On the Exact Cryptographic Hardness of Finding a Nash Equilibrium

Sanjam Garg* Omkant Pandey† Akshayaram Srinivasan‡

Abstract

The exact hardness of computing a Nash equilibrium is a fundamental open question in algorithmic game theory. This problem is complete for the complexity class PPAD. It is well known that problems in PPAD cannot be NP-complete unless NP = coNP. Therefore, a natural direction is to reduce the hardness of PPAD to the hardness of problems used in cryptography.

Bitansky, Paneth, and Rosen [FOCS 2015] prove the hardness of PPAD assuming the existence of quasi-polynomially hard indistinguishability obfuscation and sub-exponentially hard one-way functions. This leaves open the possibility of basing PPAD hardness on simpler, polynomially hard, computational assumptions.

We make further progress in this direction and reduce PPAD hardness directly to polynomially hard assumptions. Our first result proves hardness of PPAD assuming the existence of *polynomially hard* indistinguishability obfuscation ($i\mathcal{O}$) and one-way permutations. While this improves upon Bitansky et al.'s work, it does not give us a reduction to simpler, polynomially hard computational assumption because constructions of $i\mathcal{O}$ inherently seems to require assumptions with sub-exponential hardness. In contrast, *public key functional encryption* is a much simpler primitive and does not suffer from this drawback. Our second result shows that PPAD hardness can be based on *polynomially hard* public key functional encryption and one-way permutations. Our results further demonstrate the power of polynomially hard public key functional encryption which is believed to be weaker than indistinguishability obfuscation.

*University of California, Berkeley, `sanjamg@berkeley.edu`
†University of California, Berkeley, `omkant@berkeley.edu`
‡University of California, Berkeley, `akshayaram@berkeley.edu`

1 Introduction

The problem of computing a *Nash equilibrium* is fundamental to algorithmic game theory. The hardness of this problem has attracted significant attention. Since a mixed Nash equilibrium is guaranteed to exist for every game [Nas51], the problem belongs to the complexity class TFNP [MP91]. In a series of works, originating from Papadimitriou [Pap94], the problem was established to be complete for the complexity class PPAD [DGP09, CDT09].

The exact hardness of this problem, however, is still not fully understood. Since the class PPAD is *total*, it is unlikely to contain NP-complete problems unless polynomial hierarchy collapses to the first level [MP91, Pap94]. This is similar to the status of hardness assumptions in cryptography which are not believed to be NP-complete, but nevertheless, hard. Due to this similarity, cryptographic problems were suggested as natural candidates in [Pap94] for studying the hardness of PPAD. Indeed, the hardness of some total super-classes of PPAD, such as PPA and PPP, can already be reduced to "standard" cryptographic problems like factoring and collision-resistant hashing [Jer12]. However, such a reduction is not known for PPAD.

A natural extension of this idea is to consider cryptographic problems with a richer and more powerful structure. One of the richest cryptographic structure is *program obfuscation* as formulated by Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan, and Yang [BGI+12]. It is a compiler to transform any computer program into an "unintelligible one" while preserving its functionality. Ideally, the obfuscation of a program should be a "virtual black-box" (VBB), i.e., access to the obfuscated program should be no better than access to a black-box implementing the program [BGI+12]. Abbot, Kane and Valiant [AKV04] show that PPAD-hardness can be based on VBB obfuscation of a natural pseudo random function. Unfortunately, VBB obfuscation is impossible in general [BGI+12], and there are strong limitations to obfuscating pseudorandom functions [GK05, BCC+14], including the one in [AKV04].

A natural relaxation of VBB obfuscation is *indistinguishability obfuscation* ($i\mathcal{O}$) [BGI+12]. Roughly speaking, $i\mathcal{O}$ guarantees that the obfuscation of a circuit looks indistinguishable from the obfuscation of any another, functionally equivalent, circuit of same size. Starting from the work of Garg, Gentry, Halevi, Raykova, Sahai and Waters [GGH+13b], several candidate constructions [BR14, BGK+14, PST14, GLSW15, Zim15, AB15] for $i\mathcal{O}$ have been suggested based on various assumptions on multilinear maps [GGH13a] and public key functional encryption [AJ15, BV15a, AJS15].

