent)**.** *We say two circuits C_1, C_2 are (ℓ, s, t)-**path decomposing equivalent** if there exist at most $t-1$ circuit assignments $L'_1, L'_2, \cdots, L'_{t-1}$ such that, for any $1 \le i \le t$, L'_{i-1} and L'_i are (ℓ, s)-locally decomposing equivalent, where $L'_0 = \{(\varepsilon, C_1)\}$ and $L'_t = \{(\varepsilon, C_2)\}$.*

The final notion of equivalence is a "one-shot" notion, which allows for exactly two steps to get between C_1 and C_2. Now the steps are not confined to be local, but instead the first step is allowed to decompose the root to a given tree covering, and the second then merges the tree covering all the way back to the root.

Definition 3.6 (one shot decomposing equivalent)**.** *Given two circuits C_0, C_1 defined on inputs of length n, we say they are τ-**one shot decomposing equivalent** or simply τ-**decomposing equivalent** if the following hold:*

- *There exists a tree covering $\mathcal{X} = \{x_i\}_i$ of size at most τ;*
- *For all $x_i \in \mathcal{X}$, $C_0(x_i, \cdot) = C_1(x_i, \cdot)$ (as circuits).*

An equivalent definition for "τ-one shot decomposing equivalent" is that there exists a tree covering $\mathcal{X}$ of size at most τ, such that $\mathsf{DecomposeTo}(\{(\varepsilon, C_0)\}, \mathcal{X}) = \mathsf{DecomposeTo}(\{(\varepsilon, C_1)\}, \mathcal{X})$, in other words, the tree coverings are the same and the corresponding fragments for each node are the same.

We note that since the operations defining path and one shot decomposing equivalence all preserve functionalities, we have that these notions imply standard functional equivalence for the circuits:

Lemma 3.7. *If C_0, C_1 are (ℓ, s, t)-path decomposing equivalent for any ℓ, s, t, or if C_0, C_1 are τ-one shot decomposing equivalent for any τ, then C_0, C_1 compute equivalent functions ($C_0(x) = C_1(x), \forall x \in \{0,1\}^n$).*

We also observe a partial converse:

Lemma 3.8. *Two circuits C_0, C_1 (defined on n bits string) are 2^n-one shot decomposing equivalent if and only if they are functionally equivalent ($C_0(x) = C_1(x), \forall x \in \{0,1\}^n$).*

Proof. We only need to show the case that functional equivalence implies 2^n-one shot decomposing equivalence. If C_0, C_1 are functionally equivalent, we can let the tree covering be $\mathcal{X} = \{0,1\}^n$. Because $C_0(x) = C_1(x)$ for all $x \in \{0,1\}^n = \mathcal{X}$, we have $\mathsf{DecomposeTo}(\{(\varepsilon, C_0)\}, \mathcal{X}) = \mathsf{DecomposeTo}(\{(\varepsilon, C_1)\}, \mathcal{X})$. Therefore C_0, C_1 are 2^n-one shot decomposing equivalent. □

3.4 Deciding Decomposing Equivalence

Definition 3.9. *A tree covering TC is a* witness *that $C_0 \equiv C_1$ if TC satisfies*

$$\mathsf{DecomposeTo}(\{(\varepsilon, C_0)\}, \mathcal{X}) = \mathsf{DecomposeTo}(\{(\varepsilon, C_1)\}, \mathcal{X})$$

In other words, decomposing C_0 and C_1 to TC gives the same circuit assignment (as in, the circuit fragments themselves are identical).

TC is an minimal *witness if, for all other TC' that are witnesses to $C_0 \equiv C_1$, we have that $TC \preceq TC'$. In particular, this means that TC is strictly smaller than all other witnesses.*

We define a node x as "good" for C_0, C_1 if $C_0(x, \cdot) = C_1(x, \cdot)$ as circuits. Notice that the children of a good node are also good. We say that a good node x is "minimal" if its parent is not good.

Lemma 3.10. *For any two equivalent circuits C_0, C_1, there is always exactly one minimal witness TC^*, and it consists of all of the minimal good nodes for C_0, C_1.*

Proof. First, any witness TC for $C_0 \equiv C_1$ must only contain good nodes. Moreover, if TC contains a non-minimal good node x, we can derive another witness $TC' \preceq TC$ by replacing x with its parent y, and removing all descendants of y. Thus any *minimal* witness must only contain *minimal* good nodes.

Now, since $C_0 \equiv C_1$, all the leaves are good. For each leaf, consider the path from the root to the leaf. There will be some node x on the path such that all nodes in the path before x are not good, but x and all nodes after x are good. Therefore, that x is an minimal good node. Moreover, no minimal good node can be a descendant of any other minimal good node (since no minimal good node can be the descendant of *any* good node). Therefore, the set of minimal good nodes form a tree covering. This tree covering always exists, and must also be minimal. □

Lemma 3.11. *τ-one shot decomposing equivalence can be decided deterministically in time $\tau \times \mathsf{poly}(n, \max\{|C_0|, |C_1|\})$. Moreover, if $C_0 \equiv C_1$, then the optimal witness TC^* can also be computed in this time.*

Proof. The algorithm is simple: process the nodes in a depth-first manner, and keep a global list R. When processing a node x, if $C_0(x, \cdot) = C_1(x, \cdot)$ as circuits, it adds x to R, and then does not recurse. Otherwise, it recurses on the children as normal. If the list R ever grows to exceed τ elements, it aborts the search and reports non-decomposing equivalence. If the search finishes with $|R| \leq \tau$, then it reports decomposing equivalence and outputs R.

The total running time is bounded by $O(n\tau \cdot \mathsf{poly}(\max\{|C_0|, |C_1|\}))$: at most $n\tau$ nodes are processed (up to τ nodes in R, plus their ancestors), and processing each node takes time proportional to the sizes of C_0, C_1. □

3.5 Relations Between Equivalence Notions

As noted above, our equivalence notions imply standard functional equivalence, and functional equivalence implies 2^n-one shot exploding equivalence. Here, we show some additional relationship between our definitions and functional equivalence. First, we show that one-shot and path equivalence are actually essentially the same.

Lemma 3.12. *If two circuits C_0, C_1 are $(t/2+1)$-one shot decomposing equivalent, then they are $(n+1, s, t)$-path decomposing equivalent where $s = \max\{|C_0|, |C_1|\}$.*

Proof. We start from the covering that has C_0 assigned to the root. We perform a depth-first traversal of the binary search tree consisting of the "bad" nodes: nodes for which the partial evaluations of C_0 and C_1 are different. Equivalently, we search over the ancestors of nodes in

the tree covering. There are $t/2$ such nodes. When we first visit a node on our way down the tree, we Decompose the fragment at that node to its children. When we visit a node x for the second time after processing both children, we merge the fragments in the two children, using a TargetedMerge toward the circuit $(C_1)_x$. This operation is always valid since for each child either: (1) the child is a "good" node, in which case the partial evaluations at that node is identical to the partial evaluation of $(C_1)_{x||b}$; or (2) the child is a "bad" node, in which case it was, by induction, already processed and replaced with the partial evaluation of $(C_1)_{x||b}$. The cardinality of any circuit assignment in this path is at most $n+1$ since we will only have fragments adjacent to the path from the root to the node we are visiting. The circuit size is moreover always bounded by $s = \max\{|C_0|, |C_1|\}$ because all the intermediate fragments are partial evaluations of either C_0 or C_1. Finally, the path performs an Decompose and TargetedMerge for each "bad" node, corresponding to t operations. □

Now we prove the converse, that path DE implies one-shot DE.

Lemma 3.13. *If two circuits C_0, C_1 are (ℓ, s, t)-path decomposing equivalent, then they are $(t/2+1)$-one shot decomposing equivalent*

Proof. If C_0, C_1 are (ℓ, s, t)-path decomposing equivalent, they are equivalent, and therefore there exists a minimal tree covering TC^*. We observe that, for each of the ancestors of nodes in TC^*, there must be a step in the path where that node is decomposed, and there must also be a step in the path where that node is merged. The number of ancestors for any tree covering is exactly one less than the size of the covering. From this, we deduce that $|TC^*| \leq t/2+1$. Since TC^* exists and the size is bounded by $t/2+1$, these two circuits are $(t/2+1)$-one shot decomposing equivalent. □

We emphasize that the above lemma and proof were independent of the bounds ℓ and s. Putting together Lemmas 3.12 and 3.13, we find that the path equivalence definition is independent of the parameters ℓ, s.

We also see that path decomposing equivalence can be computed efficiently, following Lemmas 3.11, 3.12, and 3.13.

We then show that path/one-shot decomposing equivalence is a strictly stronger notion than standard functional equivalence, when a reasonable bound is placed on the path length/witness size. The rough idea is the use the fact that, say, polynomial decomposing equivalence can be decided in polynomial time, whereas in general deciding equivalence is hard.

Lemma 3.14. *For any n, there exist two circuits on n bit inputs $C_0 \equiv C_1$ that are not $(2^{n-1}-1)$-one-shot decomposing equivalent.*

Proof. Let D_0, D_1 be two equivalent but non-identical circuits on 2 input bits (for example, two different circuits computing the XOR). Let TC^* be the tree covering consisting of all 2^{n-1} nodes in the layer just above the leaves. Let L_b for $b = 0, 1$ be the circuit assignment assigning D_b to every node in TC^*. Finally, Let C_b be the result of canonically merging L_b all the way to the root node.

Now, TC^* is clearly the optimal witness that $C_0 \equiv C_1$. Therefore, any witness must have size at least $|TC^*| = 2^{n-1}$. Therefore, C_0, C_1 are not $(2^{n-1} - 1)$ one-shot decomposing equivalent. □

Note that the above separation constructed exponentially-large C_0, C_1. We can even show a similar separation in the case where C_0, C_1 have polynomial size, assuming $P \neq NP$. Indeed, since poly-one shot decomposing equivalence is decideable in polynomial time, but functional equivalence is not (assuming $P \neq NP$), there must be circuit pairs that are equivalent but not poly-one shot decomposing equivalent.

Next, we even demonstrate an explicit ensemble of circuit pairs that are equivalent but not poly-decomposing equivlant, assuming one-way functions exist.

Lemma 3.15. *Assuming one-way functions exist, there is an explicit family of circuit pairs* (C_0, C_1) *that are equivalent, but are not* $\mathsf{poly}(n)$*-decomposing equivalent for any polynomial* $\mathsf{poly}(n)$.

Proof. Let PRG be a length-doubling pseudorandom generator (which can be constructed from any one-way function). Let $C_0(x) =$ "**return** 0" and $C_1(x) =$ '**return** 1 if $\mathsf{PRG}(x) = v$; 0 otherwise'' where v is uniformly chosen from $\{0,1\}^{2\kappa}$. When v is uniformly chosen, except with probability $\frac{1}{2^\kappa}$, v has no pre-image under PRG. Therefore, with probability $1 - \frac{1}{2^\kappa}$, C_0 and C_1 are functionally equivalent.

Next, assume there exists a polynomial τ and a non-negligible probability δ such that C_0 and C_1 are τ-decomposing equivalent with probability δ. Now we can build an adversary $\mathcal{B}$ for this length-doubling PRG:

- The adversary $\mathcal{B}$ gets u from the challenger;
- $\mathcal{B}$ prepares the following two circuits: $C_0(x) =$"**return** 0" and $C_1(x) =$ '**return** 1 if $\mathsf{PRG}(x) = u$; 0 otherwise''.
- $\mathcal{B}$ runs the algorithm to see if they are τ-decomposing equivalent. If the algorithm returns **true**, $\mathcal{B}$ guesses u is a truly random string; otherwise it guesses u is generated by PRG.

When u is generated by PRG, it will always return the correct answer since C_1 does not return 0 at some point but C_0 does; when u is truly random, the probability that $\mathcal{B}$ is correct equal to the probability C_0 and C_1 are τ-decomposing equivalent which is a non-negligible δ. So $\mathcal{B}$ has non-negligible advantage δ in breaking PRG. □

4 Decomposable Obfuscation

In this section, we give more discussions about decomposing equivalence and introduce dO. And finally we give the construction of dO from compact functional encryption schemes.

4.1 Locally, One Shot dO

Here, we give several new obfuscation definitions. Decomposable obfuscator, roughly, is an indistinguishability obfuscator, but where the indistinguishability security requirement only applies to pairs of circuits that are decomposing equivalent (as opposed to applying to all functionally equivalent circuits).

Definition 4.1. dO *with two PPT algorithms* {dO.ParaGen, dO.Eval} *is a* $\tau(n, s, \kappa)$*-decomposable obfuscator if the following conditions hold*

- ***Efficiency:*** dO.ParaGen, dO.Eval *are efficient algorithms;*
- ***Functionality preserving:*** dO.ParaGen *takes as input a security parameter* κ *and a circuit* C*, and outputs the description* $\hat{C}$ *of an obfuscated circuit. For all* κ *and all circuit* C*, for all input* $x \in \{0,1\}^n$*, we have* $\mathsf{dO.Eval}(\mathsf{dO.ParaGen}(1^\kappa, C), x) = C(x)$*;*
- ***Decomposing indistinguishability :*** *Consider a pair of PPT adversaries* $(Samp, D)$ *where Samp outputs a tuple* (C_0, C_1, σ) *where* C_0, C_1 *are circuits of the same size* $s = s(\kappa)$ *and input length* $n = n(\kappa)$*. We require that, for any such PPT* $(Samp, D)$*, if*

$$\Pr[C_0 \text{ is } \tau(n,s,\kappa)\text{-decomposing equivalent to } C_1 : (C_0, C_1, \sigma) \leftarrow Samp(\kappa)] = 1$$

then there exists a negligible function $\mathsf{negl}(\kappa)$ *such that*

$$\begin{aligned}|\Pr[D(\sigma, \mathsf{dO.ParaGen}(1^\kappa, C_0)) = 1] \\ - \Pr[D(\sigma, \mathsf{dO.ParaGen}(1^\kappa, C_1)) = 1]| \leq \mathsf{negl}(\kappa)\end{aligned}$$

Since 2^n-decomposing equivalence corresponds to standard equivalence, 2^n-dO is equivalent to the standard notion of iO. In this work, we will usually consider a much weaker setting, where τ is restricted to a polynomial.

The following obfuscator, called *local* dO (ldO), will be used to help us build dO. Roughly, ldO is an obfuscator for *circuit assignments* with the property that local changes to the assignment (that is, decomposing operations) are computationally undetectable.

