Exploding 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 *Exploding iO*. We show (1) how to build exploding iO from functional encryption, and (2) how to build a variety of applications from exploding iO, 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, exploding iO represents a convenient new platform for obtaining more applications from polynomial hardness.

Contents

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 simulteneously, 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.

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

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 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.

This exponential loss appears somewhat 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.[2]

The reductions above avoid this argument by not being efficient. The reduction starts with an obfuscation of C_0, and then performs an exponential sequence of steps, one for each input. In the step for input x, the reduction changes the obfuscated program to use C_1 to evaluate x instead of C_0. This change is local, only depending on the input x, and can be shown to be indistinguishable using only the polynomial hardness of the underlying assumptions. In essence, the proof for input x just needs to check that $C_0(x) = C_1(x)$ — which can be done efficiently — as opposed to checking equivalence for all inputs. After performing 2^n local changes, the result is that the obfuscated program now evaluates C_1. The reduction can be thought of as checking equivalence in the brute-force way by running the program on all possible inputs.

Current techniques for building iO seem incapable of circumventing this behavior. Unfortunately, this means every application of iO also suffers from this exponential loss. However, while the exponential loss may be inherent for iO, it is likely not inherent to the application. For example, the work of Garg, Gentry, Halevi, and Zhandry [GGHZ16] shows that this loss is *not* inherent to functional encryption, the original application of iO. Therefore, an important question is whether

[2] One potential objection is that this argument appears to apply to zero knowledge as well, and we know how to build zero knowledge without the exponential loss. Indeed, the zero knowledge property only applies to true instances. However, a fundamental difference is that for zero knowledge, the languages are in NP, and the security reduction potentially gets to know the witness. As such, the reduction does not have to decide if the instance is true and there is no barrier. In contrast, with iO, the language of equivalent circuits is in $coNP$, and unless $NP = coNP$, there will in general be no witnesses that can be given to the reduction. To further illustrate the difference, notice that the security experiment for zero knowledge is not even defined in the case of false statements, since security says that a valid proof (which does not exist in this case) is indistinguishable from a simulated proof. However, the iO security experiment is clearly defined for pairs of inequivalent circuits.

other cutting edge cryptographic applications of iO can be based on just the polynomial hardness of just one or a few assumptions.

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[3].

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: Exploding Indistinguishability Obfuscation

In this work, we define a new notion of obfuscation, called *Exploding Indistinguishability Obfuscation*, or eiO, that addresses the limitations above. This notion abstracts away many of the common techniques in [GPS16, GPSZ16, GS16]; we use those techniques to build eiO from the *polynomial* hardness of functional encryption. Then we can show that the eiO can be used to build the various applications. With our new notion in hand, the eiO-to-Application 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 $f(m_0) = f(m_1)$[4].

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

[4] 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

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 the the circuit that is encrypted in c_y. The root is *explicitly* assigned C by handing out the ciphertext c since we explicity 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 a circuit C from any internal node y[5], 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 binary 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

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

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 replaced with an appropriate 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 output according to the usual rules, (3) any wire that is not used is deleted, and (4) this process is repeated until there are no gates with hardwired inputs and no unused wires. 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 *exploding* the node. The reverse of exploding 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 sequence of ℓ steps (where a step consists of either exploding a node to its children or merging two siblings to the parent) such that at the end of the steps, we are left with C_1 assigned to the root. Then we say the circuits C_0, C_1 are ℓ-path exploding equivalent.

Our new obfuscation notion, called *exploding* iO, is parameterized by ℓ and says, roughly, that the obfuscations of two ℓ-path exploding 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 eiO. 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.

  The loss in the security reduction is proportional to ℓ. Note, however, that the obfuscator itself does not depend on ℓ, and therefore ℓ can be taken to be an arbitrary polynomial or even exponential. If we restrict ℓ to a polynomial, we obtain eiO from polynomially secure FE. Our results also naturally generalize to larger ℓ: we obtain eiO for quasipolynomial ℓ from quasipolynomially secure FE, and we obtain eiO for exponential ℓ from (sub)exponentially secure FE.

- We note that by setting ℓ to be $O(2^n)$, ℓ-exploding equivalence corresponds to standard functional equivalence (since we can make a sequence of moves that hits every leaf). Then eiO coincides with the usual notion of indistinguishability obfuscation, giving us iO from sub-exponential FE. This re-derives the results of [BV15, AJ15].

  Our reduction is actually superior to the reductions in these works; in the prior works, the loss in the reduction turns out for technical reasons to be $2^{\Omega(n^2)}$. In our reduction, the loss is $O(2^n)$, which in some sense is the "right" loss given the sub-exponential barrier discussed above. As such, we obtain a much tighter (though still exponential) reduction from FE to iO. We believe that our improvement here is not due to a fundamental limitation of prior techniques; rather, the improvement is revealed by our new eiO abstraction.

- We then show how to obtain several applications of obfuscation from eiO with *polynomial* ℓ. 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 exploding equivalence: by exploding 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 (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) eiO 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 hot to obtain public key functional encryption from just polynomially hard eiO and one-way functions (more on this below). We do show that a weaker multi-key *secret key* functional encryption scheme does follow from eiO 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, eiO. We hope that eiO will also serve as a starting point for further constructions based on polynomially-hard assumptions.

1.5 Discussion

We observe that poly-exploding equivalence is an NP relation: the polynomial sequence of steps gives a witness that two circuits are equivalent. The security proof, using this witness, can verify that the two circuits are equivalent. In this way, eiO avoids the sub-exponential barrier.