Motivated by the progress on obfuscation, Bitansky, Paneth and Rosen [BPR15] revisit the hardness of PPAD and provide an elegant reduction to the hardness of $i\mathcal{O}$. This is the first reduction of its kind which reduces PPAD-hardness to the security of a concrete and plausible cryptographic primitive. This, together with the progress on $i\mathcal{O}$, gives hope to the possibility of basing PPAD-hardness on simpler, more standard cryptographic primitives.

1.1 Our contribution

In this work, we revisit the problem of reducing PPAD-hardness to rich and expressive cryptographic systems. We build upon the work of [BPR15] with two specific goals:

- **Rely on polynomial-hardness of $i\mathcal{O}$:** One drawback of the BPR reduction is that it requires $i\mathcal{O}$ schemes with at least quasi-polynomial security. It is not clear if such a large loss in the reduction is necessary. Our first goal is to obtain an improved, polynomial time reduction.

- **Rely on simpler, polynomially hard, assumptions:** While tremendous progress has been made on justifying the security of current $i\mathcal{O}$ schemes, ultimately the security of the resulting constructions still either relies on an exponential number of assumptions (basically, one per circuit), or a polynomial set of assumptions with exponential loss in the reduction. Our second goal is thus to completely get rid of $i\mathcal{O}$ or any other component with non-polynomial time flavor, and reduce PPAD-hardness to simpler, polynomially hard, assumptions.

With respect to our first goal, we prove the following theorem:

Theorem 1 *Assuming the existence of polynomially hard one-way permutations and indistinguishability obfuscation for* P/poly*, the* END-OF-LINE *problem is hard for polynomial-time algorithms.*

This polynomially reduces the hardness of PPAD to $i\mathcal{O}$ since PPAD is the class of problems that are reducible to the END-OF-LINE problem.

With respect to our second goal, we show that PPAD-hardness can be reduced to the security of public-key *functional encryption* ($\mathcal{FE}$) in polynomial time.

A public key functional encryption ($\mathcal{FE}$) scheme for general circuits [BSW11, O'N10] is similar to an ordinary (public-key) encryption scheme with the crucial difference that there are many decryption keys, each of which has an associated function f; when an encryption of a message m is decrypted with a key for function f, it decrypts to the value $f(m)$. Our second result proves the following theorem:

Theorem 2 *Assuming the existence of polynomially-hard one-way permutations and compact public key functional encryption for general circuits, the* END-OF-LINE *problem is hard for polynomial-time algorithms.*

Compact functional encryption, as demonstrated by the recent results of Bitansky and Vaikuntanathan [BV15b] and Ananth, Jain and Sahai [AJS15], can be generically constructed from the so called "collusion-resistant function encryption with collusion-succinct ciphertexts," which in turn can be constructed from simpler polynomial hardness assumptions over multi-linear maps, as shown by Garg, Gentry, Halevi, and Zhanrdy [GGHZ16]. This is in sharp contrast to $i\mathcal{O}$ where all constructions still inherently seem to require exponential loss in the security reduction. Combined with the results of [GGHZ16, BV15b, AJS15], theorem 2 bases PPAD-hardness to simpler polynomial time assumptions.

1.2 Our Techniques

We now present a technical overview of our approach. Building upon the work of [BPR15], we will generate hard instances of SVL. We will first show how to generate such instances using polynomially-hard $i\mathcal{O}$ and then discuss how to do the same using polynomially-hard $\mathcal{FE}$.