Definition 4.2. ldO *with two PPT algorithms* {ldO.ParaGen, ldO.Eval} *is a locally decomposable obfuscator if the following conditions hold*

- ***Efficiency:*** ldO.ParaGen, ldO.Eval *are efficient algorithms;*
- ***Functionality preserving:*** ldO.ParaGen *takes as input a security parameter* κ*, a circuit assignment* L*, a cardinality bound* ℓ*, and a circuit size bound* s*. For all* κ *and all circuit assignment* L *with cardinality at most* ℓ *and circuit size at most* s*, for all input* $x \in \{0,1\}^n$*, we have* $\mathsf{ldO.Eval}(\mathsf{ldO.ParaGen}(1^\kappa, L, \ell, s), x) = L(x)$*;*

- ***Local decomposing indistinguishability:*** *Consider polynomials $\ell = \ell(\kappa)$ and $s = s(\kappa)$. For any such polynomials, and any pair of PPT adversaries $(Samp, D)$, we require that if*

$$\Pr[L_0 \text{ is } (\ell(\kappa), s(\kappa))\text{-local decomp. equiv. to } L_1 : (L_0, L_1, \sigma) \leftarrow Samp(\kappa)] = 1$$

then there exists a negligible function $\mathsf{negl}(\kappa)$ such that

$$\begin{aligned}|\Pr[D(\sigma, \mathsf{ldO.ParaGen}(1^\kappa, L_0, \ell, s)) = 1] \\ - \Pr[D(\sigma, \mathsf{ldO.ParaGen}(1^\kappa, L_1, \ell, s)) = 1]| \leq \ell \cdot \mathsf{negl}(\kappa)\end{aligned}$$

We will also consider a stronger variant, called sub-exponentially secure local dO, where in the definition of local decomposing indistinguishability, the negligible function negl is replaced by a subexponential function subexp.

4.2 ldO implies dO

Now we show that the existence of ldO implies the existence of dO.

Lemma 4.3. *If ldO exists, then τ-dO exists, where the loss in the security reduction is $2(\tau - 1)$. In particular, if polynomially secure ldO exists, then τ-dO exists for any polynomial function τ. Moreover, if subexponentially secure ldO exists, then 2^n-dO, and hence iO, exists.*

Proof. The construction of ldO from dO is the natural one: to obfuscate a circuit C, we simply consider the circuit as a circuit assignment with C assigned to the root node, and obfuscate this circuit assignment. We take the maximum cardinality for ldO to be $n + 1$ and the circuit size to be $|C|$.

- $\mathsf{dO.ParaGen}(1^\kappa, C) = \mathsf{ldO.ParaGen}(1^\kappa, \{(\varepsilon, C)\}, n + 1, |C|)$;
- $\mathsf{dO.Eval}(\mathsf{params}, x) = \mathsf{ldO.Eval}(\mathsf{params}, x)$;

Efficiency and functionality preservation follow immediately from the underlying ldO. Now we focus on security. Let (Samp, D) be two PPT adversaries, and s, n be polynomials in κ. Suppose the circuits C_0, C_1 outputted by $\mathrm{Samp}(\kappa)$ always have the same size $s(\kappa)$, same input length $n(\kappa)$, and are $\tau(n, s, \kappa)$-decomposing equivalent with probability 1. Then C_0 and C_1 are also $(n + 1, s, 2(\tau - 1))$-path decomposing equivalent by Lemma 3.12. By the definition of path decomposing equivalence and Lemma 3.11 (which states that the minimum tree covering is efficiently computable), there exist $L'_1, L'_2, \cdots, L'_{2(\tau-2)}, L'_{2(\tau-1)-1}$ and $L'_0 = \{(\varepsilon, C_0)\}, L'_{2(\tau-1)} = \{(\varepsilon, C_1)\}$ such that any two adjacent circuit assignments are $(n + 1, s)$-locally decomposing equivalent. So we have that

$$\begin{aligned}&|\Pr[D(\mathsf{dO.ParaGen}(1^\kappa, C_0))] - \Pr[D(\mathsf{dO.ParaGen}(1^\kappa, C_1))]| \\ &\leq \sum_{i=1}^{2(\tau-1)} \left| \begin{array}{l}\Pr[D(\mathsf{ldO.ParaGen}(1^\kappa, L'_{i-1}), n + 1, |C_0|)] \\ \quad - \Pr[D(\mathsf{ldO.ParaGen}(1^\kappa, L'_i), n + 1, |C_0|)]\end{array} \right| \\ &\leq 2(\tau - 1) \cdot \epsilon(\kappa)\end{aligned}$$

Here, ϵ is the advantage of the following adversary pair (Samp', D) in the local dO security game (where D is from above). Samp' runs $(C_0, C_1, \sigma) \leftarrow \text{Samp}'$, computes the path $L'_0, \cdots, L'_{2(\tau-1)}$, chooses a random $i \in [2(\tau-1)]$, and outputs (L'_{i-1}, L'_i, σ).

Therefore, as desired, we get an adversary for the local dO where the loss is $2(\tau-1)$. If we assume the polynomial hardness of ldO, the adversary (Samp', D) must have negligible advantage ϵ, and so we get $\tau - \mathsf{dO}$ for any polynomial τ. If we assume the subexponential hardness of ldO, we can set κ so that $\epsilon = 2^{-n}\mathsf{negl}(\kappa)$ for some negligible function negl. In this case, we even get 2^n-dO, which is equivalent to iO. In the regime of subexponential hardness, we can even set $\epsilon = 2^{-n}\mathsf{subexp}(\kappa)$ for some subexponential function subexp, in which case we get subexponentially secure 2^n-dO and hence subexponentially secure iO. □ □

Next, we focus on constructing ldO, which we now know is sufficient for constructing dO.

4.3 Compact FE implies dO

Theorem 4.4. *If compact single-key selective secure functional encryption schemes exist, then there exists local decomposable obfuscators* ldO.

With theorem 4.4 and lemma 4.3, we have the following theorem 4.5.

Theorem 4.5. *If compact single-key selective secure functional encryption schemes exist, then there exist decomposable obfuscators* dO.

Now we prove theorem 4.4.

Proof. Let us first give the construction of our $\mathsf{ldO}.\mathsf{ParaGen}$(see algorithm 1) where FE is a compact functional encryption scheme, SKE is a symmetric key encryption scheme and PRG is a pseudorandom generator.

Algorithm 1 does the following: for a circuit assignment defined on n bits, it generates $n+1$ layers; the i-th layer has two pairs of master public key and secret key $(\mathsf{mpk}_i^0, \mathsf{msk}_i^0)$ and $(\mathsf{mpk}_i^1, \mathsf{msk}_i^1)$ and two pairs of function keys $\mathsf{fsk}_i^0, \mathsf{fsk}_i^1$ allowing you to evaluate the function on a ciphertext.

For each function $f_i^{b,Z_i^b}(1 \le i \le n)$, it basically computes a partial evaluation of an input circuit and encrypts it under two different functional encryption schemes (See Algorithm 2). But instead of doing this, this function also allows us to cheat and output a result given a secret key and the corresponding ciphertext from Z.

For each function f_{n+1}^b, it is given a circuit with no input, and simply evaluates it (see algorithm 3).

Finally it generates the ciphertexts c_0, c_1 corresponding to the root of the tree by CGen algorithm.

Algorithm 1 locally decomposable obfuscator $\mathsf{IdO.ParaGen}$

procedure $\mathsf{IdO.ParaGen}(1^\kappa, L = \{(x_i, C_{x_i})\}, \ell, s)$
  for $i = 1, 2, \cdots, n, n+1$ **do**
    $(\mathsf{mpk}_i^b, \mathsf{msk}_i^b) \leftarrow \mathsf{FE.Gen}(1^\kappa)$ for $b \in \{0,1\}$
  end for
  prepare a list of secret keys $\mathsf{sk}_{i,j}^b \leftarrow \mathsf{SKE.KeyGen}(1^\kappa)$ for $1 \le i \le n, 1 \le j \le \ell$ and $b \in \{0,1\}$
  prepare $Z_i^b = Z_{i,1}^b, Z_{i,2}^b, \cdots, Z_{i,\ell}^b$ for $1 \le i \le n$ and $b \in \{0,1\}$ where $Z_{i,j}^b = \mathsf{SKE.Enc}(\mathsf{sk}_{i,j}^b, 0^{t_1})$ and t_1 is a length bound;
  generate c_0, c_1 by calling a recursive algorithm $\mathsf{CGen}(\varepsilon, L)$
  for $i = 1, 2, \cdots, n$ **do**
    $\mathsf{fsk}_i^b \leftarrow \mathsf{FE.KeyGen}(\mathsf{msk}_i^b, f_i^{b,Z_i^b})$ for $b \in \{0,1\}$
  end for
  $\mathsf{fsk}_{n+1}^b \leftarrow \mathsf{FE.KeyGen}(\mathsf{msk}_{n+1}^b, f_{n+1}^b)$ for $b \in \{0,1\}$
  return the parameters $\{c_0, c_1, \{\mathsf{mpk}_i^0, \mathsf{mpk}_i^1\}_{i=1}^{n+1}, \{\mathsf{fsk}_i^0, \mathsf{fsk}_i^1\}_{i=1}^{n+1}\}$
end procedure

Algorithm 2 f_i^{b,Z_i^b} for $1 \le i \le n$

procedure $f_i^{b,Z_i^b}(C, K, \sigma, \mathsf{sk})$
  Hardcoded : Z_i^b
  if $\sigma \neq 0$ **then**
    return $\mathsf{SKE.Dec}(\mathsf{sk}, Z_{i,\sigma}^b)$
  else
    $C' \leftarrow C(b, \cdot)$ and pad C' to have length s
    return $\{\mathsf{FE.Enc}(\mathsf{mpk}_{i+1}^0, \langle C', K_{i+1}^0, 0, 0^{t_2}\rangle; r_1),$
      $\mathsf{FE.Enc}(\mathsf{mpk}_{i+1}^1 \langle C', K_{i+1}^1, 0, 0^{t_2}\rangle; r_2)\}$ where
      $K_{i+1}^0 \leftarrow r_3$
      $K_{i+1}^1 \leftarrow r_4$
      using randomness $r_1, r_2, r_3, r_4 \leftarrow \mathsf{PRG}(K)$
  end if
end procedure

Algorithm 3 f_{n+1}^b

procedure $f_{n+1}^b(C, K, \sigma, \mathsf{sk})$
  return the evaluation of C
end procedure

Algorithm 4 generating c_0, c_1 recursively

procedure $\mathsf{CGen}(x, L)$
  if L only contains one pair, it must be (x, C_x) **then**
    x is a node in the tree covering of the input circuit assignment
    Generate $K^b \leftarrow \{0,1\}^\kappa$ for $b \in \{0,1\}$
    $c_b \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}_d^b, \langle C_x, K^b, 0, 0^{t_2}\rangle)$ for $b \in \{0,1\}$, and $d = |x| + 1$
    return c_0, c_1
  end if
  Split L into L_0, L_1 where L_0 contains all the pairs (y, C_y) where y starts with $x||0$ and L_1 contains all the pairs (y, C_y) where y starts with $x||1$
  $(c_0', c_1') \leftarrow \mathsf{CGen}(x||0, L_0)$ and $(c_0'', c_1'') \leftarrow \mathsf{CGen}(x||1, L_1)$
  Choose an integer j_0 randomly from 1 to ℓ that has not been used yet in Z_d^0 and replace Z_{d,j_0}^0 with $\mathsf{SKE.Enc}(\mathsf{sk}_{d,j_0}^0, \langle c_0', c_1'\rangle)$
  Choose j_1 in the same way and replace Z_{d,j_1}^1 with $\mathsf{SKE.Enc}(\mathsf{sk}_{d,j_1}^1, \langle c_0'', c_1''\rangle)$
  return c_0, c_1 where $c_0 = \mathsf{FE.Enc}(\mathsf{mpk}_d^0, \langle \perp, \perp, j_0, \mathsf{sk}_{d,j_0}^0\rangle)$ and $c_1 = \mathsf{FE.Enc}(\mathsf{mpk}_d^1, \langle \perp, \perp, j_1, \mathsf{sk}_{d,j_1}^1\rangle)$
end procedure

Evaluation and Correctness

Now let us look at how $\mathsf{IdO.Eval}$ works. By fixing the first two ciphers and keys, given a input $x \in \{0,1\}^n$,

- It begins with c_0, c_1;
- For $i = 1, 2, \cdots, n$, it picks the function key $\mathsf{fsk}_i^{x_i}$ and c_{x_i}; then does the update: $(c_0, c_1) \leftarrow \mathsf{FE.Dec}(\mathsf{fsk}_i^{x_i}, c_{x_i})$;
- Finally we can either output $\mathsf{FE.Dec}(\mathsf{fsk}_{n+1}^0, c_0)$ or $\mathsf{FE.Dec}(\mathsf{fsk}_{n+1}^1, c_1)$;

$\mathsf{IdO.Eval}(c_0, c_1, \{\mathsf{mpk}_i^0, \mathsf{mpk}_i^1\}_{i=1}^{n+1}, \{\mathsf{fsk}_i^0, \mathsf{fsk}_i^1\}_{i=1}^{n+1}, \cdots)$ actually has the same functionalities with the circuit assignment L since basically on input x, it finds a fragment corresponding to a prefix y of $x = y||x'$ and keeps doing partial evaluations on each input bit of x'. Here y is a node in the tree covering. Because of the way CGen works, after applying the partial evaluations corresponding to the string y, the ciphertext corresponding to C_y, in other words $c_b \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}_d^b, \langle C_y, K^b, 0, 0^{t_2}\rangle)$ for $b \in \{0,1\}$, will be recovrered from Z. Since the cardinality is at most ℓ, ℓ different $Z_{i,j}^b$ in a single layer Z_i^b are enough for use.

Efficiency

Let us look at the parameter size. All the master keys $\{\mathsf{mpk}_i^0, \mathsf{mpk}_i^1\}_{i=1}^{n+1}$ are of length $\mathsf{poly}(\kappa)$. t_2 is the length of a secret key for SKE scheme so it is also of $\mathsf{poly}(\kappa)$. And we assume FE is a compact functional encryption scheme which means the size of ciphertexts c_0, c_1 is bounded by

$O(\mathsf{poly}(s, \log \ell, \kappa))$ and also the size of f circuit is bounded by $O(\mathsf{poly}(s, \ell, \kappa))$ which implies the size of $\{\mathsf{fsk}_i^b\}$ is bounded by $O(\mathsf{poly}(s, \ell, \kappa))$. Finally t_1 is bounded by $O(\mathsf{poly}(s, \log \ell, \kappa))$.

So $\mathsf{IdO.ParaGen}$ and $\mathsf{IdO.Eval}$ run in time $\mathsf{poly}(s, \ell, n, \kappa)$.

Security

Without loss of generality, we have two circuit assignments L_0 and L_1 where $\mathsf{Decompose}(L_0, x) = L_1$. We are going to prove the indistinguishability when we are given either L_0 or L_1.

- **Hyb 0**: Here, an adversary is given an instance $\mathsf{IdO.ParaGen}(1^\kappa, L_0, \ell, s)$. In the process of generating c_0, c_1, we get L' during the execution of CGen at x where L' is the current partial circuit assignment corresponding to the subtree rooted at x. Since L' only contains (x, C_x), CGen will return $\mathsf{FE.Enc}(\mathsf{mpk}_d^b, \langle C_x, K^b, 0, 0^{t_2}\rangle)$ for $b \in \{0, 1\}$ and $d = |x| + 1$; we denote them as $\hat{c}_0, \hat{c}_1$.

- **Hyb 1**: In this hybrid, we change Z_d^b. Assume $\hat{c}_{b,0}, \hat{c}_{b,1} = \mathsf{FE.Dec}(\mathsf{fsk}_d^b, \hat{c}_b)$. In $\mathsf{IdO.ParaGen}$, Z_d^b are assigned to an array of encryptions of 0^{t_1} before calling CGen. We instead choose random j_0, j_1 from the unused indices (not used in CGen process) and change Z_{d,j_b}^b to be the encryption of $\langle \hat{c}_{b,0}, \hat{c}_{b,1}\rangle$. Since an adversary does not have any secret key $\mathsf{sk}_{i,j}^b$, SKE security implies **Hyb 0** and **Hyb 1** are indistinguishable.