More broadly, one can imagine avoiding the sub-exponential barrier by insisting on circuit pairs for which there is a witness proving equivalence. In other words, restrict to languages of circuit pairs in $NP \cap co\text{-}NP$ [6]. Any languages outside this set are likely to run into the same sub-exponential barrier as full iO, and meanwhile there remains some hope that languages inside might be obfuscatable without a sub-exponential loss.

Moreover, 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 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[7]. This guarantees that x has no pre-image 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.

Unfortunately, exploding iO is not strong enough to prove security in this setting. We demonstrate (Section 3) that ℓ-exploding equivalence can be decided in time proportional to ℓ, meaning poly-exploding equivalence is in P. However, the equivalence of programs $P_{x,m}$ and Z above cannot

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

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

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-exploding equivalent, despite being functionally equivalent programs.

Unsurprisingly then, all the applications we obtain using poly-exploding iO 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, and cannot to our knowledge be based on poly-eiO. In other words, unlike iO, eiO 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 eiO from functional encryption, we get public key encryption for free. However, if we are interested in placing eiO 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 for 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[8], 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.

  This strategy simply does not work in the context of eiO. Indeed, eiO only applies to circuits for which equivalence is easily decidable anyway, meaning no contradiction is reached. In order to obtain any results analgous 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, eiO could exist in Pessiland.

- More generally, eiO 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.

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

- Identify more applications for which obfuscation *must* be applied to program pairs with non-efficiently checkable equivalence.

2 Preliminaries: Definitions and Notations

In this paper, we use κ to denote the security parameter. We denote $\mathsf{poly}(\cdot)$ as a polynomial. 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. We will use $\mathrm{negl}(\cdot)$ to denote a negligible function. When 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, $x||y$ represents the concatenation of x and y.

One-Way Function

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

Definition 2.1. An 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 Generator

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
- **Security :** 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. With a pseudo random generator, we can construct a symmetric key encryption scheme from it.

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 cipher c;
- $\mathsf{SKE.Dec}(\mathsf{sk}, c)$ is a polynomial time algorithm that takes a secret key sk and a cipher $c \in \{0,1\}^*$, 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 cipher $c = \mathsf{SKE.Enc}(\mathsf{sk}, m_b)$ 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 every poly sized adversary $\mathcal{A}$, there exists a negligible function negl such that for every κ

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

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 cipher c;
- $\mathsf{PKE.Dec}(\mathsf{sk}, c)$ is a polynomial time algorithm that takes a secret key sk and a cipher $c \in \{0,1\}^*$, 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 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 cipher $c = \mathsf{SKE.Enc}(\mathsf{sk}, m_b)$ 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 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 Obfuscator

Here is the definition for indistinguishability obfuscator [BGI+01, GGH13a].

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

- **Preserve 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 cipher 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 cipher, 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 $\{0,1\}^n$,

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

Also most functional encryption schemes FE in our paper will be **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 cipher $c = \mathsf{FE.Enc}(\mathsf{mpk}, m_b)$ and sends $(\mathsf{mpk}, \mathsf{msk}, c)$ 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 Exploding Equivalence

In this section, we define several basic ideas that will be used for defining and constructing eiO.

3.1 Partial Evaluation 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.

It is easy to see that when C is defined on a single bit, the partial evaluation of C is a circuit with no input, just gives the output.

Definition 3.2. Consider a circuit C defined on inputs of length $n > 0$, for any binary string x of length at most n, a **partial evaluation** of C on a string x denoted as $C(x, \cdot)$ is a circuit defined on inputs of length $n - |x|$ where we repeatedly apply partial evaluations on bit x_i for $i = 1, \ldots, |x|$ to get $C(x, \cdot)$.

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

Definition 3.3 (Binary Tree)**.** A binary tree T_n is a tree of depth $n+1$ where the root 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. Leaves of the tree T_n are labeled from 0^n to 1^n. T_n has $2^{n+1}-1$ nodes and depth $n+1$ (if the root has depth 1).

We will 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.

Definition 3.4 (Tree Covering)**.** We say a set of binary string $\{x_i\}_{i=1}^{\ell}$ 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 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 easy to see that if $TC_0 \preceq TC_1$, then $|TC_0| \leq |TC_1|$ where $|TC_0| = |TC_1|$ if and only 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 3.5 (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: finds the unique x_j which is a prefix of $x = x_j||x_{-j}$ and returns $C_{x_j}(x_{-j})$.

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 overload 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.

- $\mathsf{Explode}(L, x)$ 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 explodes 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 merge them to get the following circuit $C_x(b, y) = (b \wedge C_{x||0}(y)) \vee (\bar{b} \wedge C_{x||1}(y))$. 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 where and $C_{x||0}, C_{x||1}$ can not be simplified any further, $\mathsf{Explode}(\mathsf{CanonicalMerge}(L, x), x) = L$.

- $\mathsf{TargetedMerge}(L, x, C)$ 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

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

- $\mathsf{ExplodeTo}(L, x)$: 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 u to x.

  $\mathsf{ExplodeTo}$ first sets $L_0 = L$, and then runs $L_i \leftarrow \mathsf{Explode}(L_{i-1}, p_{i-1})$ 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 $\mathsf{ExplodeTo}$ to operate on circuits in addition to assignments, by first interpreting the circuit as an assignment, and performing $\mathsf{ExplodeTo}$ on the assignment.

- $\mathsf{ExplodeTo}(L, TC)$: It takes a circuit assignment L and a tree covering TC where TC is below the covering in L. Let $x_1, \ldots, x_t$ be the nodes in TC in order from left to right.

  $\mathsf{ExplodeTo}$ first sets $L_0 = L$, and then runs $L_i \leftarrow \mathsf{ExplodeTo}(L_{i-1}, x_i)$ for $i = 1, \ldots, t$. The output is the new circuit assignment $L' = L_t$.

  Notice that the new tree covering for L' is exactly TC.