1.2.1 PPAD Hardness from Indistinguishability Obfuscation

Let us start by recalling the definition of PPAD. The class PPAD consists of search problems that are polynomial time reducible to the END-OF-LINE problem. The hardness of PPAD was proven in [BPR15] by considering a different problem, proposed in [AKV04], called SINK-OF-VERIFIABLE-LINE problem (SVL) in [BPR]. The SVL problem may not be total and hence may not be in PPAD. However, it is reducible to the EOL problem [AKV04, BPR15], and therefore hardness of SVL implies hardness of EOL and PPAD.

Informally, an instance of the SVL problem is specificied by a tuple $(x_s, \mathsf{Succ}, \mathsf{Ver}, T)$ where x_s is called the source node, Succ and Ver are called successor and verification circuits respectively, and T is a target index. Succ defines a directed line graph starting from the source node x_s. A node x is connected to a node y through an outgoing edge if and only if $y = \mathsf{Succ}(x)$. Ver is used to verify whether a given node is the i^{th} node (starting from the source node x_s) on the path defined by Succ. That is, $\mathsf{Ver}(x, i) = 1$ if and only if $x = \mathsf{Succ}^{i-1}(x_s)$. The goal, given the instance, is to find the T-th node on the path. We want to construct an efficiently samplable distribution over instances of SVL for which no polynomial time algorithm can find the T-th node with non-negligible probability.

Bitansky et al., building upon [AKV04], consider a line graph where the i-the node is defined by the output of pseudorandom function (PRF) on i, i.e., the i-th node is (i, σ) such that $\sigma = \mathsf{PRF}_S(i)$ for a randomly chosen key S. Intuitively, σ is a signature on i. The successor circuit of the SVL instance, Succ, is then defined by *obfuscating* a "verify and sign" circuit, VS_S, using general purpose $i\mathcal{O}$; VS_S simply outputs the next point $(i+1, \mathsf{PRF}_S(i+1))$ if the input is a valid point (i, σ) and rejects otherwise. The verification circuit Ver simply tests that a given input will not be rejected by the successor circuit. The source node is given by $(1, \mathsf{PRF}_S(1))$ and the target index T is set to a super-polynomial value in the security parameter.

Intuitively, the hardness of the above instance follows as it is impossible to obtain a signature on a node before obtaining the signature on the previous node in the path. While the underlying idea of this reduction is intuitive, reducing its hardness to $i\mathcal{O}$ is more involved. This is shown by first changing the obfuscated circuit Succ so that it does not behave correctly on a randomly chosen point u, and simply outputs $\perp$. Since the new circuit now does not have to sign at $u+1$, it can be implemented using "punctured version" of S (at point $u+1$) [SW14]. The circuit is then made to output $\perp$ on both u and $u+1$ using "punctured" programming techniques [SW14]. At a high level, this process is then repeated for the next point $u+1$, and then for $u+2$, and so on, until the program does not have the ability to sign on any point in the interval $[u, T]$. Performing this change however requires more care since the number of points in $[u, T]$ is not polynomial. In hindsight, the primary reason for sub-exponential loss in this approach is because it is not possible to "puncture" a large interval in a "single shot." In particular, to be able to use the security of $i\mathcal{O}$, this approach must increase the "punctured" interval by one point at a time.

Our approach: many chains of varying length. Our main ides is to introduce a richer structure to the nodes in the graph, so that it is not necessary to increase the "punctured" interval by one point at a time. Instead, it would be possible to make longer "jumps," sometimes of exponential length, in the proof strategy. This would enable us to make only polynomially many jumps in total to travel from u to T.

More specifically, instead of considering one signature per node, we consider κ signatures for every node where 2^κ is the total number of nodes on the line. That is, a node in our graph is of the form $(i, \sigma_1, \ldots \sigma_\kappa)$ where σ_j is a signature on the first j bits of i computed using a key S_j (different for each index) for every $j \in [\kappa]$. The successor circuit is obfuscation of a program which simply checks each signature on appropriate prefixes of i, and if so, it signs all κ prefixes of $i+1$ using appropriate keys. The verification circuit is as before, the source node is simply the signatures on the first node, i.e, $(0^\kappa, \mathsf{PRF}_{S_1}(0), \ldots, \mathsf{PRF}_{S_\kappa}(0^\kappa))$, and $T = 2^\kappa - 1$. Observe that the BPR reduction is equivalent to having only σ_κ.