- **Hyb 2**: In this hybrid, we change the ciphertexts $\hat{c}_0, \hat{c}_1$ to

  $$\hat{c}_b = \mathsf{FE.Enc}(\mathsf{mpk}_d^b, \langle \bot, \bot, j_b, \mathsf{sk}_{d,j_b}^b\rangle)$$

  where $\bot$ means filling it with zeroes and j_b are the indices chosen in **Hyb 1**. Notice that

  $$f_d^{b,Z_d^b}(\bot, \bot, j_b, \mathsf{sk}_{d,j_b}^b) = f_d^{b,Z_d^b}(C_x, K^b, 0, 0^{t_2})$$

  Therefore, FE security means **Hyb 1** and **Hyb 2** are indistinguishable.

- **Hyb 3**: In this hybrid, we change Z_{d,j_0}^0 and Z_{d,j_1}^1. In **Hyb 1**, $\hat{c}_{b,0}, \hat{c}_{b,1}$ were computed using the randomness from a pseudorandom generator. In **Hyb 2**, we removed the seed feed to PRG. Therefore we can replace $\hat{c}_{b,0}, \hat{c}_{b,1}$ to be the values computed using uniformly chosen randomness. Indistinguishability from **Hyb 2** easily follows from PRG security. We observe that the distribution of the instances in **Hyb 3** is identical to the distribution of $\mathsf{IdO.ParaGen}(1^\kappa, L_1, \ell, s)$.

This completes our proof for theorem 4.4. □

5 Applications

5.1 Notations

Before all the applications, let us first introduce several definitions for convenience.

We now define a decomposing-compatible pseudorandom function. The construction [GGM86] satisfies the definition below.

Definition 5.1. *A decomposing compatible pseudorandom function* DPRF *consists the following algorithms* DPRF.KeyGen *and* DPRF.Eval *where*

- DPRF.Eval *takes a input of length n and the output is of length $p(n)$ where p is a fixed polynomial;*
- *(*PRF ***Security)*** *For any poly sized adversary $\mathcal{A}$, there exists a negligible function* negl*, for any string $y_0 \in \{0,1\}^n$ and any κ,*

$$|\Pr[\mathcal{A}(\mathsf{DPRF.Eval}(S, y_0)) = 1] - \Pr[\mathcal{A}(r) = 1]| \leq \mathsf{negl}(\kappa)$$

  where $S \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$ and $r \in \{0,1\}^{p(n)}$ is a uniformly random string.

- *(*DPRF ***Security)*** *Consider the following game, let $\mathsf{Game}_{\kappa,\mathcal{A},b}$ be*
  - *The challenger prepares $S \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$;*
  - *The adversary makes queries about x and gets $\mathsf{DPRF.Eval}(S, x)$ back from the challenger;*
  - *The adversary gives a tree covering TC and $y^* \in TC$ to the challenger where y^* is not a prefix of any x that has been asked;*
  - *The challenger sends the assignment D_b back to the adversary $\mathcal{A}$ where*
    - *D_0: let the circuit D to be $D(\cdot) = \mathsf{DPRF.Eval}(S, \cdot)$ defined on $\{0,1\}^n$, the circuit assignment is $\mathsf{DecomposeTo}(D, TC)$. We observe that the fragment corresponding to y is $\mathsf{DPRF.Eval}(S, y, \cdot)$ defined on $\{0,1\}^{n-|y|}$.*
    - *D_1: For each $y \neq y^* \in TC$, let the fragment corresponding to y be $D_y(\cdot) = \mathsf{DPRF.Eval}(S, y, \cdot)$ defined on $\{0,1\}^{n-|y|}$ and for y^*, $D_{y^*}(\cdot) = \mathsf{DPRF.Eval}(S', y^*, \cdot)$ defined on $\{0,1\}^{n-|y^*|}$ where $S' \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$.*
  - *The adversary can keep making queries about x which does not have prefix y^* and gets $\mathsf{DPRF.Eval}(S, x)$ back from the challenger;*
  - *The output of this game is the output of $\mathcal{A}$.*

  For any poly sized adversary $\mathcal{A}$, there exists a negligible function negl *such that:*

$$|\Pr[\mathsf{Game}_{\kappa,\mathcal{A},0} = 1] - \Pr[\mathsf{Game}_{\kappa,\mathcal{A},1} = 1]| \leq \mathsf{negl}(\kappa)$$

Let us define an another operation on a circuit assignment and a circuit.

Definition 5.2. *By given a circuit C and a circuit assignment L where C takes two inputs x and $L(x)$, $C(\cdot, L(\cdot))$ is a circuit assignment defined below:*

- *Let TC be the tree covering inside $L = \{(x, D_x)\}_{x \in TC}$.*
- *Let $L' = \mathsf{DecomposeTo}(C, TC) = \{(x, C_x)\}_{x \in TC}$*

- *For each fragment in the output circuit assignment corresponding to $x \in TC$, it is $C_x(\cdot, D_x(\cdot))$ simplified, which is defined on $\{0,1\}^{n-|x|}$.*

We can also define similar operations on several circuit assignments and one circuit as long as these circuit assignments have the same tree covering. In other words, let $L_1, \cdots, L_m (L_i = \{(x, D_x^i)\}$ are circuit assignments with the same tree covering TC, then $C(\cdot, L_1(\cdot), L_2(\cdot), \cdots, L_m(\cdot))$ is a circuit assignment whose fragment corresponding to $y \in TC$ is $C(y, \cdot, D_y^1(\cdot), \cdots, D_y^m(\cdot))$ simplified.

Then we have the following lemma:

Lemma 5.3. *For any two circuits C, D where D takes a single input x and C takes two inputs x and $D(x)$, for any tree covering TC, we have*

$$\mathsf{DecomposeTo}(C(\cdot, D(\cdot)), TC) = C(\cdot, [\mathsf{DecomposeTo}(D, TC)](\cdot))$$

For $m+1$ circuits $C, D_1, D_2, \cdots, D_m$, where $D_1, \cdots, D_m$ take a single input x and C takes x and $D_1(x) \cdots D_m(x)$ as inputs, we have

$$\begin{aligned}&\mathsf{DecomposeTo}(C(\cdot, D_1(\cdot), \cdots, D_m(\cdot)), TC)\\ &\quad = C(\cdot, \mathsf{DecomposeTo}(D_1, TC), \cdots, \mathsf{DecomposeTo}(D_m, TC))\end{aligned}$$

Proof. Let us first look at the left side. It is a circuit assignment with the tree covering TC. For the fragment corresponding to $y \in TC$, it is the partial evaluation of $C(\cdot, D(\cdot))$ on y.

For the right side, we first have a circuit assignment $\mathsf{DecomposeTo}(D, TC)$ where the fragment corresponding to y is $D(y, \cdot)$. So by the definition of our operation, the fragment corresponding to y in the right side is $C(y, \cdot, D(y, \cdot))$ simplified. Since each pair of fragments are the same, the left side is equal to the right side. □

5.2 Short Signatures

Here, we show how to use dO to build short signatures, following [SW14]. As in [SW14], we will construct statically secure signatures.

The signature is simply of the following form $f(\mathsf{DPRF.Eval}(S, m))$ where f is a one-way function.

Definition 5.4. *A signature scheme SS consists of the following algorithms:*

- $\mathsf{SS.Setup}(1^\kappa)$*: it outputs a verification key vk and a signature key sk;*
- $\mathsf{SS.Sign}(\mathsf{sk}, m)$*: it is a deterministic procedure; it takes a signature key and a message, then outputs a signature σ;*
- $\mathsf{SS.Ver}(\mathsf{vk}, m, \sigma)$*: it is a deterministic algorithm; it takes a verification key, a message m and a signature σ, it outputs 1 if it accepts; 0 otherwise.*

We say a short signature scheme is correct if for any message $m \in \{0,1\}^\ell$:

$$\Pr\left[\mathsf{SS.Ver}(\mathsf{vk}, m, \sigma) = 1 \;\middle|\; \begin{matrix}(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{SS.Setup}(1^\kappa) \\ \sigma \leftarrow \mathsf{SS.Sign}(\mathsf{sk}, m)\end{matrix}\right] = 1$$

We now define security for short signatures.

Definition 5.5. *We denote $\mathsf{Game}_{\kappa,\mathcal{A}}$ to be the following where κ is the security parameter and $\mathcal{A}$ is an adversary:*

- *First $\mathcal{A}$ announces a message m^* of length ℓ;*
- *The challenger gets m^* and prepares two keys sk and vk; it then sends vk back to $\mathcal{A}$;*
- *$\mathcal{A}$ can keep making queries m' to the challenger and gets $\mathsf{Sign}(\mathsf{sk}, m')$ back for any $m' \neq m^*$;*
- *Finally $\mathcal{A}$ sends a forged signature σ^* and the output of the game is $\mathsf{Ver}(\mathsf{vk}, m^*, \sigma^*)$.*

We say SS is secure if for any polysized $\mathcal{A}$, there exists a negligible function negl,

$$\Pr[\mathsf{Game}_{\kappa,\mathcal{A}} = 1] \leq \mathsf{negl}(\kappa)$$

Construction.

We now give a signature scheme where signatures are short. The construction is similar with that in [SW14] but we use dO instead of iO. Our SS has the following algorithms:

- $\mathsf{SS.Setup}(1^\kappa)$: it takes a security parameter κ and prepares a key $S \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$. S is the secret key sk. Then it prepares the verification key (which is a description of an obfuscated circuit) $\mathsf{vk} \leftarrow \mathsf{dO.ParaGen}(1^\kappa, V(\cdot, \mathsf{DPRF.Eval}(S, \cdot)))$ where V is given in Figure 5 (we will pad programs to a length upper bound before applying dO).

Algorithm 5 Verification Algorithm

procedure $V(m, \sigma, \mathsf{DPRF.Eval}(S, m))$
    it computes $\sigma' \leftarrow \mathsf{DPRF.Eval}(S, m)$
    if $f(\sigma) = f(\sigma')$ **then**
        return 1
    else
        return 0
    end if
end procedure

- $\mathsf{SS.Sign}(\mathsf{sk}, m) = \mathsf{DPRF.Eval}(S, m)$
- $\mathsf{SS.Ver}(\mathsf{vk}, m, \sigma) = \mathsf{dO.Eval}(\mathsf{vk}, \{m, \sigma\})$

It is straightforward to see that the construction satisfies correctness.

Security

Theorem 5.6. *If* dO *is a secure* poly-dO*,* DPRF *is a secure decomposing compatible* PRF*, and* f *is a one-way function, then the construction above is a short secure signature scheme.*

Proof. We prove the security through a sequence of hybrid experiments.

- **Hyb 0:** In this hybrid, we are in $\mathsf{Game}_{\kappa,\mathcal{A}}$;
- **Hyb 1:** In this hybrid, since the challenger gets m^* before it releases vk, we decompose the circuit to get $L = \mathsf{DecomposeTo}(V(\cdot, \mathsf{DPRF.Eval}(S, \cdot)), m^*)$. By lemma 5.3, the circuit assignment is $V(\cdot, \mathsf{DecomposeTo}(\mathsf{DPRF.Eval}(S, \cdot), m^*))$.

  Therefore we have that the distributions $\mathsf{dO.ParaGen}(1^\kappa, V(\cdot, \mathsf{DPRF.Eval}(S, \cdot)))$ and $\mathsf{dO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L))$ are indistinguishable, since these two circuits are $\ell + 1$-decomposing equivalent by applying dO,
- **Hyb 2:** This is the same as **Hyb 1**, except that we replace the fragment in L corresponding to m^* — which is "**return** $\mathsf{DPRF.Eval}(S, m^*)$" — by "**return** $\mathsf{DPRF.Eval}(S', m^*)$" where $S' \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$ is a fresh random DPRF key that is independent of S. We call the new circuit assignment L'. **Hyb 1** and **Hyb 2** are indistinguishable because of the DPRF security.
- **Hyb 3:** This is the same as **Hyb 2**, except that we replace the fragment in L', which is "**return** $\mathsf{DPRF.Eval}(S', m^*)$" by "**return** r^*" where r^* is a uniformly random string. We call the new circuit assignment L''. As we don't have S' in the program anywhere except this fragment, **Hyb 2** and **Hyb 3** are indistinguishable because of the PRF security.

  We find that in $\mathsf{CanonicalMerge}(L'')$, the fragment corresponding to m^* is: on input σ, it returns 1 if $f(\sigma) = v^*$; 0 otherwise, where $v^* = f(r^*)$ for a uniformly random r^*.

Lemma 5.7. *If there exists a poly sized adversary* $\mathcal{A}$ *for Hyb 3, then we can break one-way function* f*.*

Proof. Given z^* which is $f(r^*)$ for a truly random r^*, we can actually simulate **Hyb 3**. If we successfully find a forged signature for **Hyb 3** with non-negligible probability, it is actually a pre-image of z^* which means we break one-way function with non-negligible probability. □

This completes the security proof. □

5.3 Universal Samplers

Here we construct universal samplers from dO. For the sake of simplicity, we will show how to construct samplers meeting the one-time static definition from [HJK+16]. However, note that the same techniques also can be used to construct the more complicated k-time interactive simulation notion of [GPSZ16].

Let US denote an universal sampler. It has the following procedures:

- $\mathsf{params} \leftarrow \mathsf{US.Setup}(1^\kappa, 1^\ell, 1^t)$: the Setup procedure takes a security parameter κ, a program size upper bound ℓ and a output length t and outputs an parameter params;
- $\mathsf{US.Sample}(\mathsf{params}, C)$ is a deterministic procedure that takes a params and a sampler C of length at most ℓ where C outputs a sample of length t. This procedure outputs a sample s;
- $\mathsf{params}' \leftarrow \mathsf{US.Sim}(1^\kappa, 1^\ell, 1^t, C^*, s^*)$ takes a security parameter κ, a program size upper bound ℓ and a output length t, also a circuit C^* and a sample s^* in the image of C^*; it outputs a parameter.

Correctness

For any C^* and s^* in the image of C^*, and for any $\ell \geq |C^*|$, and t is a upper bound for C^*'s outputs, we have

$$\Pr\left[\mathsf{US.Sample}(\mathsf{params}', C^*)] = s^* \,|\, \mathsf{params}' \leftarrow \mathsf{US.Sim}(1^\kappa, 1^\ell, 1^t, C^*, s^*)\right] = 1$$

Security

For any ℓ and t, for any C^* of size at most ℓ and output size at most t, for any poly sized adversary $\mathcal{A}$, there exists a negligible function negl, such that

$$\Big|\Pr[\mathcal{A}(\mathsf{params}, C^*) = 1 \,|\, \mathsf{params} \leftarrow \mathsf{US.Setup}(1^\kappa, 1^\ell, 1^t)] - \Pr\left[\mathcal{A}(\mathsf{params}', C^*) = 1 \,\middle|\, \begin{array}{l} \mathsf{params}' \leftarrow \mathsf{US.Sim}(1^\kappa, 1^\ell, 1^t, C^*, s^*), \\ s^* \underset{R}{\leftarrow} C^*(\cdot) \end{array}\right]\Big| \leq \mathsf{negl}(\kappa)$$

where $s^* \underset{R}{\leftarrow} C^*(\cdot)$ means s^* is a truly random sample from $C^*(\cdot)$.