  We can also define $\mathsf{ExplodeTo}(L, \{x_i\}_{i=1}^m)$ analogously for arbitrary sets of nodes x_i such that no x_i is an ancestor of an x_j. In this case, the new tree covering TC for L' is the minimal TC such that TC is below the tree covering for L and contains each of the x_i.

  Again, we can also extend this definition to operate on circuit inputs.

- $\mathsf{CanonicalMerge}(L, TC)$: 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.

- $\mathsf{CanonicalMerge}(L) = \mathsf{CanonicalMerge}(L, \{\varepsilon\})$. In other words, $\mathsf{CanonicalMerge}(L)$ canonically merges all the way to the root. We will alternatively think of the output as being a circuit assignment, as well as a circuit (corresponding to the circuit assigned to the root).

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

3.3 New Notions of Equivalence for Circuits

We now define two new equivalence notions for circuits based on the explode and merge operations defined above. First, we define a local equivalence condition on circuit assignments:

Definition 3.6 (locally exploding equivalent)**.** We say two circuit assignments $L_1 = \{(x_i, C_{x_i})\}, L_2 = \{(y_i, C'_{y_i})\}$ are (ℓ, s)-**locally exploding equivalent** if the following hold:

- The circuit size of L_1, L_2 is at most s;
- The cardinality of L_1, L_2 is at most ℓ;
- L_1 can be obtained from L_2 by applying $\mathsf{Explode}(L_2, x)$ 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 exploding equivalence (Local EE) means that we can transform L_1 into L_2 by making just a single simple local change, namely exploding a node or merging two nodes. Notice that since exploding a node does not change functionality, local EE implies that L_1 and L_2 compute equivalent functions. For any ℓ, s, (ℓ, s)-local exploding equivalence forms a graph, where nodes are circuit assignments and edges denote local exploding equivalence. Next, we define a notion of *path* exploding equivalence for circuits (which can be thought of 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 3.7 (path exploding equivalent)**.** We say two circuits C_1, C_2 are (ℓ, s, t)-**path exploding equivalent** if there exists at most $t-1$ circuit assignments $L'_1, L'_2, \cdots, L'_{t-1}$ such that, for any $1 \leq i \leq t$, L'_{i-1} and L'_i are (ℓ, s)-locally exploding equivalent, where $L'_0 = \{(\varepsilon, C_1)\}$ and $L'_t = \{(\varepsilon, C_2)\}$.

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

Definition 3.8 (one shot exploding equivalent)**.** Given two circuits C_0, C_1 defined on inputs of length n, we say they are τ-**one shot exploding equivalent** or simply τ-**exploding 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 exploding equivalent" is that there exists a tree covering $\mathcal{X}$ of size at most τ, such that $\mathsf{ExplodeTo}(\{(\varepsilon, C_0)\}, \mathcal{X}) = \mathsf{ExplodeTo}(\{(\varepsilon, C_1)\}, \mathcal{X})$, 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 exploding equivalence all preserve functionality, we have that these notions imply standard functional equivalence for the circuits in question:

Lemma 3.9. *If C_0, C_1 are (ℓ, s, t)-path exploding equivalent for any ℓ, s, t, or if C_0, C_1 are τ-one shot exploding 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 3.10. *Two circuits C_0, C_1 (defined on n bits string) are 2^n-one shot exploding 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 exploding 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 $\mathsf{ExplodeTo}(\{(\varepsilon, C_0)\}, \mathcal{X}) = \mathsf{ExplodeTo}(\{(\varepsilon, C_1)\}, \mathcal{X})$. Therefore C_0, C_1 are 2^n-one shot exploding equivalent. □

3.4 Deciding Exploding Equivalence

Definition 3.11. A tree covering TC is a *witness that* $C_0 \equiv C_1$ if TC satisfies $\mathsf{ExplodeTo}(\{(\varepsilon, C_0)\}, \mathcal{X}) = \mathsf{ExplodeTo}(\{(\varepsilon, C_1)\}, \mathcal{X})$. In other words, exploding C_0 and C_1 to TC give 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 3.12. *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. Since $C_0 \equiv C_1$, all the leaves are good, and at least the set of leaves form a tree covering that is a witness. Now, for each leaf, consider the path from the leaf to the root. 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. □

Lemma 3.13. *τ-one shot exploding 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, keeping a global list R. When processing a node x, if $C_0(x, \dots) = C_1(x, \cdot)$ as circuits, add x to R, and then do not recurse. Otherwise, recurse on the children as normal. If the list R every grows to exceed τ elements, abort the search and report non-exploding equivalence. If the search finishes with $|R| \leq \tau$, then report exploding equivalence and output 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 (the 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 equivalent.

Lemma 3.14. *If two circuits C_0, C_1 are (ℓ, s, t)-path exploding equivalent, then they are $(t/2+1)$-one shot exploding equivalent*

Proof. If C_0, C_1 are (ℓ, s, t)-path exploding equivalent, 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 exploded, and there must also be a step in the path where that node is merged. It is straightforward to show that 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 exploding equivalent. □

We emphasize that the above lemma and proof were independent of the bounds ℓ and s. Next, we show a converse statement:

Lemma 3.15. *If two circuits C_0, C_1 are $(t/2 + 1)$-one shot exploding equivalent, then they are $(n + 1, s, t)$-path exploding 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 Explode 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 Explode and TargetedMerge for each "bad" node, corresponding to t operations. □

Putting together Lemmas 3.14 and 3.15, we find that the path equivalence definition is independent of the parameters ℓ, s. We also see that path exploding equivalence can be computed efficiently, following Lemmas 3.13, 3.14 and 3.15.

Finally, we show that path/one-shot exploding equivalence is a strictly stronger notion that 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 exploding equivalence can be decided in polynomial time, whereas in general deciding equivalence is hard.