We now explain how this structure on the nodes helps us in achieving a polynomial loss in the reduction. As before, we start by changing the successor circuit to output $\perp$ on a random point u. To illustrate the main idea, let us assume that the binary representation of u has k trailing 1s, i.e., u is of the form: $u_1 \cdots u_{\kappa-k-1} \| 01^k$ where $1 \leq k \leq \kappa$. Then, $u + 1 = u_1 \cdots u_{\kappa-k-1} \| 10^k$, i.e., it has k trailing 0s. Observe that:

1. The first $\kappa - k$ prefix bits of $u + 1$ are *identical* to the first $\kappa - k$ prefix bits of *all points* in the interval $[u + 1, u + 2^k]$.
2. Signature $\sigma_{\kappa-k}$ (corresponding to the prefix of length $\kappa - k$) for the node $u + 1$ is not needed (for checking and signing) anywhere else on the line graph except for nodes in the interval $[u + 1, u + 2^k]$.

As before, suppose that we have punctured the successor circuit at a random node u. Then, the fact that the punctured circuit does not output *any* signature on $u + 1$ means that it does not output the signature $\sigma_{\kappa-k}$ on the first $\kappa - k$ bits of $u + 1$; consequently, and most importantly, this means that it does not output this signature on the first $\kappa - k$ bits of *any point in the interval* $[u + 1, u + 2^k]$. This allows us to increase the interval from $[u + 1, u + 2^k]$ by considering only a constant number of hybrids. We then repeat this process by considering $u + 2^k$ as our next point and iterate until we reach T.

Metaphorically, the signatures can be thought of as "chains" emanating from each node and connecting to other nodes. The first chain coming out of a node i is connected to i's immediate neighbor which is $i + 1$. The second chain is connected to a node two hops away from i and the j-th chain is connected to a node 2^j hops away from i and so on. The number of chains coming out from a node i is one more than the number of trailing ones in the binary representation of i. Equivalently, the number of chains coming out of i is the number of bits that change from i to $i + 1$. Puncturing the circuit is viewed as cutting chains of appropriate lengths between points. While BPR strategy always cuts a chain of length 1, our proof strategy cuts the longest possible chain it can and then iterates the process again until it reaches the target T. See Figure 1 for an illustration.

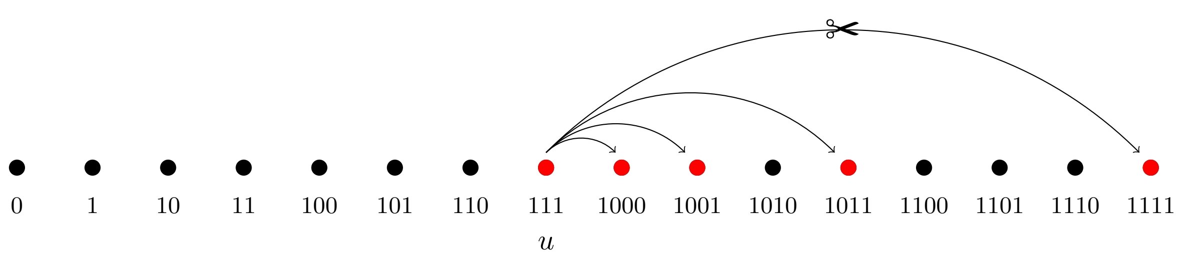

Figure 1: Illustration of cutting a chain for $u = 0111$

While implementing the above idea we face the difficulty that for a random u the number of chains coming out of u could be very small (as small as 1). We get over this difficulty by initially cutting "smaller" length chains until we have the ability to cut "larger" length chains. We show that we need to cut no more than a linear (in the security parameter κ) number of chains to reach T and hence our reduction suffers only a polynomial (in fact linear) loss in the security parameter.