Construction

Now we give the detailed construction for our universal sampler:

- Define U to be the size upper bound among all the circuits being obfuscated in our proof (not the size of circuits fed into the universal sampler). It is straightforward to see that $U = \mathsf{poly}(\kappa, \ell, t)$; Whenever we mention $\mathsf{dO.ParaGen}(1^\kappa, C)$, we will pad C to have size U before feeding it to dO.
- For simplicity, we assume circuits C fed into the universal sampler is always padded to length ℓ so that we can consider only circuits of a fixed size.
- $\mathsf{US.Setup}(1^\kappa, 1^\ell, 1^t)$ randomly samples a key $S \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$, and constructs a circuit $\mathsf{Sampler}$ (see algorithm 6) as follows: on input circuit C of size ℓ, it outputs a sample based on the randomness generated by DPRF; and the output of the procedure $\mathsf{US.Setup}$ is $\mathsf{params} = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{Sampler}(\cdot, \mathsf{DPRF.Eval}(S, \cdot)))$.

Algorithm 6 Sampler Algorithm

procedure $\mathsf{Sampler}(C = c_1 c_2 \cdots c_\ell, \mathsf{DPRF.Eval}(S, C))$
  $r_C \leftarrow \mathsf{DPRF.Eval}(S, C)$
  return $C(; r_C)$
end procedure

- $\mathsf{US.Sample}(\mathsf{params}, C)$: it simply outputs $\mathsf{dO.Eval}(\mathsf{params}, C)$;
- $\mathsf{US.Sim}(1^\kappa, 1^\ell, 1^t, C^*, s^*)$: it randomly samples a key $S \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$, let L be a circuit assignment $\mathsf{Sampler}(\cdot, \mathsf{DecomposeTo}(\mathsf{DPRF.Eval}(S, \cdot), C^*))$. And finally it replaces the fragment corresponding to C^* in L with "**return** s^*" instead of returning $C^*(; \mathsf{DPRF.Eval}(S, C^*))$. Let $\mathsf{Sampler}' = \mathsf{CanonicalMerge}(L)$ and the output of $\mathsf{US.Sim}$ is $\mathsf{params}' = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{Sampler}')$.

Theorem 5.8. *If* dO *and one-way functions exist, then there exists an universal sampler.*

Proof. First, it is straightforward that correctness is satisfied. Next we prove security. Fix a circuit C^* and suppose there is an adversary $\mathcal{A}$ for the security game with respect to C^*. We prove the indistinguishability through a sequence of hybrids:

- **Hyb 0:** Here, the adversary receives $\mathsf{params} \leftarrow \mathsf{US.Setup}(1^\kappa, 1^\ell, 1^t)$;
- **Hyb 1:** In this hybrid, let s^* be the instance sampled by $C^*(; \mathsf{DPRF.Eval}(S, C^*))$. We get $\mathsf{params}_1 \leftarrow \mathsf{US.Sim}(1^\kappa, 1^\ell, 1^t, C^*, s^*)$ where let $\mathsf{Sampler}_1$ be the circuit constructed in $\mathsf{US.Sim}$ and we are using the same seed S as in **Hyb 0**.

  It is straightforward that $\mathsf{Sampler}_1$ and $\mathsf{Sampler}$ are $\ell + 1$-decomposing equivalent. Therefore $\mathsf{params}_1 = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{Sampler}_1)$ and $\mathsf{params} = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{Sampler})$ are indistinguishable by dO security, meaning **Hyb 0** and **Hyb 1** are indistinguishable.
- **Hyb 2 :** This is the same as **Hyb 1**, except we replace the fragment in the circuit assignment $\mathsf{DecomposeTo}(\mathsf{DPRF.Eval}(S, \cdot), C^*)$ corresponding to C^* with the fragment "**return** $\mathsf{DPRF.Eval}(S', C^*)$" where $S' \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$ is a new key generated uniformly at random. We call the new circuit assignment L'. The indistinguishability between **Hyb 1** and **Hyb 2** follows from the DPRF security.
- **Hyb 3:** In this hybrid, since the fragment in L' corresponding to C^* is now returning $C^*(; \mathsf{DPRF.Eval}(S', C^*))$ and we don't have S' in the program, by PRF security, we can replace the return value with $C(; r^*)$ where r^* is a truly random string. This is equivalent to the adversary receiving $\mathsf{params} \leftarrow \mathsf{US.Sim}(1^\kappa, 1^\ell, 1^t, C^*, s^*)$ for a fresh sample $s^* \leftarrow C^*$.

□

5.4 Equivalence of dO and FE assuming Public Key Encryption

In this section we are going to prove the following theorem:

Theorem 5.9. *Assume public key encryption and* dO *exist, there exists compact (multi-key selective secure) functional encryption.*

Our construction is similar with [GS16]. Before mentioning the construction, wel first give two more definitions which are used in the construction.

5.4.1 Background

Definition 5.10 (Symmetric Key Encryption with Disjoint Ranges [LP09])**.** *A symmetric key encryption* DSKE *with disjoint ranges consists a tuple of algorithms* DSKE.KeyGen, DSKE.Enc, DSKE.Dec , DSKE.InRange *and satisfies every property below:*

- $\mathsf{DSKE.KeyGen}(1^\kappa)$ *is a probabilistic polynomial time algorithm that takes a security parameter* κ*, outputs a secret key* sk*;*
- $\mathsf{DSKE.Enc}(\mathsf{sk}, m)$ *is a polynomial time algorithm that takes a secret key* sk *and a message* $m \in \{0,1\}^*$*, outputs a ciphertext* c*;*
- $\mathsf{DSKE.Dec}(\mathsf{sk}, c)$ *is a polynomial time algorithm that takes a secret key* sk *and a ciphertext* $c \in \{0,1\}^*$*, outputs a message* m'*;*
- ***Range Disjoint :*** *Let* $\mathsf{Range}_n(\mathsf{sk})$ *denote the set* $\{\mathsf{DSKE.Enc}(\mathsf{sk}, x) \,|\, x \in \{0,1\}^n\}$*. For any two different secret key* $\mathsf{sk}_0 \neq \mathsf{sk}_1$*,* $\mathsf{Range}_n(\mathsf{sk}_0) \cap \mathsf{Range}_n(\mathsf{sk}_1) \neq \emptyset$ *with overwhelming probability, i.e., for any* $\mathsf{sk}_0 \neq \mathsf{sk}_1$*, there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{Range}_n(\mathsf{sk}_0) \cap \mathsf{Range}_n(\mathsf{sk}_1) \neq \emptyset\right] \geq 1 - \mathsf{negl}(\kappa)$$

- $\mathsf{DSKE.InRange}(\mathsf{sk}, c)$ *is an efficient algorithm that checks if a given ciphertext* c *is in* $\mathsf{Range}_n(\mathsf{sk})$*;*
- ***Correctness :*** *symmetric key encryption with disjoint ranges is said to be correct if for all* κ *and all message* $m \in \{0,1\}^*$*,*

$$\Pr\left[\mathsf{DSKE.Dec}(\mathsf{sk}, c) = m \,|\, \mathsf{sk} \leftarrow \mathsf{DSKE.KeyGen}(1^\kappa)\,;\, c \leftarrow \mathsf{DSKE.Enc}(\mathsf{sk}, m)\right] = 1$$

- ***Security :*** *It has* SKE *security.*

Symmetric key encryption with disjoint ranges can be obtained from one-way functions [LP09].

We now define a circuit garbling scheme from [Yao86]. We use the definition from [LP09].

Definition 5.11 (Garbled Circuits [Yao86, LP09])**.** *A circuit garbling scheme consists a tuple of PPT algorithms* (Garb.ParaGen, Garb.Eval) *satisfying the following properties:*

- $\mathsf{Garb.ParaGen}(C)$: *It is an efficient randomized procedure that takes a circuit defined on κ bits to be garbled and outputs a garbled parameter and the set of garbled input labels: $\widetilde{C}$ and $\{\mathsf{inp}_{i,b_i}\}_{i\in[\kappa],b_i\in\{0,1\}}$;*

- $\mathsf{Garb.Eval}(\widetilde{C},\{\mathsf{inp}_{i,x_i}\}_{i\in[\kappa]})$: *It is a deterministic algorithm that takes a parameter $\widetilde{C}$ and input labels $\{\mathsf{inp}_{i,x_i}\}$ corresponding to x, it outputs a string y;*

- ***Correctness :*** Garb *is said to be correct if for all circuits C and all inputs x, we have the following:*

$$\Pr\left[\mathsf{Garb.Eval}(\widetilde{C},\{\mathsf{inp}_{i,x_i}\}) = C(x) \,|\, \widetilde{C},\{\mathsf{inp}_{i,b_i}\}_{i\in[\kappa],b_i\in\{0,1\}} \leftarrow \mathsf{Garb.ParaGen}(C)\right] = 1$$

- ***Security :*** *There exists a simulator* Sim *such that for all circuits C and all input x,*

$$\left\{\widetilde{C},\{\mathsf{inp}_{i,x_i}\}_{i\in[\kappa]}\right\} \approx_c \{\mathsf{Sim}(1^\kappa, C, C(x))\}$$

In other words, for any poly sized adversary $\mathcal{A}$, there exists a negligible function negl *such that for any C and any input x,*

$$\left|\Pr\left[\mathcal{A}(\widetilde{C},\{\mathsf{inp}_{i,x_i}\}) = 1\right] - \Pr\left[\mathcal{A}(\mathsf{Sim}(1^\kappa, C, C(x))) = 1\right]\right| \leq \mathsf{negl}(\kappa)$$

Assuming the existence of one-way functions, there exists a circuit garbling scheme satisfying Definition 5.11 [Yao86, LP09].

5.4.2 dO and Public Key Encryption implies Compact FE

We will first prove dO implies single-key compact functional encryption. The proof of multi-key FE from dO is similar. Another way to get multi-key FE is to use the result of [GS16]. Now let us give the construction:

- $\mathsf{FE.Setup}(1^\kappa)$: it first randomly samples two keys $S \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$ and $K \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$; consider the following algorithm:

Algorithm 7 Algorithm G with S, K hardcoded

procedure $G(\mathsf{pk} = \mathsf{pk}_1\mathsf{pk}_2,\cdots\mathsf{pk}_\kappa, \mathsf{DPRF.Eval}(S,\mathsf{pk}), \mathsf{DPRF.Eval}(K,\mathsf{pk}))$
  $K_{\mathsf{pk}} \leftarrow \mathsf{DPRF.Eval}(K,\mathsf{pk})$;
  $S_{\mathsf{pk}} \leftarrow \mathsf{DPRF.Eval}(S,\mathsf{pk})$;
  return $\mathsf{PKE.Enc}(\mathsf{pk}, S_{\mathsf{pk}}; K_{\mathsf{pk}})$
end procedure

Algorithm G takes a public key pk for a functional encryption scheme and outputs the encryption of S_{pk} using the public key pk. It pads G to have a length U_1 which is a circuit size upper bound (it is a specific polynomial of $|G|$). Let $\mathsf{mpk} = \mathsf{dO.ParaGen}(1^\kappa, G(\cdot, \mathsf{DPRF.Eval}(S,\cdot), \mathsf{DPRF.Eval}(K,\cdot)))$ and $\mathsf{msk} = S$. Finally it outputs $(\mathsf{mpk},\mathsf{msk})$.

- $\mathsf{FE.Enc}(\mathsf{mpk}, m)$: it takes a message $m \in \{0,1\}^*$ and does the following (see Algorithm 8): it first generates a key pair $(\mathsf{pk}, \mathsf{sk})$ for a public key encryption scheme; then it gets the encryption of S_{pk} from G and decrypts it by using the secret key sk; finally it outputs pk and L_{i,m_i} for $1 \leq i \leq n$;

Algorithm 8 FE encryption algorithm

procedure $\mathsf{FE.Enc}(\mathsf{mpk}, m)$
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\kappa)$;
  $c \leftarrow \mathsf{dO.Eval}(\mathsf{mpk}, \mathsf{pk})$;
  $S'_{\mathsf{pk}} \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}, c)$;
  $\hat{S}'_{\mathsf{pk}} \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$ using the randomness from S'_{pk};
  for $i = 1, 2, \cdots, n$ **do**
    $L_{i,m_i} \leftarrow \mathsf{DPRF.Eval}(\hat{S}'_{\mathsf{pk}}, i||m_i)$; ▷ Here i is padded to have $\lceil \log_2 n \rceil$ bits before being feed to DPRF
  end for
  return $(\mathsf{pk}, \{L_{i,m_i}\}_{i\in[n]})$
end procedure

- $\mathsf{FE.KeyGen}(\mathsf{msk}, C_f)$: it takes a secret key $\mathsf{msk} = S$ and a circuit C_f describing the function f, it then constructs the following circuit (Algorithm 9) where $K' \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$ is a new key.

Algorithm 9 Algorithm H

procedure $H(\mathsf{pk} = \mathsf{pk}_1\mathsf{pk}_2\cdots\mathsf{pk}_\kappa, \mathsf{DPRF.Eval}(S, \mathsf{pk}), \mathsf{DPRF.Eval}(K', \mathsf{pk}))$
  $K'_{\mathsf{pk}} \leftarrow \mathsf{DPRF.Eval}(K', \mathsf{pk})$;
  $S_{\mathsf{pk}} \leftarrow \mathsf{DPRF.Eval}(S, \mathsf{pk})$;
  $\hat{S}_{\mathsf{pk}} \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa)$ using the randomness from S_{pk};
  $(\widetilde{C}_f, \{\mathsf{inp}_{i,b_i}\}_{i\in[\kappa],b_i\in\{0,1\}}) \leftarrow \mathsf{Garb.ParaGen}(C_f; K'_{\mathsf{pk}})$;
  $c_{i,b} = \mathsf{DSKE.Enc}(L_{i,b}, \mathsf{inp}_{i,b})$ for any $i \in [n]$ and $b \in \{0,1\}$ where $L_{i,b} = \mathsf{DPRF.Eval}(\hat{S}_{\mathsf{pk}}, i||b)$;
  return $(\widetilde{C}_f, \{c_{i,b_i}\}_{i\in[n],b_i\in\{0,1\}})$;
end procedure

  H simply applies $\mathsf{Garb.ParaGen}$ on C_f but encrypts all the input labels using L_{i,b_i}. This procedure outputs $\mathsf{dO.ParaGen}(1^\kappa, H(\cdot, \mathsf{DPRF.Eval}(S, \cdot), \mathsf{DPRF.Eval}(K', \cdot)))$ (H is padded to a length upper bound U_2).

- $\mathsf{FE.Dec}(\mathsf{fsk}_f, c)$ where $\mathsf{fsk}_f = \mathsf{dO.ParaGen}(1^\kappa, H)$ and $c = (\mathsf{pk}, \{L_{i,m_i}\}_{i\in[n]})$.