Lemma 3.16. *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 exploding 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 exploding 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 exploding equivalence is decideable in polynomial time, but functional equivalence is not (assuming $P \neq NP$), there must be circuits pairs that are equivalent but not poly-one shot exploding equivalent.

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

Lemma 3.17. *Assuming one-way functions exist, there is an explicit family of circuit pairs* (C_0, C_1) *that are equivalent, but are not* $\mathsf{poly}(n)$*-exploding 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-exploding equivalent with probability δ. Now let us 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-exploding 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-exploding equivalent which is a non-negligible δ. So $\mathcal{B}$ has advantage non-negligible advantage δ in breaking PRG. □

4 Exploding Indistinguishability Obfuscator

In this section, we will define exploding iO (eiO). Exploding iO, roughly, is an indistinguishability obfuscator, but where the indistinguishability security requirement only applies to pairs of circuits that are exploding equivalent (as opposed to applying to all equivalent circuits).

Definition 4.1. eiO wtih two PPT algorithms $\{\mathsf{eiO.ParaGen}, \mathsf{eiO.Eval}\}$ is a $\tau(n, s, \kappa)$-exploding indistinguishability obfuscator if the following conditions hold

- **Efficiency:** $\mathsf{eiO.ParaGen}, \mathsf{eiO.Eval}$ are efficient algorithms;
- **Functionality preserving:** $\mathsf{eiO.ParaGen}$ takes as input a security parameter κ and a circuit C, and outputs the description $\hat{C}$ of an obfuscated program. For all κ and all circuit C, for all input $x \in \{0,1\}^n$, we have $\mathsf{eiO.Eval}(\mathsf{eiO.ParaGen}(1^\kappa, C), x) = C(x)$;
- **Exploding 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

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

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

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

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

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

Definition 4.2. leiO with two PPT algorithms $\{\mathsf{leiO.ParaGen}, \mathsf{leiO.Eval}\}$ is a locally exploding indistinguishability obfuscator if the following conditions hold

- **Efficiency:** $\mathsf{leiO.ParaGen}, \mathsf{leiO.Eval}$ are efficient algorithms;
- **Functionality preserving:** $\mathsf{leiO.ParaGen}$ 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 $\mathsf{leiO.Eval}(\mathsf{leiO.ParaGen}(1^\kappa, L, \ell, s), x) = L(x)$;
- **Local exploding 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

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

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

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

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

Now we show that the existence of leiO implies the existence of eiO.

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

Proof. The construction of leiO from eiO 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 leiO to be $n+1$ and the circuit size to be $|C|$.

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

Efficiency and functionality preservation are easy to prove. 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 $\tau(n, s, \kappa)$-exploding equivalent with probability 1. Then C_0 and C_1 are also $(n+1, s, 2(\tau-1))$-path exploding equivalent by Lemma 3.15. By the definition of path exploding equivalence and Lemma 3.13, there exist (efficiently computable) $L'_1, L'_2, \cdots, L'_{2(\tau-2)}, L'_{2(\tau-1)-1}$ 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 exploding equivalent. So we have that

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

Here, ϵ is the advantage of the following adversary pair (Samp', D) in the local eiO security game (where D is from above). Samp' runs $(C_0, C_1, \sigma) \leftarrow \mathrm{Samp}'$, 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 eiO where the loss is $2(\tau-1)$. If we assume the polynomial hardness of leiO, the adversary (Samp', D) must have negligible advantage ϵ, and so we get $\tau-\mathsf{eiO}$ for any polynomial τ. If we assume the subexponential hardness of leiO, 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-eiO, 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-eiO and hence subexponentially secure iO. □ □

Next, we focus on constructing leiO, which we now know is sufficient for construction eiO.

4.1 Compact FE implies eiO

Theorem 4.4. *If compact single-key selective secure functional encryption schemes and one-way functions exist, then there exists local exploding indistinguishability obfuscators* leiO.

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 and one-way functions exist, then there exist exploding indistinguishability obfuscators* eiO.

Now we prove theorem 4.4.

Algorithm 1 locally exploding indistinguishability obfuscator $\mathsf{leiO.ParaGen}$

procedure $\mathsf{leiO.ParaGen}(1^\kappa, L = \{(x_i, C_{x_i})\}, \ell, s)$
  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 specified later;
  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
  $\mathsf{fsk}_{n+1}^b \leftarrow \mathsf{FE.KeyGen}(\mathsf{msk}_{n+1}^b, g^b)$ 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

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

Algorithm 2 generating c_0, c_1 recursively

procedure $\mathsf{CGen}(x, L)$
  if L only contains one pair, it must be (x, C_x) **then**
    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

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 3). But instead of doing this, this function also allows us to cheat and output a result given a secret key.

For each function f_{n+1}^b, it is given a circuit with no input, evaluates it and outputs a result (see

Algorithm 3 f_i^{b,Z_i^b} for $1 \leq i \leq n$

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 4 f_{n+1}^b

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

algorithm 4).

Evaluation and Correctness

Now let us look at how $\mathsf{leiO.Eval}$ 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: $(c_0, c_1) \leftarrow \mathsf{FE.Dec}(\mathsf{fsk}_i^{x_i}, c_{x_i})$;
- Then we can either output $\mathsf{FE.Dec}(\mathsf{fsk}_{n+1}^0, c_0)$ or $\mathsf{FE.Dec}(\mathsf{fsk}_{n+1}^1, c_1)$;

$\mathsf{leiO.Eval}(c_0, c_1, \{\mathsf{mpk}_i^0, \mathsf{mpk}_i^1\}_{i=1}^{n+1}, \{\mathsf{fsk}_i^0, \mathsf{fsk}_i^1\}_{i=1}^{n+1}, \cdots)$ 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'. Since the cardinality is at most ℓ, ℓ different $Z_{i,j}^b$ in 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 ciphers 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 $\{\mathsf{fsk}_i^b\}$ is bounded by $O(\mathsf{poly}(s, \ell, \kappa))$. Finally t_1 is bounded by $O(\mathsf{poly}(s, \log \ell, \kappa))$.