1.2.2 PPAD Hardness from Functional Encryption

We now give a technical overview of our hardness result for PPAD from functional encryption with *polynomial loss*. As noted earlier, although $i\mathcal{O}$ can be reduced to $\mathcal{FE}$ [AJ15, BV15a], we cannot directly rely on this reduction since it suffers sub-exponential security loss. Instead, we try to directly reduce PPAD-hardness to compact $\mathcal{FE}$.

To directly reduce PPAD-hardness to $\mathcal{FE}$, we follow the same approach as before, and generate hard on average instances of SVL using function encryption. It will be sufficient to have *selectively secure single key* $\mathcal{FE}$. To demonstrate the technical challenges while proving the result from $\mathcal{FE}$ we will be considering a single PRF key, as in BPR [BPR15], instead of our idea of using κ keys to implement "multiple chains of varying length." The scenario with a single PRF key already captures the main technical challenges while keeping the exposition simple. Later, we will explain how to combine the two ideas together to obtain a direct polynomial reduction to $\mathcal{FE}$.

The line graph implicitly defined by this successor circuit will be similar to the BPR reduction as before. The successor circuit encodes a pseudo random function $\mathsf{PRF}_S : \{0,1\}^\kappa \to \{0,1\}^\kappa$ in its description. The source node is given by $(0^\kappa, \mathsf{PRF}_S(0^\kappa))$. A node (x, σ) is present on the line graph if and only if $\sigma = \mathsf{PRF}_S(x)$. The successor circuit takes as input (x, σ), checks the validity of the node and if the node is valid outputs $(x+1, \mathsf{PRF}_S(x+1))$. The target index is given by $2^\kappa - 1$.

Our goal is to produce an "obfuscated" (or encrypted) version of this successor circuit using $\mathcal{FE}$. To do this, we will rely on the "binary tree construction" idea of [AJ15, BV15a] for constructing $i\mathcal{O}$ from $\mathcal{FE}$. Note that though this reduction suffers from sub-exponential loss we tailor the construction of our successor circuit so that it suffers only from a polynomial loss.

Binary tree based evaluation [AJ15, BV15a]. Let us quickly recall the main ideas of [AJ15, BV15a]. An indistinguishability obfuscator for a circuit $C : \{0,1\}^\kappa \to \{0,1\}^*$ is a sequence of $\kappa+1$ functional keys $\mathsf{FSK}_1, \cdots, \mathsf{FSK}_{\kappa+1}$ generated using independently sampled master secret keys $MSK_1, \cdots, MSK_{\kappa+1}$ along with a ciphertext c_ϕ encrypting the empty string under public-key PK_1 (corresponding to MSK_1). The function key FSK_i corresponds to a function F_i for all $i \in [\kappa]$ and $\mathsf{FSK}_{\kappa+1}$ corresponds to the circuit C. Intuitively, the circuit F_i is a "bit-extension" function that takes as input a string $x \in \{0,1\}^{i-1}$ and produces functional encryptions of $x\|0$ and $x\|1$ under the public key PK_{i+1}. In the actual construction, the function key FSK_i corresponds to an augumented function $\tilde{F}_i$ which has encryptions of F_i under two independently sampled symmetric keys. The symmetric key used to decrypt the ciphertext containing F_i is propagated along with the input to $\tilde{F}_i$.

To evaluate the obfuscated circuit on an input $x \in \{0,1\}^\kappa$, one does the following: decrypt c_ϕ under FSK_1 to obtain encryptions of 0 and 1. Depending on the bit x_1, choose either the left or right encryption and decrypt it using FSK_2 and so on. Thus, in κ steps one can obtain an encryption of x under $PK_{\kappa+1}$ which can be used to compute $C(x)$ using $\mathsf{FSK}_{\kappa+1}$. One can think of the construction having a binary tree structure where evaluating the circuit on an input x corresponds to traversing along the path labeled x.