  It will first get $(\widetilde{C}_f, \{c_{i,b_i}\}_{i\in[n],b_i\in\{0,1\}})$ by $\mathsf{dO.Eval}(\mathsf{fsk}_f, \mathsf{pk})$. Since either $c_{i,0}$ or $c_{i,1}$ will be in $\mathsf{Range}_n(L_{i,m_i})$, it can use $\mathsf{DSKE.InRange}(L_{i,m_i}, c_{i,b})$ to know which ciphertext to

choose. So it can exactly decrypt c_{i,m_i} using L_{i,m_i} to get inp_{i,m_i} for $1 \le i \le n$. Finally it runs $\mathsf{Garb.Eval}(\widetilde{C}, \{\mathsf{inp}_{i,m_i}\}) = C_f(m)$.

Correctness and Efficiency

It is easy to see that the above construction satisfies the correctness of functional encryption. Since given a fsk_f and $c = (\mathsf{pk}, \{L_{i,m_i}\}_{i\in[n]})$, $\mathsf{FE.Dec}$ will find the right c_{i,m_i} by $\mathsf{DSKE.InRange}$. Given the right $\{\mathsf{inp}_{i,m_i}\}$, by correctness of the circuit garbling scheme, $\mathsf{Garb.Eval}(\widetilde{C}, \{\mathsf{inp}_{i,m_i}\}) = C_f(m)$,

Let us argue it is compact. The running time of $\mathsf{FE.Enc}$ is bounded by $\mathsf{poly}(\kappa, |m|)$ since the running time of $\mathsf{PKE.KeyGen}$, $\mathsf{dO.Eval}$, $\mathsf{PKE.Dec}$, DPRF and the output size of $\mathsf{dO.ParaGen}$ is bounded by $\mathsf{poly}(\kappa, |m|)$. It is independent of C_f.

Security

Now let us prove the security through a sequence of hybrids.

- **Hyb 0**: The adversary is given the following:
  - a master public key $\mathsf{mpk} = \mathsf{dO.ParaGen}(1^\kappa, G(\cdot, \mathsf{DPRF.Eval}(S, \cdot), \mathsf{DPRF.Eval}(K, \cdot)))$,
  - a ciphertext $c^* = (\mathsf{pk}^*, \{L_{i,m_{0,i}}\}_{i\in[n]})$,
  - a function key $\mathsf{fsk}_f = \mathsf{dO.ParaGen}(1^\kappa, H(\cdot, \mathsf{DPRF.Eval}(S, \cdot), \mathsf{DPRF.Eval}(K', \cdot)))$.
- **Hyb 1**: In this hybrid, we decompose the program G to get a circuit assignment $L_G = \mathsf{DecomposeTo}(G, \mathsf{pk}^*)$. By lemma 5.3, indeed we have

  $$L_G = G(\cdot, \mathsf{DecomposeTo}(\mathsf{DPRF.Eval}(S, \cdot), \mathsf{pk}^*), \mathsf{DecomposeTo}(\mathsf{DPRF.Eval}(K, \cdot), \mathsf{pk}^*))$$

  Then we get its corresponding circuit $\mathsf{CanonicalMerge}(L_G)$ (pad it to length U_1) and a new master public key $\mathsf{mpk}_1 = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L_G))$. The indistinguishability comes from the security of dO.
- **Hyb 2**: In this hybrid, we decompose H along pk^* to get a circuit assignment $L_H = \mathsf{DecomposeTo}(H, \mathsf{pk}^*)$. By lemma 5.3, indeed we have

  $$L_H = H(\cdot, \mathsf{DecomposeTo}(\mathsf{DPRF.Eval}(S, \cdot), \mathsf{pk}^*), \mathsf{DecomposeTo}(\mathsf{DPRF.Eval}(K', \cdot), \mathsf{pk}^*))$$

  Then we get its corresponding circuit $\mathsf{CanonicalMerge}(L_H)$ (pad it to length U_2) and a new function key $\mathsf{fsk}_1 = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L_H))$. The indistinguishability comes from the security of dO.
- **Hyb 3**: In this hybrid, we are going to change the fragment in L_G corresponding to pk^*. The fragment is "**return** $\mathsf{PKE.Enc}(\mathsf{pk}^*, S_{\mathsf{pk}^*}; K_{\mathsf{pk}^*})$". By DPRF security, K_{pk^*} can be replaced as $\mathsf{DPRF.Eval}(K'', \mathsf{pk}^*)$ where K'' is a new key generated using a uniformly

random string. And as we don't have K'' inside the program (the fragment is simplified), by PRF security, K_{pk^*} can again be replaced with a uniformly random string r_1.

So in this step, the fragment now becomes "**return** $\mathsf{PKE.Enc}(\mathsf{pk}^*, S_{\mathsf{pk}^*}; r_1)$". Let us call the new circuit assignment L'_G. Finally we get $\mathsf{mpk}_2 = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L'_G))$. The indistinguishability comes from PRF and DPRF security.

- **Hyb 4**: In this hybrid, we are still going to change the fragment in L'_G corresponding to pk^*. It is now "**return** $\mathsf{PKE.Enc}(\mathsf{pk}^*, \bot; r_1)$" where $\bot$ means filling it with zeroes. Let us call the new circuit assignment L''_G. The indistinguishability comes from the security of PKE since we only have pk^* but don't have sk^*. (Observe that L''_G no longer has S_{pk^*} inside.)

- **Hyb 5**: In this hybrid, we can replace K'_{pk^*} using the same way as in **Hyb 3** and replace it with a uniformly random string r_2. So the fragment in the new circuit assignment L'_H corresponding to pk^* is now using r_2 to compute $\mathsf{Garb.ParaGen}(C_f)$ instead of the old K'_{pk^*}. The indistinguishability comes from the security of DPRF.

- **Hyb 6**: By the DPRF security, we can replace the fragment $\mathsf{DPRF.Eval}(S, \mathsf{pk}^*)$ in $\mathsf{DecomposeTo}(\mathsf{DPRF.Eval}(S, \cdot))$ with $\mathsf{DPRF.Eval}(\widetilde{S}, \mathsf{pk}^*)$ where $\widetilde{S}$ is a newly generated key using fresh randomness. We find that $\mathsf{DPRF.Eval}(S, \mathsf{pk}^*)$ does not appear in L''_G as the fragment corresponding to pk^* is now "**return** $\mathsf{PKE.Enc}(\mathsf{pk}^*, \bot; r_1)$". And the fragment in L'_H has $\mathsf{DPRF.Eval}(\widetilde{S}, \mathsf{pk}^*)$ but not that of S.

- **Hyb 7**: We can now replace the $\mathsf{DPRF.Eval}(\widetilde{S}, \mathsf{pk}^*)$ in the fragment corresponding to pk^* in L'_H with a truly random string r_3. The indistinguishability comes from PRF security. Now $\hat{S}_{\mathsf{pk}^*}$ in G can be viewed as being generated by fresh randomness, in other words, $\hat{S}_{\mathsf{pk}^*} \leftarrow \mathsf{DPRF.KeyGen}(1^\kappa; r_3)$.

- **Hyb 8**: We replace each L_{i,b_i} with truly random strings. We have the following observations:

  - $\hat{S}_{\mathsf{pk}^*}$ does not appear anywhere now;
  - We can replace each L_{i,b_i} for all $i \in [n]$ and $b_i \in \{0,1\}$ one by one, by DPRF security and PRF security, and replace it with truly random strings r_{i,b_i}. If L_{i,b_i} is part of c^*, we also replace it in the ciphertext c^*. Otherwise, we only need to replace it in the fragment.

    That is, we change the secret key (to encrypt inp_{i,b_i}) from L_{i,b_i} to a uniformly random string, and update c^*_{i,b_i}.
  - After all these replacement, the fragment in the new L''_H corresponding to pk^* now becomes "**return** $(\widetilde{C_f}, \{c^*_{i,b_i}\}_{i\in[n],b_i\in\{0,1\}})$" where c^*_{i,b_i} are ciphertexts using the new symmetric keys.

- **Hyb 9**: Since in the previous hybrid, the adversary no longer has $L_{i,\neg m_{0,i}}$ (which is replaced with truly randomness and the circuit is simplified), we can replace any $c_{i,\neg m_{0,i}}$

as $\mathsf{SKE.Enc}(r_{i,\neg m_{0,i}}, \bot)$ instead of $\mathsf{SKE.Enc}(r_{i,\neg m_{0,i}}, \mathsf{inp}_{i,\neg m_{0,i}})$ where the symmetric key $r_{i,\neg m_{0,i}}$ is drawn uniformly at random. It comes from indistinguishability of SKE.

So we have $c^* = (\mathsf{pk}^*, \{r_{i,m_{0,i}}\}_{i\in[n]})$, $\mathsf{mpk} = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L''_G))$ and $\mathsf{fsk}_f = \mathsf{dO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L''_H))$ where

$$\begin{aligned}&\mathsf{dO.Eval}(\mathsf{fsk}_f, \mathsf{pk}^*)\\ &= (\widetilde{C_f}, \{c_{i,m_{0,i}} = \mathsf{SKE.Enc}(r_{i,m_{0,i}}, \mathsf{inp}_{i,m_{0,i}})\}_{i\in[n]} \cup \{c_{i,\neg m_{0,i}} = \mathsf{SKE.Enc}(r_{i,\neg m_{0,i}}, \bot\}_{i\in[n]})\end{aligned}$$

- **Hyb 10**: In this Hyb, we replace $\{\widetilde{C_f}, \{\mathsf{inp}_{i,m_{0,i}}\}_{i\in[n]}\}$ with $\{\mathsf{Sim}(1^\kappa, C, C(m_0))\}$. The indistinguishability comes from the security of a circuit garbling scheme.
- **Hyb 11**: In this Hyb, we replace $\{\mathsf{Sim}(1^\kappa, C, C(m_0))\}$ with $\{\mathsf{Sim}(1^\kappa, C, C(m_1))\}$ since $C(m_0) = C(m_1)$. The two distributions are identical.

If it starts with $c^* = (\mathsf{pk}^*, \{L_{i,m_{1,i}}\}_{i\in[n]})$, it will finally goes to Hyb 9 through several hybrids. So an adversary can not distinguish whether it is given the encryption of m_0 or m_1 when the function query C_f satisfies $C_f(m_0) = C_f(m_1)$. We complete the proof of security.

Theorem 5.12. *Assume one-way functions and* dO *exist, there exists compact (multi-key selective secure) secret functional encryption.*

We don't give the full proof for this theorem. But it is quite similar to the proof for theorem 5.9. And it is even simpler because the challenger does not need to give mpk to the adversary which allows us to get rid of PKE in the theorem.

5.5 PPAD Hardness from polynomially hard dO

In this section, we will first mention the background and notations and then give the main result.

5.5.1 Background

Most of this subsection are taken verbatim from [BPR15, GPS16]. A search problem is given by a tuple (I, R). I defines the set of instances and R is an NP relation. Given $x \in I$, the goal is to find a witness w (if it exists) such that $R(x, w) = 1$. We say a search problem (I_1, R_1) is polynomial time reducible to another research problem (I_2, R_2) if there exist polynomial time algorithms P, Q such that for every $x_1 \in I_1$, $P(x_1) \in I_2$ and given w_2 such that $R_1(P(x_1), w_2) = 1$, we have $R_1(x_1, Q(w_2)) = 1$.

A search problem is said to be total if for any $x \in \{0,1\}^*$, there exists a polynomial time procedure to test whether $R(x, w) = 1$ for given x, w and for all $x \in I$, the set of witness w such that $R(x, w) = 1$ is non-empty. The class of total search problems is denoted by TFNP. PPAD [Pap94] is a subset of TFNP and is defined by its complete problem called $\mathsf{END\text{-}OF\text{-}LINE}$ (abbreviated as EOL).

Definition 5.13. END-OF-LINE *is the following problem:* $\mathsf{EOL} = \{I_{\mathsf{EOL}}, R_{\mathsf{EOL}}\}$ *where* $I_{\mathsf{EOL}} = \{(x_s, \mathsf{Succ}, \mathsf{Pred}) : \mathsf{Succ}(x_s) \neq x_s = \mathsf{Pred}(x_s)\}$ *and* $R_{\mathsf{EOL}}((x_s, \mathsf{Succ}, \mathsf{Pred}), w) = 1$ *if and only if* $(\mathsf{Pred}(\mathsf{Succ}(w)) \neq w) \vee (\mathsf{Succ}(\mathsf{Pred}(w)) \neq w \wedge w \neq x_s)$.

Definition 5.14. *The complexity class* PPAD *is the set of all search problems* (I, R) *such that* $(I, R) \in \mathsf{TFNP}$ *and* (I, R) *polynomial time reduces to* EOL.

Now let us look at a related problem to EOL which is SINK-OF-VERFIABLE-LINE (abbreviated as SVL) which is defined as follows:

Definition 5.15. $\mathsf{SVL} = \{I_{\mathsf{SVL}}, R_{\mathsf{SVL}}\}$ *where* $I_{\mathsf{SVL}} = \{(x_s, \mathsf{Succ}, \mathsf{Ver}, T)\}$ *is the set of instances and the relation* $R_{\mathsf{SVL}}((x_s, \mathsf{Succ}, \mathsf{Ver}, T), w)$ *is true* $\iff \mathsf{Ver}(w, T) = 1$.

SVL instance defines a single directed path with the source being x_s. Succ is the successor circuit and there is a directed edge between u and v if and only if $\mathsf{Succ}(u) = v$. Ver is the verification circuit and is used to test whether a given node is the i-th node from x_s. That is, $\mathsf{Ver}(x, i) = 1$ iff $x = \mathsf{Succ}^{i-1}(x_s)$. The goal is to find the T-th node in the path. We have the following lemma from [AKV04, BPR15].

Lemma 5.16. SVL *polynomial time reduces to* EOL.

So to prove the existence of dO implies PPAD hardness, we only need to show dO implies SVL hardness.

5.5.2 Construction

The construction is similar to [GPS16].