So $\mathsf{leiO.ParaGen}$ and $\mathsf{leiO.Eval}$ 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 $\mathsf{Explode}(L_0, x) = L_1$. We are going to prove the indistinguishability when we are given either L_0 or L_1.

- **Hyb 0**: In this hybrid, an adversary is given an instance $\mathsf{leiO.ParaGen}(1^\kappa, L_0, \ell, s)$. In the process of generating c_0, c_1, we will get to CGen on x and L' where L' is the current partial circuit assignment. 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 $\mathsf{leiO.ParaGen}$, 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_0}^0 and Z_{d,j_1}^1 to encryptions 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 means **Hyb 0** and **Hyb 1** are indistinguishable.

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

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

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

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

  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 pseudo random 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 $\mathsf{leiO.ParaGen}(1^\kappa, L_1, \ell, s)$.

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 an exploding compatible pseudo random function.The construction [GGM86] automatically satisfies the definition below.

Definition 5.1. An exploding compatible pseudo random function EPRF consists the following algorithms $\mathsf{EPRF.KeyGen}$ and $\mathsf{EPRF.Eval}$ where

- $\mathsf{EPRF.Eval}$ 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,

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

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

- **(EPRF Security)** Consider the following game, let $\mathsf{Game}_{\kappa,\mathcal{A},b}$ be
  - The challenger prepares $S \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$;
  - The adversary makes queries about x and gets $\mathsf{EPRF.Eval}(S, x)$ back from the challenger;
  - The adversary gives 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 distribution D_b back to the adversary $\mathcal{A}$ where
    - D_0: let the circuit D to be $D(\cdot) = \mathsf{EPRF.Eval}(S, \cdot)$ defined on $\{0,1\}^n$, the circuit assignment is $\mathsf{ExplodeTo}(D, TC)$.
      One observation is that the fragment corresponding to y is $\mathsf{EPRF.Eval}(S, y, \cdot)$ defined on $\{0,1\}^{n-|y|}$.
    - D_1: For each $y \neq y^* \in TC$, let the fragment corresponding to y be $D_y(\cdot) = \mathsf{EPRF.Eval}(S, y, \cdot)$ defined on $\{0,1\}^{n-|y|}$ and for y^*, $D_{y^*}(\cdot) = \mathsf{EPRF.Eval}(S', y^*, \cdot)$ defined on $\{0,1\}^{n-|y^*|}$ where $S' \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$.
  - The adversary can keep making queries about x which does not have prefix y^* and gets $\mathsf{EPRF.Eval}(S, x)$ 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 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 $L' = \mathsf{ExplodeTo}(C, TC) = \{(x, C_x)\}_{x \in TC}$
- 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*

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

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

$$\mathsf{ExplodeTo}(C(\cdot, D_1(\cdot), \cdots, D_m(\cdot)), TC) = C(\cdot, \mathsf{ExplodeTo}(D_1, TC), \cdots, \mathsf{ExplodeTo}(D_m, TC))$$

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 $\mathsf{ExplodeTo}(D, TC)$ 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 eiO to build short signatures, following [SW14]. As in [SW14], we will construct statically secure signatures.

The signature is simply of the following form $f(\mathsf{EPRF.Eval}(S, m))$ where f is a one-way function and EPRF is a prefix puncturable pseudo random function. Now let us look at the details.

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 $m \in \{0,1\}^\ell$:

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

We now define security for 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 to $\mathsf{Sign}(\mathsf{sk}, m')$ 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 eiO instead of iO. Our SS has the following algorithms:

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

Algorithm 5 Verification Algorithm

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

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

It is easy to see that the construction satisfies correctness.

Security

Theorem 5.6. *If* eiO *is a secure* $\mathsf{poly\text{-}eiO}$, EPRF *is a secure exploding compatible* PRF, *and* f *is a secure one-way function, then the construction above is a secure signature scheme.*

Proof. Now prove 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 explode the circuit to get $L = \mathsf{ExplodeTo}(V(\cdot, \mathsf{EPRF.Eval}(S, \cdot)), m^*)$. Since we have lemma 5.3, the circuit assignment is just $V(\cdot, \mathsf{ExplodeTo}(\mathsf{EPRF.Eval}(S, \cdot), m^*))$.

  So $\mathsf{eiO.ParaGen}(1^\kappa, V(\cdot, \mathsf{EPRF.Eval}(S, \cdot)))$ and $\mathsf{eiO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L))$ are indistinguishable, since these two circuits are $\ell + 1$-exploding equivalent by applying eiO,
- **Hyb 2:** In this hybrid, we replace the fragment in $\mathsf{ExplodeTo}(\mathsf{EPRF.Eval}(S, \cdot), m^*)$ corresponding to m^* — which is "**return** $\mathsf{EPRF.Eval}(S, m^*)$" — by "**return** $\mathsf{EPRF.Eval}(S', m^*)$" where $S' \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$ is a fresh random EPRF key that is independent of S. We call the new circuit assignment L'. **Hyb 1** and **Hyb 2** are indistinguishable because of the EPRF security.

- **Hyb 3:** In this hybrid, we replace the fragment in L', which is "**return** $\mathsf{EPRF.Eval}(S', m^*)$" 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 eiO. 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 D^* and a sample s^* in the image of D^*.