Our Construction We will explain our construction through a series of attempts and the problems in those constructions with their corresponding fixes.

First attempt. Our first attempt was to mimic the construction of [AJ15, BV15a] but with the following difference. We do not encrypt the bit extension functions or the final circuit under two independently sampled symmetric keys. This is necessary as changing from one symmetric key to another has to be done across all inputs and this "inherently" incurs a sub-exponential loss in the security of $\mathcal{FE}$ which is what we wanted to avoid in the first place. We generate $2\kappa+1$ functional keys $\mathsf{FSK}_1, \cdots, \mathsf{FSK}_{2\kappa+1}$ where the first 2κ of them correspond to the bit-extension function used for encrypting (x, σ) under $PK_{2\kappa+1}$ and $\mathsf{FSK}_{2\kappa+1}$ corresponds to the circuit Next that checks the validity of the node (x, σ) and outputs the next node in the graph if (x, σ) is valid. The main question with this approach is: How does the circuit Next check the validity of the input node and output the next node in the path? The circuit Next must somehow have access to the PRF key S but this access should not be "visible" to the outside world.

We definitely cannot hardwire the PRF key S in the circuit as the current constructions of public key encryptions do not provide any meaningful notions of "function-privacy". One possible approach is to "propagate" the key S along the entire tree. That is, encrypt the key S in the ciphertext c_ϕ and the bit extension functions output encryptions that also includes S. Though this approach sounds promising we are not able to directly use it as we cannot implement the techniques of "puncturing" the keys that were crucial to the reduction to $i\mathcal{O}$. In order to puncture the key on a string x, we need to puncture the key along every path which incurs a sub-exponential loss which we want to avoid. We modify the approach that enables us to puncture the keys.

Second attempt: "prefix puncturing." To solve the problem explained earlier, we develop techniques to "surgically" puncture the PRF key S along a path x without affecting the distribution on rest of the paths. At a high level, this is implemented by propagating at every node "punctured" versions of the PRF key that can be used to check the validity of the input node as well generate the next node in the path *only* along the subtree rooted at that node. We now explain the details.

Every string $y \in \{0,1\}^{\leq \kappa}$ is associated with a node in the binary tree with the root labeled with the empty string ϕ. We "propagate" a set of punctured keys along every node y such that set of keys appearing at the node y should have the property that it is sufficient to generate the "punctured" keys for any node in the subtree rooted at y. That is, a key punctured at prefix y can be used for computing the PRF value on any leaf appearing in the sub-tree rooted at y. We denote this property as *prefix puncturability*. Also, the punctured keys appearing at the leaves should have the property that it is sufficient to check the validity of the input and for generating the next node in the path. A pseudo random function that has a natural binary tree structure and has the prefix-puncturable property is the construction due to Goldreich, Goldwasser and Micali [GGM86]. We exploit this property in the GGM construction to propagate the "prefix-punctured" keys along the binary tree.

At every node $y \in \{0,1\}^{\leq \kappa}$, we propagate two keys S_y, S_{y+1} where S_y denotes the key S prefix-punctured at string y. Intuitively, S_y is the key used for checking the input node is valid S_{y+1} is used for generating the next node on the path.[1] The bit extension function generates $S_{y\|0}, S_{y\|0+1}$ and $S_{y\|1}, S_{y\|1+1}$ from S_y, S_{y+1} and propagates these values along with $y\|0$ and $y\|1$ respectively. The circuit Next receives S_x, S_{x+1} where $x \in \{0,1\}^\kappa$ and checks the validity of the input signature using S_x and generates the next node in the path if the input is valid using S_{x+1}.