Given the Next function (see algorithm 11), we can now construct a SVL instance $(x_s, \mathsf{Succ}, \mathsf{Ver}, T)$ where

- $x_s = (0^\kappa, \mathsf{DPRF.Eval}(S_1, 0), \cdots, \mathsf{DPRF.Eval}(S_\kappa, 0^\kappa))$;
- $\mathsf{Succ}(k, \sigma_1, \cdots, \sigma_\kappa) = \mathsf{Next}(k, \sigma_1, \cdots, \sigma_\kappa)$;
- $\mathsf{Ver}(x, k) = 1$ iff $\mathsf{Succ}^{k-1}(x_s) = x$;
- $T = 1^\kappa$;

where $S_1, S_2, \cdots, S_\kappa$ are sampled from $\mathsf{DPRF.KeyGen}(1^\kappa)$. And PRG is a pseudorandom generator with expansion factor 4 where PRG_0 denotes the left part of PRG's output and PRG_1 denotes the right part. And $v \xleftarrow[R]{} \{0,1\}^{4\kappa}$. For simplicity, we define $P_1(x_1) = \mathsf{DPRF.Eval}(S_1, x_1)$ defined on inputs of length 1, $P_2(x_{[2]}) = \mathsf{DPRF.Eval}(S_2, x_{[2]})$ defined on inputs of length 2, and similarly $P_\kappa(x) = \mathsf{DPRF.Eval}(S_\kappa, x)$ defined on inputs of length κ. So x_s can be written as $(0^\kappa, P_1(0), P_2(00), \cdots, P_\kappa(0^\kappa))$. Also we can define $Q_i(x)$ computes $P_i(x+1)$; it means Q_i

Algorithm 10 G outputs a sequence

procedure $G(x = x_1 x_2 \cdots x_\kappa, P_1(x_1), Q_1(x_1), \cdots, P_\kappa(x), Q_\kappa(x))$
  Hardcoded: v
  initialize arrays $\{t_i\}, \{\alpha_i\}, \{\gamma_i\}$ as zeroes for $i = 1 \cdots \kappa$;
  initialize j as 1 (j will finally be the smallest integer, $x = x_{[j]}||1^{\kappa-j}$);
  for $i = 1, 2, \cdots, \kappa$ **do**
    if $x_i = 0$ **then**
      fill γ_l and t_l with zeroes for all l between j and $i-1$
      update j as i since $f(x)$ will be at least i
    end if
    compute $s_i = \mathsf{DPRF.Eval}(S_i, x_{[i]})$ by $P_i(x_{[i]})$
    compute and store $\alpha_i = \mathsf{PRG}_0(s_i)$ and $\gamma_i = \mathsf{PRG}_1(s_i)$
    compute and store $t_i = \mathsf{DPRF.Eval}(S_i, x_{[i]} + 1)$ by $Q_i(x_{[i]})$
  end for
  for $i = j, j+1, \cdots, \kappa$ **do**
    compute $\beta_i = \mathsf{SKE.Enc}(\gamma_j|| \cdots ||\gamma_\kappa, t_i)$
  end for
  if $\mathsf{PRG}(x) = v$ **then**
    return $\perp$
  end if
  return $\alpha_1, \cdots, \alpha_\kappa, \beta_j, \cdots, \beta_\kappa$
end procedure

Algorithm 11 Next computes the next feasible node

procedure NEXT($x = x_1 x_2 \cdots x_\kappa, \sigma_1, \cdots, \sigma_\kappa$)
  Hardcoded : $G' \leftarrow \mathsf{dO.ParaGen}(1^\kappa, G(\cdot, P_1(\cdot), Q_1(\cdot), \cdots, P_\kappa(\cdot), Q_\kappa(\cdot)))$
  $(\alpha_1, \cdots, \alpha_\kappa, \beta_j, \cdots, \beta_\kappa) \leftarrow \mathsf{dO.Eval}(G', x_1 x_2 \cdots x_\kappa)$
  if the output is $\perp$ or $\mathsf{PRG}_0(\sigma_i) \neq \alpha_i$ for any $i \in [\kappa]$ **then**
    output $\perp$
  end if
  if $x = 1^\kappa$, output **SOLVED**
  compute $j = f(x)$ which is the smallest integer, $x = x_{[j]}||1^{\kappa-j}$
  for $i = 1, \cdots, j-1$ **do**
    $\sigma'_i \leftarrow \sigma_i$
  end for
  for $i = j, \cdots, \kappa$ **do**
    $\gamma_i \leftarrow \mathsf{PRG}_1(\sigma_i)$
  end for
  for $i = j, \cdots, \kappa$ **do**
    $\sigma'_i \leftarrow \mathsf{SKE.Dec}(\gamma_j|| \cdots ||\gamma_\kappa, \beta_i)$
  end for
  return $(x+1, \sigma'_1, \cdots, \sigma'_\kappa)$
end procedure

Algorithm 12 Q_m computes $P_m(x+1) = \mathsf{DPRF.Eval}(S_m, x+1)$

procedure $Q_m(x = x_1 x_2 \cdots x_m)$
  let P be the fragment $P_m(\cdot)$
  let $\mathsf{tmp} = P_m(0^m)$
  for $i = 1, 2, \cdots, m$ **do**
    if $x_i = 0$ **then**
      $\mathsf{tmp} \leftarrow P(1||0^{m-i+1})$
    end if
    update fragment $P \leftarrow P(x_i, \cdot)$
  end for
  return tmp
end procedure

first computes $y = x + 1$ and then gets $P_i(y)$ (P_i, Q_i can also be viewed as defined on inputs of length κ by simply ignoring the last few input bits.)

This algorithm Next describes a line graph where the starting node is x_s and Next will first check the given input is valid and then return the next node in the graph.

Correctness

First, when $S_1, S_2, \cdots, S_\kappa$ are sampled from $\mathsf{DPRF.KeyGen}(1^\kappa)$ and $v \xleftarrow{R} \{0,1\}^{4\kappa}$, with overwhelming probability $1 - \frac{1}{2^{3\kappa}}$, we have a valid instance of SVL. Since with overwhelming probability,

$$\Pr[\exists x \text{ such that } \mathsf{PRG}(x) = v] \leq \bigcup_x \Pr[\mathsf{PRG}(x) = v] = \frac{2^\kappa}{2^{4\kappa}} = 1 \Big/ 2^{3\kappa}$$

the 17-th line of G will never be executed. With a node description $(x, \sigma_1, \cdots, \sigma_\kappa)$, we first compute a list $\alpha_1, \alpha_2, \cdots, \alpha_\kappa, \beta_j, \cdots, \beta_\kappa$ where $j = f(x)$. And in Next, we check that for all i, $\mathsf{PRG}(\sigma_i) = \alpha_i$ holds. Because x and $x+1$ share the longest common prefix with length $j-1$ (in details $x = x_{[j-1]}||0||1^{\kappa-j}$ and $x+1 = x_{[j-1]}||1||0^{\kappa-j}$) it is easy to see that $\mathsf{DPRF.Eval}(S_i, x_{[i]}) = \mathsf{DPRF.Eval}(S_i, (x+1)_{[i]})$ for $1 \leq i \leq j-1$. Then we decrypt $\beta_j, \cdots, \beta_\kappa$ to get $\mathsf{DPRF.Eval}(S_i, (x+1)_{[i]}) = \mathsf{DPRF.Eval}(S_i, x_{[i]} + 1) = Q_i(x_{[i]})$ for all $j \leq i \leq \kappa$.

Security

Proof. Now let us prove PPAD hardness. We are doing the proof through polynomial number of hybrids to the function G. In the following hybrids, before using dO, we will first pad the programs to length U where U is the length upper bound of all the programs which is of $\mathsf{poly}(\kappa, |G|)$.

- **Hyb 0:** The adversary is given an SVL instance in the above construction where every S_i is sampled by $\mathsf{DPRF.KeyGen}(1^\kappa)$. The adversary has x_s and G.
- **Hyb 1:** In this hybrid, we change the hardcoded v in G by v' where $v' \leftarrow \mathsf{PRG}(u)$ and $u \xleftarrow{R} \{0,1\}^\kappa$. It is easy to show the indistinguishability from the security of the pseudorandom generator. We denote the algorithm as G_1. Given u, we define a sequence $u = u_0, u_1, \cdots, u_\delta = 1^\kappa$ where δ is at most κ, $u_{i+1} = u_i + 2^{\kappa - f(u_i)}$ and $f(x)$ is the smallest integer j such that $x = x_{[j]}||1^{\kappa-j}$.
- **Hyb 2:** In this hybrid, we can decompose the program G along u. Let the circuit assignment $L = \mathsf{DecomposeTo}(G, u)$. The fragment in L corresponding to u is "**return** $\perp$". Let $G_2 = \mathsf{CanonicalMerge}(L)$. We find $\mathsf{dO.ParaGen}(1^\kappa, G_2)$ and $\mathsf{dO.ParaGen}(1^\kappa, G_1)$ are indistinguishable since G_1 and G_2 are $(\kappa+1)$-decomposing equivalent.
- **Hyb 3** In this hybrid, we will decompose the program into more pieces and replace some fragments. After the replacement, the two programs are indistinguishable but with overwhelming probability, for any $x \in [u_0 + 1, u_1]$, there does not exist $\sigma_1, \cdots, \sigma_\kappa$ that pass the test $\mathsf{PRG}_0(\sigma_i) = \alpha_i$ in Next function. Now let us look at the details:

– **Hyb 3.1** In this sub-hybrid, we puncture at $[u_0+1, u_1]$. We realize that for all $x \in [u_0+1, u_1]$, they share a common prefix of length $f_0 = f(u_0)$ (recall the definition of f in both algorithm 10 and **Hyb 1**). Let t_0 be the longest common prefix of u_0+1 and u_1, $f_0 = |t_0|$.

  We decompose our original program G into pieces along the path $u = u_0$ and u_1 to get $L_2 = \mathsf{DecomposeTo}(G, \{u_0, u_1\})$. And let $G_3 = \mathsf{CanonicalMerge}(L_2)$. We claim that $\mathsf{dO.ParaGen}(G_3)$ is indistinguishable from $\mathsf{dO.ParaGen}(G_2)$ because G_2 and G_3 are $(2\kappa+1)$-decomposing equivalent.

– **Hyb 3.2** For each fragment in L_2 which corresponds to y, the fragment is

$$G(y, \cdot, P_1(y,\cdot), Q_1(y,\cdot), \cdots, P_\kappa(y,\cdot), Q_\kappa(y,\cdot)) \text{ simplified}$$

  Here we view P_m, Q_m as functions defined on κ bit strings, so the above holds because of Lemma 5.3. For any Q_m and string y ($|y| \le m$), $Q_m(y,\cdot)$ can be constructed from $P_m(y,\cdot)$ and $P_m(y+1, 0^{m-|y|})$ since we can easily reconstruct Q_m in the $|y|$-th round (recall the definition of Q_m) using the fragment $P_m(y,\cdot)$ and the value $P_m(y+1, 0^{m-|y|})$ as tmp. Assume $y = y'||0||1^l$, in the $|y'|+1$ round, the algorithm Q_m will update tmp in that round by $P_m(y'||1||0^l, 0^{m-|y|}) = P_m(y+1, 0^{m-|y|})$ and never update tmp during $(|y'|+2)$-th round to $|y|$-th round.

  The above argument implies another observation: for any tree covering TC, $\mathsf{DecomposeTo}(Q_m, TC)$ can be constructed from $\mathsf{DecomposeTo}(P_m, TC)$. So now we only care about the circuit assignments $\mathsf{DecomposeTo}(P_m, \{x_1, x_2\})$.

– **Hyb 3.3** In this sub-hybrid, we replace the fragment in $\mathsf{DecomposeTo}(P_{f_0}, \cdot)$ corresponding to t_0 (or in other words, the value $\mathsf{DPRF.Eval}(S_{f_0}, t_0)$) with a uniformly random string r_{f_0,t_0}. The indistinguishability comes from DPRF and PRF security. Also we find that r_{f_0,t_0} only appears in the fragment (of the circuit Next) as $\mathsf{PRG}_0(r_{f_0,t_0}), \mathsf{PRG}_1(r_{f_0,t_0})$:

  * For any fragment corresponding to $y = u_{0,[l-1]}\neg u_{0,l}$ that $l < f_0$, there does not exist z such that $y||z = t_0$ or $t_0 - 1$, so the value r_{f_0,t_0} does not appear in this fragment;
  * For any fragment corresponding to $y = u_{0,[l-1]}\neg u_{0,l}$ that $l > f_0$, we know that $y_l = 0$. So by the definition of f function, for any z, $f(y||z)$ will be at least $l > f_0$. So the variable t_{f_0} in the function G has been replaced with zeroes and $\alpha_{f_0}, \beta_{f_0}$ are computed from $\mathsf{PRG}(\mathsf{DPRF.Eval}(S_{f_0}, t_{0,[f_0-1]}||0))$ instead of from $\mathsf{PRG}(\mathsf{DPRF.Eval}(S_{f_0}, t_0))$ as we know the f_0-th bit of t_0 is 1.
  * For the fragment corresponding to u_0, it already becomes "**return** $\perp$". (Here for **Hyb (2+i).3** where $i > 1$, the fragment corresponding to u_{i-1} is not "**return** $\perp$". It is "**return** $\{\alpha\}$ and $\{\beta\}$" where $\{\beta\}$ have been replaced with encryptions of random strings. So r_{f_i,t_i} is still not in the fragment.)
  * For any fragment corresponding to $y = u_{1,[l-1]}\neg u_{1,l}$ that $l < f_0$, there does not exist z such that $y||z = t_0$;

* For any fragment corresponding to $y = u_{1,[l-1]}\neg u_{1,l}$ that $l > f_0$, we know that $y_l = 0$. So by the definition of f function, for any z, $f(y||z)$ will be at least $l > f_0$. So it only has $\mathsf{PRG}_0(r_{f_0,t_0})$ inside the program.
* For the fragment corresponding to u_1, it has both $\mathsf{PRG}_0(r_{f_0,t_0}), \mathsf{PRG}_1(r_{f_0,t_0})$.

– **Hyb 3.4** In this sub-hybrid, we replace $\mathsf{PRG}(r_{f_0,t_0}) = \mathsf{PRG}_0(r_{f_0,t_0})||\mathsf{PRG}_1(r_{f_0,t_0})$ with truly random strings. Since we don't have r_{f_0,t_0} hardcoded in the circuit assignment, we can simply replace $\mathsf{PRG}_0(r_{f_0,t_0})$ and $\mathsf{PRG}_1(r_{f_0,t_0})$ with truly randomness $v_{0,0}, v_{0,1} \leftarrow \{0,1\}^{2\kappa}$ by the PRG security.

After the replacement, with overwhelming probability $(1 - 1/2^\kappa)$, there does not exist any σ_{f_0} such that $\mathsf{PRG}_0(\sigma_{f_0}) = v_{0,0}$. And the fragment corresponding to u_1 is returning $\{\alpha\}, \{\beta\}$ where $\{\beta\}$ are now encrypted by random keys because $\mathsf{PRG}_1(r_{f_0,t_0})$ has been replaced with a truly random string.

– **Hyb 3.5** In this sub-hybrid, as we no longer have the secret key for encrypting $\{\beta\}$, we can replace them with encryptions of random strings. The indistinguishability comes from SKE security.

- **Hyb (2 + i) for** $1 \leq i \leq \delta$ **:** In this hybrid, we will decompose the program into pieces along the path $u_0, u_1, u_2, \cdots, u_i$ and replace hardcoded values like **Hyb 3**. After the replacement, the two programs are indistinguishable but with overwhelming probability, for any $x \in [u_0, u_{2+i}]$, there does not exist $\sigma_1, \cdots, \sigma_\kappa$ that pass the test $\mathsf{PRG}_0(\sigma_i) = \alpha_i$ in Next function. The proof is similar like that for Hyb 3. Let G_{2+i} denote the current program we have.

Let $U = \max_{i=1}^{2+\delta} |G_i| \leq \mathsf{poly}(|G|, \kappa, \delta)$ be the upper bound of all the programs in the above hybrids.