Correctness

For any C^* and s^* in the image of C^*, and for any $\ell \geq |C^*|$, and t is a feasible 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

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

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 to be the sizes of circuits fed into the universal sampler). It is easy to see that $U = \mathsf{poly}(\kappa, \ell, t)$; Whenever we mention $\mathsf{eiO.ParaGen}(1^\kappa, C)$, we will pad C to have size U.
- For simplicity, we will assume circuits C fed into the universal sampler will always be 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 $S \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$, 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 EPRF; and the output of the procedure $\mathsf{US.Setup}$ is $\mathsf{params} = \mathsf{eiO.ParaGen}(1^\kappa, \mathsf{Sampler}(\cdot, \mathsf{EPRF.Eval}(S, \cdot)))$.

Algorithm 6 Sampler Algorithm

```
procedure Sampler(C = c_1 c_2 ··· c_ℓ, EPRF.Eval(S, C))
    r_C ← EPRF.Eval(S, C)
    return C(; r_C)
end procedure
```

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

Therefore now prove the following theorem.

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

Proof. First, it is easy to see that correctness is satisfied. Next we prove security. Fix a circuit C^* and suppose there is an adversary $\mathcal{A}$ for the sampler security game for 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 $s^* \leftarrow C^*(; \mathsf{EPRF.Eval}(S, C^*))$. We get $\mathsf{params}_1 \leftarrow \mathsf{US.Sim}(1^\kappa, 1^\ell, 1^t, C^*, s^*)$ where $\mathsf{Sampler}_1$ is the circuit constructed in $\mathsf{US.Sim}$ where we are using the same S in **Hyb 0**.

  It is straightforward that $\mathsf{Sampler}_1$ and $\mathsf{Sampler}$ are $\ell + 1$-exploding equivalent. Therefore $\mathsf{params}_1 = \mathsf{eiO.ParaGen}(1^\kappa, \mathsf{Sampler}_1)$ and $\mathsf{params} = \mathsf{eiO.ParaGen}(1^\kappa, \mathsf{Sampler})$ are indistinguishable by eiO security, meaning **Hyb 0** and **Hyb 1** are indistinguishable.
- **Hyb 2 :** In this hybrid, we replace the fragment in $\mathsf{ExplodeTo}(\mathsf{EPRF.Eval}(S, \cdot), C^*)$ corresponding to C^* with "**return** $\mathsf{EPRF.Eval}(S', C^*)$" where $S' \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$ is a new key generated by a uniformly random string. We call the new circuit assignment L'. The indistinguishability between **Hyb 1** and **Hyb 2** follows from the EPRF security.

- **Hyb 3:** In this hybrid, since the fragment in L' corresponding to C^* is now returning $C^*(;\mathsf{EPRF.Eval}(S', C^*))$ 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 eiO and FE assuming Public Key Encryptions

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

Theorem 5.9. *Assume public key encryption, one-way function and* eiO *exist, there exists compact (multi-key selective secure) functional encryption scheme.*

Our construction is similar with [GS16]. Before mentioning the construction, we will first give two more definitions.

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 $\mathsf{DSKE.KeyGen}$, $\mathsf{DSKE.Enc}$, $\mathsf{DSKE.Dec}$, $\mathsf{DSKE.InRange}$ 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 cipher c;
- $\mathsf{DSKE.Dec}(\mathsf{sk}, c)$ is a polynomial time algorithm that takes a secret key sk and a cipher $c \in \{0,1\}^*$, 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 cipher c is in $\mathsf{Range}_n(\mathsf{sk})$;
- **Correctness :** a 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.

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

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

Definition 5.11 (Garbled Circuits [Yao86, LP09])**.** A circuit garbling scheme consists a tuple of PPT algorithms $(\mathsf{Garb.ParaGen}, \mathsf{Garb.Eval})$ 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 the Definition 5.11 [Yao86, LP09].

5.4.2 eiO and Public Key Encryption implies Compact FE

We will first prove eiO implies single-key compact functional encryption. The proof of multi-key FE from eiO 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 randomly samples $S \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$ and $K \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$; consider the following algorithm:

Algorithm 7 Algorithm G with S, K hardcoded

procedure $G(\mathsf{pk} = \mathsf{pk}_1\mathsf{pk}_2, \cdots \mathsf{pk}_\kappa, \mathsf{EPRF.Eval}(S, \mathsf{pk}), \mathsf{EPRF.Eval}(K, \mathsf{pk}))$
  $K_{\mathsf{pk}} \leftarrow \mathsf{EPRF.Eval}(S, \mathsf{pk})$;
  $S_{\mathsf{pk}} \leftarrow \mathsf{EPRF.Eval}(K, \mathsf{pk})$;
  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 $\mathsf{mpk} = \mathsf{eiO.ParaGen}(1^\kappa, G(\cdot, \mathsf{EPRF.Eval}(S, \cdot), \mathsf{EPRF.Eval}(K, \cdot)))$ 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 $1 \le i \le n$;

Algorithm 8 FE encryption algorithm

procedure $\mathsf{FE.Enc}(\mathsf{mpk}, m)$
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\kappa)$;
  $c \leftarrow \mathsf{eiO.Eval}(\mathsf{mpk}, \mathsf{pk})$;
  $S'_{\mathsf{pk}} \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}, c)$;
  $\hat{S}'_{\mathsf{pk}} \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$ using the randomness from S'_{pk};
  for $i = 1, 2, \cdots, n$ **do**
    $L_{i,m_i} \leftarrow \mathsf{EPRF.Eval}(\hat{S}'_{\mathsf{pk}}, i||m_i)$; ▷ Here i is padded to have $\lceil \log_2 n \rceil$ bits before being feed to EPRF
  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 constructs the following circuit where $K' \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$:

Algorithm 9 Algorithm H

procedure $H(\mathsf{pk} = \mathsf{pk}_1\mathsf{pk}_2\cdots\mathsf{pk}_\kappa, \mathsf{EPRF.Eval}(S, \mathsf{pk}), \mathsf{EPRF.Eval}(K', \mathsf{pk}))$
  $K'_{\mathsf{pk}} \leftarrow \mathsf{EPRF.Eval}(K', \mathsf{pk})$;
  $S_{\mathsf{pk}} \leftarrow \mathsf{EPRF.Eval}(S, \mathsf{pk})$;
  $\hat{S}_{\mathsf{pk}} \leftarrow \mathsf{EPRF.KeyGen}(1^\kappa)$ 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 $L_{i,b} = \mathsf{EPRF.Eval}(\hat{S}_{\mathsf{pk}}, i||b)$;
  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 $\mathsf{eiO.ParaGen}(1^\kappa, H(\cdot, \mathsf{EPRF.Eval}(S, \cdot), \mathsf{EPRF.Eval}(K', \cdot)))$ (H is padded to a length upper bound U_2).

- $\mathsf{FE.Dec}(\mathsf{fsk}_f, c)$ where $\mathsf{fsk}_f = \mathsf{eiO.ParaGen}(1^\kappa, H)$ 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 $\mathsf{eiO.Eval}(\mathsf{fsk}_f, \mathsf{pk})$. 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 check it. 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 $\{c_{m_i}\}$, by correctness of the circuit garbling scheme, $\mathsf{Garb.Eval}(\widetilde{C}, \{c_{i,m_i}\}) = C_f(m)$,

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 $\mathsf{PKE.KeyGen}$, $\mathsf{eiO.Eval}$, $\mathsf{PKE.Dec}$, EPRF and the output size of $\mathsf{eiO.ParaGen}$ 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: $\mathsf{mpk} = \mathsf{eiO.ParaGen}(1^\kappa, G(\cdot, \mathsf{EPRF.Eval}(S, \cdot), \mathsf{EPRF.Eval}(K, \cdot)))$, $c^* = (\mathsf{pk}^*, \{L_{i,m_{0,i}}\}_{i\in[n]})$, and $\mathsf{fsk}_f = \mathsf{eiO.ParaGen}(1^\kappa, H(\cdot, \mathsf{EPRF.Eval}(S, \cdot), \mathsf{EPRF.Eval}(K', \cdot)))$.
- **Hyb 1**: In this hybrid, we explode the program G to get a circuit assignment $L_G = \mathsf{ExplodeTo}(G, \mathsf{pk}^*)$. By lemma 5.3, indeed we have

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

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

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

  Then we get its corresponding circuit $\mathsf{CanonicalMerge}(L_H)$ (pad it to length U_2) and a new function key $\mathsf{fsk}_1 = \mathsf{eiO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L_H))$. The indistinguishability comes from the security of eiO.
- **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 EPRF security, K_{pk^*} can be replaced as $\mathsf{EPRF.Eval}(K'', \mathsf{pk}^*)$ 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 new $\mathsf{mpk}_2 = \mathsf{eiO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L'_G))$. The indistinguishability comes from PRF and EPRF 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^* here 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 EPRF.

- **Hyb 6**: By the EPRF security, we can replace the fragment $\mathsf{EPRF.Eval}(S, \mathsf{pk}^*)$ in $\mathsf{ExplodeTo}(\mathsf{EPRF.Eval}(S, \cdot))$ with $\mathsf{EPRF.Eval}(S', \mathsf{pk}^*)$ where S' is a newly generated key using fresh randomness. We find that $\mathsf{EPRF.Eval}(S', \mathsf{pk}^*)$ 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 $\mathsf{EPRF.Eval}(S', \mathsf{pk}^*)$ but not S'.

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

- **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 EPRF 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 that part. 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 ciphers 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 $c^* = (\mathsf{pk}^*, \{r_{i,m_{0,i}}\}_{i\in[n]})$, $\mathsf{mpk} = \mathsf{eiO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L''_G))$ and $\mathsf{fsk}_f = \mathsf{eiO.ParaGen}(1^\kappa, \mathsf{CanonicalMerge}(L''_H))$ where $\mathsf{eiO.Eval}(\mathsf{fsk}_f, \mathsf{pk}^*) = (\widetilde{C_f}, \{c_{i,m_{0,i}} = \mathsf{SKE.Enc}(r_{i,m_{0,i}}, \mathsf{inp}_{i,m_{0,i}})\}_{i\in[n]} \cup \{c_{i,\neg m_{0,i}} = \mathsf{SKE.Enc}(r_{i,\neg m_{0,i}}, \bot\}_{i\in[n]})$.

- **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 an 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 for security.

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

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 eiO

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 exists 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 $x \in I$ 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 TNFP and is defined by its complete problem called as END-OF-LINE (abbreviated as EOL).

Definition 5.13. $\mathsf{EOL} = \{I_{\mathsf{EOL}}, R_{\mathsf{EOL}}\}$ 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 $(i, R) \in \mathsf{TNFP}$ 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 $I_{\mathsf{SVL}} = \{(x_s, \mathsf{Succ}, \mathsf{Ver}, T)\}$ and $R_{\mathsf{SVL}}((x_s, \mathsf{Succ}, \mathsf{Ver}, T), w) = 1 \iff \mathsf{Ver}(w, T) = 1$.

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 eiO implies PPAD hardness, we only need to show eiO implies SVL hardness.

5.5.2 Construction

The construction is similar with it from [GPS16]. Now let us look at the construction.