Finally we are in Hyb $(2 + \delta)$. We have already replaced $\mathsf{PRG}_0(P_{f_i}(t_i))$ with true randomness.

$$\Pr\left[\exists \sigma_1, \cdots, \sigma_\kappa, \text{ such that } G_{2+\delta}(1^\kappa, \sigma_1, \cdots, \sigma_\kappa) \neq \bot\right] \leq \frac{1}{2^\kappa}$$

So with overwhelming probability, we can never find a valid signature (or witness $w = (\sigma_1, \sigma_2, \cdots, \sigma_\kappa)$) for the destination $T = 1^\kappa$ such that $\mathsf{Ver}(w, T) = 1$. □

Theorem 5.17. *If polynomially hard* dO *and one way functions exist, then the* $\mathsf{END\text{-}OF\text{-}LINE}$ *problem is hard for polynomial-time algorithms.*

5.6 Trapdoor Permutations

In this section, we present the construction of trapdoor permutations from dO. The construction is inspired from [GPSZ16].

5.6.1 Background

Most of this subsection are taken verbatim from [GPSZ16].

Definition 5.18. *An efficiently computable family of functions:*

$$\mathcal{TDP} = \{\mathsf{TDP}_{\mathsf{PK}} : D_{\mathsf{PK}} \to D_{\mathsf{PK}}\}$$

over the domain D_{PK} with associated probabilistic algorithms (KeyGen, $\mathsf{SampGen}$*) is a weakly samplable trapdoor permutation if it satisfies:*

- ***Trapdoor Invertibility:*** *For any $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, $\mathsf{TDP}_{\mathsf{PK}}$ is a permutation over D_{PK}. And for any $y \in D_{\mathsf{PK}}$, $\mathsf{TDP}^{-1}_{\mathsf{PK}}$ is efficiently computable gvien the trapdoor SK;*
- ***Weakly Pseudorandom Sampling:*** *For any $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and $\mathsf{Sampler} \leftarrow \mathsf{SampGen}(\mathsf{SK})$, $\mathsf{Sampler}(\cdot)$ samples pseudo random points in D_{PK}. Formally, for any poly sized adversary $\mathcal{A}$, we define the following game: $\mathsf{Game}_{\lambda,\mathcal{A},b}$:*
  - *The challenger prepares $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and $\mathsf{Sampler} \leftarrow \mathsf{SampGen}(\mathsf{SK})$.*
  - *If $b = 0$, the challenger sends $\langle \mathsf{PK}, \mathsf{Sampler}, x\rangle$ where x is a uniformly random sample in D_{PK}; otherwise, the challengers sends $\langle \mathsf{PK}, \mathsf{Sampler}, x'\rangle$ where x' is sampled by $\mathsf{Sampler}(\cdot)$.*
  - *The result of the game is the output of $\mathcal{A}$.*

  And the following holds: for any poly sized adversary $\mathcal{A}$, there exists a negligible function negl such that for every security parameter λ,

$$|\Pr[\mathsf{Game}_{\lambda,\mathcal{A},0} = 0] - \Pr[\mathsf{Game}_{\lambda,\mathcal{A},1} = 0]| \leq \mathsf{negl}(\lambda)$$

- ***One-wayness:*** *For all poly sized adversary $\mathcal{A}$, there exists a negligible function negl, for all λ,*

$$\Pr\left[\mathcal{A}(\mathsf{PK}, \mathsf{Sampler}, \mathsf{TDP}_{\mathsf{PK}}(x)) = x \middle| \begin{smallmatrix}(\mathsf{PK},\mathsf{SK})\leftarrow\mathsf{KeyGen}(1^\lambda)\\ \mathsf{Sampler}\leftarrow\mathsf{SampGen}(\mathsf{SK})\\ x\leftarrow\mathsf{Sampler}(\cdot)\end{smallmatrix}\right] \leq \mathsf{negl}(\lambda)$$

This is called "weakly pseudorandom sampling" because in $\mathsf{Game}_{\lambda,\mathcal{A},b}$, the challenger does not need to provide the randomness that is used in KeyGen and $\mathsf{SampGen}$. Therefore the trapdoor permutations can be only used in applications where an honest party runs KeyGen and $\mathsf{SampGen}$ algorithms.

5.6.2 Construction

First KeyGen is the following algorithm: it takes a security parameter λ and prepares 2λ keys $S_i \leftarrow \mathsf{DPRF.KeyGen}(1^\lambda)$ for $i = 1, 2, \cdots, 2\lambda$. Given these 2λ keys, the domain D_{PK} is defined as the following:

$$D_{\mathsf{PK}} = \left\{\left(x, \mathsf{DPRF.Eval}(S_1, x_{[1]}), \cdots, \mathsf{DPRF.Eval}(S_{2\lambda}, x_{[2\lambda]})\right)\right\}_{x \in \{0,1\}^{2\lambda}}$$

In other words, for every $x \in \{0,1\}^{2\lambda}$, it computes σ_i which is $\mathsf{DPRF.Eval}(S_i, x_{[i]})$. Similar Section 5.5, all the elements in D_{PK} define a cycle. For every element $(x, \cdots)$, it has a predecessor $((x-1) \bmod 2^{2\lambda}, \cdots)$ and a successor $((x+1) \bmod 2^{2\lambda}, \cdots)$.

Intuitively PK is an obfuscated program which takes as input an element corresponding to x in D_{PK} and outputs an element corresponding to $(x+1) \bmod 2^{2\lambda}$. In other words, PK takes an element in D_{PK} and outputs the next element in the cycle. SK is just 2λ keys $(S_1, \cdots, S_{2\lambda})$ and one can easily compute the previous element in the cycle with SK.

More formally, PK is the following program Next where Next (see algorithm 13) computes the next node in the cycle. This is almost the same as that in Section 5.5 but it does not halt on $1^{2\lambda}$ or return **SOLVED**. PRG_0 and PRG_1 are the left and right part of a pseudorandom generator $\mathsf{PRG} : \{0,1\}^\lambda \to \{0,1\}^{4\lambda}$. P_i is a circuit computing $\mathsf{DPRF.Eval}(S_i, \cdot)$ and Q_i is a circuit computing $Q_i(x) = P_i((x+1) \pmod{2^i})$ just in Section 5.5

Algorithm 13 Next computes the next node in the cycle

1: **procedure** NEXT($x = x_1 x_2 \cdots x_{2\lambda}, \sigma_1, \cdots, \sigma_{2\lambda}$)
2: **Hardcoded :** $G' \leftarrow \mathsf{dO.ParaGen}(1^\kappa, G(\cdot, P_1(\cdot), Q_1(\cdot), \cdots, P_{2\lambda}(\cdot), Q_{2\lambda}(\cdot)))$
3: $(\alpha_1, \cdots, \alpha_{2\lambda}, \beta_j, \cdots, \beta_{2\lambda}) \leftarrow \mathsf{dO.Eval}(G', x_1 x_2 \cdots x_{2\lambda})$
4: **if** the output is $\perp$ or $\mathsf{PRG}_0(\sigma_i) \neq \alpha_i$ for any $i \in [2\lambda]$ **then**
5: output $\perp$
6: **end if**
7: compute $j = f(x)$ which is the smallest integer, $x = x_{[j]} || 1^{2\lambda - j}$
8: **for** $i = 1, \cdots, j-1$ **do**
9: $\sigma'_i \leftarrow \sigma_i$
10: **end for**
11: **for** $i = j, \cdots, 2\lambda$ **do**
12: $\gamma_i \leftarrow \mathsf{PRG}_1(\sigma_i)$
13: **end for**
14: **for** $i = j, \cdots, 2\lambda$ **do**
15: $\sigma'_i \leftarrow \mathsf{SKE.Dec}(\gamma_j || \cdots || \gamma_{2\lambda}, \beta_i)$
16: **end for**
17: **return** $((x+1) \bmod 2^{2\lambda}, \sigma'_1, \cdots, \sigma'_{2\lambda})$
18: **end procedure**

Here G' is an obfuscated program hard-coded in Next. The circuit G (see algorithm 14) is similar to that in Section 5.5 where the only difference is in the 17-th line; PRG' is a length

doubling injective pseudorandom generator $\{0,1\}^{\lambda/8} \to \{0,1\}^{\lambda/4}$ and Ext_w is a $(\lambda/4, \mathsf{negl}(\lambda))$ strong randomness extractor with seed length $q(\lambda)$ where a random seed w is sampled for Ext.

The KeyGen procedure returns $\mathsf{PK} = \mathsf{Next}$ and $\mathsf{SK} = (S_1, S_2, \cdots, S_{2\lambda})$.

Algorithm 14 G outputs a sequence

procedure $G(x = x_1 x_2 \cdots x_{2\lambda}, P_1(x_1), Q_1(x_1), \cdots, P_{2\lambda}(x), Q_{2\lambda}(x))$
  Hardcoded: a uniformly random $v \in \{0,1\}^{\lambda/4}$
  initialize arrays $\{t_i\}, \{\alpha_i\}, \{\gamma_i\}$ as zeroes for $i = 1 \cdots 2\lambda$;
  initialize j as 1 (j will finally be the smallest integer, $x = x_{[j]}||1^{2\lambda - j}$);
  for $i = 1, 2, \cdots, 2\lambda$ **do**
    if $x_i = 0$ **then**
      fill γ_l and t_l with zeroes for all l between j and $i-1$
      update j as i since $f(x)$ will be at least i
    end if
    compute $s_i = \mathsf{DPRF.Eval}(S_i, x_{[i]})$ by $P_i(x_{[i]})$
    compute and store $\alpha_i = \mathsf{PRG}_0(s_i)$ and $\gamma_i = \mathsf{PRG}_1(s_i)$
    compute and store $t_i = \mathsf{DPRF.Eval}(S_i, x_{[i]} + 1)$ by $Q_i(x_{[i]})$
  end for
  for $i = j, j+1, \cdots, 2\lambda$ **do**
    compute $\beta_i = \mathsf{SKE.Enc}(\gamma_j || \cdots || \gamma_{2\lambda}, t_i)$
  end for
  if $\mathsf{PRG}'(\mathsf{Ext}_w(x)) = v$ **then**
    return $\perp$
  end if
  return $\alpha_1, \cdots, \alpha_{2\lambda}, \beta_j, \cdots, \beta_{2\lambda}$
end procedure

Next let us look at $\mathsf{SampGen}$. The construction of $\mathsf{SampGen}$ and $\mathsf{Sampler}$ requires dO and PKE. The technique is similar to the equivalence of dO and FE in Section 5.4. $\mathsf{SampGen}$ generates two DPRF keys K and K', generates the circuit H corresponding to K, K' and returns $\mathsf{Sampler}$ corresponding to an obfuscated program H'.

The correctness is straightforward to verify. First with probability at least $1 - \frac{1}{2^{\lambda/8}}$, there does not exist x such that $\mathsf{PRG}'(\mathsf{Ext}_w(x)) = v$ for a random v. Thus, for every PK, $\mathsf{TDP}_{\mathsf{PK}}$ is a permutation defined on D_{PK}. And given $\mathsf{SK} = (S_1, \cdots, S_{2\lambda})$, $TDP^{-1}_{\mathsf{PK}}(x, \cdots)$ is computable in polynomial time: get $x' = (x-1) \pmod{2^{2\lambda}}$ and compute $\sigma_i = \mathsf{DPRF.Eval}(S_i, x'_{[i]})$. It remains to prove another two properties "weakly pseudorandom sampling" and "one-wayness".

5.6.3 Weakly pseudorandom sampling

Now we verify "weakly pseudorandom sampling".

Algorithm 15 $\mathsf{Sampler}$ algorithm

procedure $\mathsf{Sampler}(;r)$
  Hardcoded: an obfuscated program H'
  $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\kappa;r)$ where we assume the distribution of pk is uniform
  $(c, c_1, c_2, \cdots, c_{2\lambda}) \leftarrow \mathsf{dO.Eval}(H', \mathsf{pk})$;
  $K_{\mathsf{pk}} \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}, c)$;
  $\sigma'_i \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}, c_i)$ for $1 \le i \le 2\lambda$;
  return $(\mathsf{pk}||K_{\mathsf{pk}}, \sigma'_1, \cdots, \sigma'_{2\lambda})$
end procedure

Algorithm 16 circuit H

procedure $H(\mathsf{pk}, \mathsf{DPRF.Eval}(K,\cdot), \mathsf{DPRF.Eval}(K',\cdot))$
  Hardcoded: $S_1, S_2, \cdots, S_{2\lambda}$
  $Y = \mathsf{DPRF.Eval}(K, \mathsf{pk})$ and let $x = \mathsf{pk}||Y$;
  compute $\sigma_i \leftarrow \mathsf{DPRF.Eval}(S_i, x_{[i]})$;
  encrypt $Y, \sigma_1, \cdots, \sigma_{2\lambda}$ as $c, c_1, \cdots, c_{2\lambda}$ using pk and the randomness from K'_{pk};
  return $(c, c_1, c_2, \cdots, c_{2\lambda})$
end procedure

Proof. Let us prove $\mathsf{Game}_{\lambda,\mathcal{A},0}$ and $\mathsf{Game}_{\lambda,\mathcal{A},1}$ are indistinguishable. We do so by defining a sequence of hybrids:

- **Hyb 0:** The adversary is given the program Next (in other words, the obfuscated program G') , the sampler H', and a random point sampled from $\mathsf{Sampler}$, i.e., the challenge $(\mathsf{pk}^*||K_{\mathsf{pk}^*}, \sigma_1, \cdots, \sigma_{2\lambda})$. In this case, the adversary is in $\mathsf{Game}_{\lambda,\mathcal{A},1}$.

- **Hyb 1:** In this Hybrid, $\mathsf{DPRF.Eval}(K,\cdot)$ and $\mathsf{DPRF.Eval}(K',\cdot)$ is decomposed along pk^* and the adversary gets a new obfuscated circuit $H'_1 = \mathsf{dO.ParaGen}(1^\lambda, H_1)$:

  $$H_1 = \mathsf{DecomposeTo}(H(\cdot, \mathsf{DPRF.Eval}(K,\cdot), \mathsf{DPRF.Eval}(K',\cdot)), \mathsf{pk}^*)$$

  Because H_1 and H are $(\lambda{+}1)$ decomposing equivalent, computational indistinguishability between **Hyb 0** and **Hyb 1** follows from the security of dO.

- **Hyb 2:** We take the circuit fragment in H_1 corresponding to pk^*. This fragment is a constant circuit that always outputs $(c^*, c^*_1, \cdots, c^*_{2\lambda})$ where they are encryptions of $K_{\mathsf{pk}^*}, \sigma_1, \cdots, \sigma_{2\lambda}$ using pk^* and randomness from K'_{pk^*}.

  In this step, we only change K'_{pk^*} with a truly random string and now $(c^*, c^*_1, \cdots, c^*_{2\lambda})$ are encryptions of the same messages but using uniform randomness. Let H_2 and H'_2 be the programs of this step. The indistinguishability between H'_1 and H'_2 comes from the security of DPRF.

- **Hyb 3:** In this step, we change $c^*, c^*_1, \cdots, c^*_{2\lambda}$ in the fragment corresponding to pk^* in H_2 with encryptions of 0^λ. Because the adversary only gets pk^* but not sk^*, the

advantage of distinguishing them is negligible from PKE security. Let H_3 be the program and the fragment in H_3 corresponding to pk^* outputs encryptions of 0^λ with uniform randomness.

- **Hyb 4:** In this step, because K_{pk^*} only appears in the random point sampled from $\mathsf{Sampler}$, K_{pk^*} can be replaced with a random string $z \xleftarrow{R} \{0,1\}^\lambda$ and update $\sigma_i = \mathsf{DPRF.Eval}(S_i, (\mathsf{pk}^*||z)_{[i]})$. The challenge given to the adversary is now $(\mathsf{pk}^*||z, \sigma_1, \cdots, \sigma_{2\lambda})$.

- **Hyb 5:** We restore what we did for $c^*, c_1^*, \cdots$. Now they are encryptions of K_{pk^*} and σ_i with respect to $\mathsf{pk}^*||K_{\mathsf{pk}^*}$. The indistinguishability comes from PKE security. This is identical to **Hyb 2** except the challenge is different.

- **Hyb 6:** We restore what we did for K'_{pk^*}. Instead of replacing it with a random string and using it for encrypting, now we use K'_{pk^*} for encrypting z and σ_i. The security of DPRF implies the probability of distinguishing **Hyb 5** and **Hyb 6** is negligible. This is identical to **Hyb 1** except the challenge is different.

- **Hyb 7:** This is same as **Hyb 0** except the challenge is now $(\mathsf{pk}^*||z, \sigma_1, \cdots)$. In other words, the adversary is now in $\mathsf{Game}_{\lambda,\mathcal{A},0}$ where the challenge corresponds to $\mathsf{pk}^*||z$ which is a uniformly random string in $\{0,1\}^{2\lambda}$. The challenge is now a random point in D_{PK}. The indistinguishability comes from dO security.

□

We prove this construction satisfies the "weakly pseudorandom sampling" property. That is , for any poly sized $\mathcal{A}$, there exists a negligible function negl for all λ,

$$|\Pr[\mathsf{Game}_{\lambda,\mathcal{A},0} = 0] - \Pr[\mathsf{Game}_{\lambda,\mathcal{A},1} = 0]| \leq \mathsf{negl}(\lambda)$$

5.6.4 One-wayness

Let us prove the construction satisfies the "one-wayness" property. The technique is similar to the proof in Section 5.5. The main difference is that given a challenge $(i^*, \cdots)$, we are going to puncture the program G at a random point that is at most $2^{\lambda/4}$ away from i^* to prevent the sampler's image fall into this range.

Proof.

- **Hyb 0:** The adversrary gets $\mathsf{PK} = \mathsf{Next}$ (in other words, the obfuscated program G' in Next), $\mathsf{Sampler}$ (the obfuscated program H' in $\mathsf{Sampler}$) and a random point $(\mathsf{pk}^*||K_{\mathsf{pk}^*}, \sigma_1, \cdots)$ sampled from $\mathsf{Sampler}$.

- **Hyb 1:** The challenge is replaced with a random point $(\mathsf{pk}^*||z, \sigma_1, \cdots)$. The indistinguishability comes from **weakly pseudorandom sampling** property we have proved for this construction. Let us denote the challenge as $(i^*, \sigma_1, \cdots, \sigma_{2\lambda})$.

- **Hyb 2:** In this step, we change how the value v in G is generated. We know that in G, v is a uniformly random string in $\{0,1\}^{\lambda/4}$. Now v is replaced with $v' = \mathsf{PRG}'(u)$ where $u' \xleftarrow{R} \{0,1\}^{\lambda/8}$. The indistinguishability comes from PRG' security. Let us call the new program G_2.

- **Hyb 3:** In **Hyb 2**, $v' = \mathsf{PRG}'(u)$ where u is uniformly chosen from $\{0,1\}^{\lambda/8}$. Let us change how u is chosen. Instead of choosing uniformly from $\{0,1\}^{\lambda/8}$, u' now is computed by $\mathsf{Ext}_w((i^*_{\frac{7}{4}\lambda} - 1)||h)$ where $h \leftarrow \{0,1\}^{\lambda/4}$. The two distributions $U_{\lambda/8}$ and $\mathsf{Ext}_w((i^*_{\frac{7}{4}\lambda} - 1)||U_{\lambda/4})$ (where w is a random seed) are indistinguishable by the definition of a $(\lambda/4, \mathsf{negl}(\lambda))$ randomness extractor.

  After this change, we get the program G_3 with $v'' = \mathsf{PRG}'(u') = \mathsf{PRG}'((\mathsf{Ext}_w(i^*_{7/4\lambda} - 1)||h_0))$ where $h_0 \xleftarrow{R} \{0,1\}^{\lambda/4}$. Now Next always outputs $\perp$ for all $((i^*_{7/4\lambda} - 1)||h_0, \cdots)$.

- In the following hybrids, we are going to puncture the program G_3 such that with overwhelming probability, for any $x \in [(i^*_{7/4\lambda} - 1)||h_0, i^* - 1]$, $\mathsf{TDP}_{\mathsf{PK}}$ is not defined. As a result, $\mathsf{TDP}_{\mathsf{PK}}((i^*, \cdots))$ does not have a preimage. So for any poly sized adversary, it can never find the preimage.

  Let $u_0 = (i^*_{7/4\lambda} - 1)||h_0$. Like what we did for PPAD section, we define $u_1, u_2, \cdots, u_\delta$ such that $(u_{j+1} - u_j)$ is a power of 2 and

  - Recall the definition of f: $f(x)$ is the smallest integer j such that $x_j = x_{[j]}||1^{2\lambda - j}$.
  - $u_i = u_{i-1} + 2^{2\lambda - f(u_{i-1})}$ if the first 7/4 bits of u_i is $(i^*_{7/4\lambda} - 1)$. And finally we get $u_{\delta'} = (i^*_{7/4\lambda} - 1)||1^{\lambda/4}$.
  - $u_{i+1+\delta'} = u_{i+\delta'} + 2^{2\lambda - \rho(u_{i+\delta'}+1, i^*-1)}$ for $i \geq 0$. Here $\rho(x,y)$ returns the smallest index j such that x and y differs at that index. In our proof, the $\rho(x,y)$-th bit of x is always 0 and that of y is always 1.
  - δ is linear in δ, $\delta = O(\lambda)$.

  After defining $u_0 = (i^*_{7/4\lambda} - 1)||h_0$ and $u_1, u_2, \cdots, u_\delta = i^* - 1$, we are going to puncture the program at $[u_i + 1, u_{i+1}]$ for $0 \leq i \leq \delta - 1$ and u_0.

- **Hyb 4:** We decompose the program G_3 and H along u_0, in other words, we decompose $P_1, P_2, \cdots, P_{2\lambda}$ and $Q_1, Q_2, \cdots, Q_{2\lambda}$ along u_0. According to the observation in PPAD section that $\mathsf{DecomposeTo}(Q_m, TC)$ can be reconstructed from $\mathsf{DecomposeTo}(P_m, TC)$ for any tree covering TC, we only care about $P_1, P_2, \cdots, P_{2\lambda}$ in the rest of the proof. The indistinguishability here comes from dO security.

  Let G_4 and H_4 be the circuits in this step. One observation is that the fragment corresponding to u_0 in G_4 always outputs $\perp$.

- **Hyb 5:** In this hybrid, we decompose the program into more pieces and replace some fragments. After this step, the new program and the old program are indistinguishable

but with overwhelming probability, for any $x \in [u_0+1, u_1]$, there does not exist $\sigma_1, \cdots, \sigma_{2\lambda}$ that pass the test $\mathsf{PRG}_0(\sigma_i) = \alpha_i$ for all i.

- **Hyb 5.1 :** In this step, let f_0 be the length of the longest common prefix shared by u_0+1, u_1 and t_0 be the longest common prefix, $|t_0| = f_0$. Because $|u_\delta - u_0| \leq 2 \cdot 2^{\lambda/4}$, we observe that $f_0 > \frac{3}{2} \cdot \lambda$.

  We then decompose the program G_4 and H_4 along the path u_0 and u_1. Let $G_{5.1}$ and $H_{5.1}$ be the programs in this step:

  $$\begin{aligned} G_{5.1} &= \mathsf{CanonicalMerge}(\mathsf{DecomposeTo}(G_4, \{u_0, u_1\})) \\ H_{5.1} &= \mathsf{CanonicalMerge}(\mathsf{DecomposeTo}(H_4, \{u_0, u_1\})) \end{aligned}$$

  This indistinguishability comes from dO security.

- **Hyb 5.2 :** In this hybrid, we change $\mathsf{DPRF.Eval}(S_{f_0}, t_0)$ to a truly random string r_{f_0,t_0}. The indistinguishability comes from DPRF and PRF security. Also we find that r_{f_0,t_0} only appears in the fragment (of the circuit Next) as $\mathsf{PRG}_0(r_{f_0,t_0})$ and $\mathsf{PRG}_1(r_{f_0,t_0})$. The proof is exactly the same as what we did for **Hyb 3.3** in PPAD section. And r_{f_0,t_0} can be removed from $\mathsf{Sampler}$. We know that $f_0 > \frac{3}{2} \cdot \lambda$, let $\hat{\mathsf{pk}} = t_{0,[\lambda]}$.
  - Decompose $\mathsf{DPRF.Eval}(K, \cdot)$ in $\mathsf{Sampler}$ along u_0; (the indistinguishability comes from dO security)
  - Replace $\mathsf{DPRF.Eval}(K, \hat{\mathsf{pk}})$ with a random string, say w_0; (the indistinguishability comes from DPRF security)
  - As long as $\hat{\mathsf{pk}}||w_0$ does not contain t_0 as a prefix, r_{f_0,t_0} can be removed from the $\mathsf{Sampler}$. Because $f_0 > \frac{3}{2} \cdot \lambda$, the probability that it happens is at most $\frac{1}{2^{\lambda/2}}$. So with overwhelming probability, r_{f_0,t_0} can be removed from $\mathsf{Sampler}$.
- **Hyb 5.3 :** As we do not have r_{f_0,t_0} inside the circuits Next and $\mathsf{Sampler}$, we can replace $\mathsf{PRG}_0(r_{f_0,t_0})$ and $\mathsf{PRG}_1(r_{f_0,t_0})$ with truly random strings $v_{0,0}$ and $v_{0,1}$ by PRF security. After it, with overwhelming probability $(1 - 1/2^{2\lambda})$, there does not exist any σ_{f_0} such that $\mathsf{PRG}_0(\sigma_{f_0}) = v_{0,0}$. And the fragment corresponding to u_1 is now returning $\{\alpha\}, \{\beta\}$ where $\{\beta\}$ are now encrypted by random keys $v_{0,1}$.
- **Hyb 5.4 :** Now we no longer have the secret keys for encrypting $\{\beta\}$, we can replace them with encryptions of random strings. The indistinguishability comes from SKE security.

- **Hyb 6** to **Hyb 4 + δ:** At **Hybrid 4 + i** for $2 \leq i \leq \delta$, we puncture at $[u_{i-1}+1, u_i]$. The argument is the same as what we did for **Hyb (2+i)** in PPAD section. We decompose the program along $\{u_0, u_1, \cdots, u_i\}$ and let f_i be the length of the longest common prefix shared by $u_{i-1}+1, u_i$ and t_i be that prefix. The argument in **Hyb 5.1** to **Hyb 5.4** applies.

- **Hyb 5 + δ**: In the final hybrid, for any $x \in [u_0, i^* - 1]$, there does not exist $\sigma_1, \cdots, \sigma_{2\lambda}$ that pass all the tests with overwhelming probability. So for $\mathcal{A}$, it has negligible probability to find a preimage of the given challenge $(i^*, \cdots)$.

□

Acknowledgments

This work is supported in part by NSF. The views expressed are those of the authors and do not reflect the official policy or position of the National Science Foundation or the U.S. Government.

References

[AJ15] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In *Annual Cryptology Conference*, pages 308–326. Springer, 2015.

[AJS15] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Achieving compactness generically: Indistinguishability obfuscation from non-compact functional encryption. Cryptology ePrint Archive, Report 2015/730, 2015. http://eprint.iacr.org/.

[AKV04] Tim Abbott, Daniel Kane, and Paul Valiant. On algorithms for nash equilibria. 2004.

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Annual International Cryptology Conference*, pages 1–18. Springer, 2001.

[BP15] Nir Bitansky and Omer Paneth. *ZAPs and Non-Interactive Witness Indistinguishability from Indistinguishability Obfuscation*, pages 401–427. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[BPR15] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a nash equilibrium. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 1480–1498. IEEE, 2015.

[BPW16] Nir Bitansky, Omer Paneth, and Daniel Wichs. *Perfect Structure on the Edge of Chaos*, pages 474–502. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[BST14] Mihir Bellare, Igors Stepanovs, and Stefano Tessaro. *Poly-Many Hardcore Bits for Any One-Way Function and a Framework for Differing-Inputs Obfuscation*, pages 102–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography Conference*, pages 253–273. Springer, 2011.

[BV15] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 171–190. IEEE, 2015.

[BZ14] Dan Boneh and Mark Zhandry. *Multiparty Key Exchange, Efficient Traitor Tracing, and More from Indistinguishability Obfuscation*, pages 480–499. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[BZ16] Mark Bun and Mark Zhandry. *Order-Revealing Encryption and the Hardness of Private Learning*, pages 176–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[CLTV15] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. *Obfuscation of Probabilistic Circuits and Applications*, pages 468–497. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–17. Springer, 2013.

[GGH+13b] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, FOCS '13, pages 40–49, Washington, DC, USA, 2013. IEEE Computer Society.

[GGHZ16] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. *Functional Encryption Without Obfuscation*, pages 480–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 467–476, New York, NY, USA, 2013. ACM.

[GLSW15] Craig Gentry, Allison Bishop Lewko, Amit Sahai, and Brent Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. In *Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, FOCS '15, pages 151–170, Washington, DC, USA, 2015. IEEE Computer Society.

[GPS16] Sanjam Garg, Omkant Pandey, and Akshayaram Srinivasan. Revisiting the cryptographic hardness of finding a nash equilibrium. In *Annual Cryptology Conference*, pages 579–604. Springer, 2016.

[GPSZ16] Sanjam Garg, Omkant Pandey, Akshayaram Srinivasan, and Mark Zhandry. Breaking the sub-exponential barrier in obfustopia. Technical report, Cryptology ePrint Archive, Report 2016/102, 2016. http://eprint. iacr. org/2016/102, 2016.

[GS16] Sanjam Garg and Akshayaram Srinivasan. Single-key to multi-key functional encryption with polynomial loss. In *Theory of Cryptography Conference*, pages 419–442. Springer, 2016.

[GT16] Shafi Goldwasser and Yael Tauman Kalai. *Cryptographic Assumptions: A Position Paper*, pages 505–522. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[HJK+16] Dennis Hofheinz, Tibor Jager, Dakshita Khurana, Amit Sahai, Brent Waters, and Mark Zhandry. *How to Generate and Use Universal Samplers*, pages 715–744. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[HW15] Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*, ITCS '15, pages 163–172, New York, NY, USA, 2015. ACM.

[KMN+14] I. Komargodski, T. Moran, M. Naor, R. Pass, A. Rosen, and E. Yogev. One-way functions and (im)perfect obfuscation. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 374–383, October 2014.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of yao?s protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[Nao03] Moni Naor. *On Cryptographic Assumptions and Challenges*, pages 96–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[O'N10] Adam O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

[Pap94] Christos H Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and system Sciences*, 48(3):498–532, 1994.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 475–484. ACM, 2014.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.