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

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

- $T = 1^\kappa$;

where $S_1, S_2, \cdots, S_\kappa$ are sampled from $\mathsf{EPRF.KeyGen}(1^\kappa)$. And PRG is a pseudo random 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 $P_1(x_1) = \mathsf{EPRF.Eval}(S_1, x_1)$ defined on inputs of length 1, $P_2(x_{[2]}) = \mathsf{EPRF.Eval}(S_2, x_{[2]})$ defined on inputs of length 2, and similarly $P_\kappa(x) = \mathsf{EPRF.Eval}(S_\kappa, x)$ 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 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.)

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 zeros 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 zeros for all l between j and $i - 1$
      update j as i since $f(x)$ will be at least i
    end if
    compute $s_i = \mathsf{EPRF.Eval}(S_i, x_{[i]})$ 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 $t_i = \mathsf{EPRF.Eval}(S_i, x_{[i]} + 1)$ 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

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 $\mathsf{EPRF.KeyGen}(1^\kappa)$ 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,

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

Algorithm 11 Next computes the next feasible node

procedure $\textsc{Next}(x = x_1 x_2 \cdots x_\kappa, \sigma_1, \cdots, \sigma_\kappa)$
  Hardcoded : $G' \leftarrow \mathsf{eiO.ParaGen}(1^\kappa, G(\cdot, P_1(\cdot), Q_1(\cdot), \cdots, P_\kappa(\cdot), Q_\kappa(\cdot)))$
  $(\alpha_1, \cdots, \alpha_\kappa, \beta_j, \cdots, \beta_\kappa) \leftarrow \mathsf{eiO.Eval}(G', x_1 x_2 \cdots x_\kappa)$
  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 $P_m(x+1) = \mathsf{EPRF.Eval}(S_m, x+1)$

procedure $Q_m(x = x_1 x_2 \cdots x_m)$
  let P be the fragment $P_m(\cdot)$
  let $\mathsf{tmp} = \varepsilon$
  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

the 17-th line of G will never be true. 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, $\mathsf{PRG}(\sigma_i) = \alpha_i$. 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 $\mathsf{EPRF.Eval}(S_i, x_{[i]}) = \mathsf{EPRF.Eval}(S_i, (x+1)_{[i]})$ for $1 \le i \le j-1$. Then we decrypt $\beta_j, \cdots, \beta_\kappa$ to get $\mathsf{EPRF.Eval}(S_i, (x+1)_{[i]}) = \mathsf{EPRF.Eval}(S_i, x_{[i]}+1) = Q_i(x_{[i]})$ for all $j \le i \le \kappa$.

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 eiO, 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 $\mathsf{EPRF.KeyGen}(1^\kappa)$. 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 pseudo random 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 explode the program G along u. Let the circuit assignment $L = \mathsf{ExplodeTo}(G, u)$. The fragment in L corresponding to u is "**return** $\perp$". Let $G_2 = \mathsf{CanonicalMerge}(L)$. We find $G_2' = \mathsf{eiO.ParaGen}(1^\kappa, G_2)$ and $\mathsf{eiO.ParaGen}(1^\kappa, G_1)$ are indistinguishable since G_1 and G_2 are $(\kappa+1)$-exploding equivalent.
- **Hyb 3** In this hybrid, we will explode 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 look at the details:
  - **Hyb 3.1** In this sub-hybrid, we will 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 explode our original program G into pieces along the path $u = u_0$ and u_1 to get $L_2 = \mathsf{ExplodeTo}(G, \{u_0, u_1\})$. And let $G_3 = \mathsf{CanonicalMerge}(L_2)$. We claim that $\mathsf{eiO.ParaGen}(G_3)$ is indistinguishable from $\mathsf{eiO.ParaGen}(G_2)$ because G_2 and G_3 are $2\kappa+1$-exploding 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 equation comes from 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

using the fragment 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, $\mathsf{ExplodeTo}(Q_m, TC)$ can be constructed from $\mathsf{ExplodeTo}(P_m, TC)$. So now we only care about the circuit assignments $\mathsf{ExplodeTo}(P_m, \{x_1, x_2\})$.

- **Hyb 3.3** In this sub-hybrid, we replace the fragment in $\mathsf{ExplodeTo}(P_{f_0}, \cdot)$ corresponding to t_0 (or in other words, the value $\mathsf{EPRF.Eval}(S_{f_0}, t_0)$) with a uniformly random string r_{f_0,t_0}. The indistinguishability comes from EPRF and PRF security. Also we find that r_{f_0,t_0} only appears in this fragments corresponding to u_1:
  - For any fragment corresponding to $y = u_{0,[l-1]}\neg u_{0,l}$ that $l \leq i$, there does not exist z such that $y||z = t_0$;
  - For any fragment corresponding to $y = u_{0,[l-1]}\neg u_{0,l}$ that $l > i$, we know that $y_l = 0$. So by the definition of f function, for any z, $f(y||z)$ will be at least $l > i$. So the variable t_{f_0} in the function G has been replaced with zeros and $\alpha_{f_0}, \beta_{f_0}$ are computed from $\mathsf{PRG}(\mathsf{EPRF.Eval}(S_{f_0}, t_{0,[f_0-1]}||0))$ instead of from $\mathsf{PRG}(\mathsf{EPRF.Eval}(S_{f_0}, t_0)$ 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 S_{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 $l \leq i$, 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 $l > i$, we know that $y_l = 0$. So by the definition of f function, for any z, $f(y||z)$ will be at least $l > i$. So it only has $\mathsf{PRG}_0(r_{f_0,t_0})$ inside the program.
- **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 $v_{0,0}, v_{0,1} \leftarrow \{0,1\}^\kappa$ by the PRG security property.

  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.
- **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 explode 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 signatures (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* eiO *and one way functions exist, then the* END-OF-LINE *problem is hard for polynomial-time algorithms.*

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.