Note that the puncturing of the keys does not happen after the level κ as by this time we have parsed the x which completely determines the the key S_x, S_{x+1}. Therefore, we need to propagate S_x, S_{x+1} along the entire subtree rooted at x where we parse σ. This creates the following problem.

[1] Note that instead of S_{y+1} it is enough to propagate $S_{y+1\|0^{\kappa-|y|}}$. It is in fact crucial for our reduction that we propagate $S_{y+1\|0^{\kappa-|y|}}$ instead of S_{y+1}. But we will use S_{y+1} for ease of notation and exposition.

Recall that the proof strategy of Bitansky et al. is to make the successor circuit output $\perp$ on an random input u for any associated signature σ and then gradually increase the interval at which the successor circuit outputs $\perp$ until we reach the target index. Suppose the successor circuit already outputs $\perp$ on the point x and we are trying to extend the interval to include $x+1$. Recall that the crucial idea behind the ability to increase the interval to include $x+1$ is that S_{x+1} does not occur anywhere else in the computation of the circuit. We observe that S_{x+1} gets propagated along the entire subtree (of exponential size) rooted at x where the input σ is parsed. Hence, to "remove all traces" of S_{x+1} along the subtree rooted at x, we need to incur a sub-exponential loss.

Final construction: "encrypt the next signature." We solve the above problem by changing the honest execution of our successor circuit to an "artificial" one that checks that validity of the input signature in a "hidden" form and outputs an "encrypted" version of the next node that can be decrypted only with a valid input.

Instead of propagating the keys S_x, S_{x+1} in clear in the subtree parsing σ we propagate an "encrypted" version S_{x+1} under S_x. We apply a length doubling injective PRG to the key S_x to obtain two halves $\mathsf{PRG}_0(S_x)$ [2] and $\mathsf{PRG}_1(S_x)$. $\mathsf{PRG}_1(S_x)$ is used as secret key to "encrypt" S_{x+1} and $\mathsf{PRG}_0(S_x)$ is used to check the validity of the input. We notice that with this fix we don't actually need the security of the functional encryption systems where σ is getting parsed and hence we output $\mathsf{PRG}_0(S_x)$ and the encrypted version of S_{x+1} in clear. The Next circuit takes $\sigma, \mathsf{PRG}_0(S_x)$ and the encrypted version of S_{x+1} and checks whether $\mathsf{PRG}_0(\sigma) = \mathsf{PRG}_0(S_x)$ and if yes it decrypts using $\mathsf{PRG}_1(\sigma)$ to obtain S_{x+1}. Notice that while trying to increase the interval we don't suffer from the problem explained earlier as we can change the "encrypted" version of S_{x+1} to encrypt some junk value by relying on the fact that S_x is not needed anywhere else in the computation. This follows since x is already in the interval where the successor circuit outputs $\perp$. Note that in order to enable successor circuit to output $\perp$ at a random point, we "artificially" change the honest execution of the circuit to have a hardwired random value v and the circuit checks if $\mathsf{PRG}(x) = v$ and if so outputs $\perp$. The honest execution does not output $\perp$ for any input x with overwhelming probability since PRG has sparse images. We then change this random v to $\mathsf{PRG}(u)$ for a random u relying on the security of the PRG. A consequence of this fix is that even our honest evaluation of the successor circuit looks somewhat "artificial". This seems necessary to circumvent the sub-exponential loss incurred while constructing obfuscation from functional encryption.

Putting it all together. To show hardness of PPAD from $\mathcal{FE}$ by incurring polynomial loss in the security reduction we need to combine the above ideas with that of "multiple-chains of varying length". As explained in the chain-cutting technique we generate κ GGM keys $S_1, \cdots, S_\kappa$. We propagate the "prefix-punctured" keys corresponding to every index $i \in [\kappa]$ along every node in the binary tree. A careful reader might have noticed that though it is necessary to check the validity of the input signatures for every prefix, it is actually sufficient to generate signatures on the next node on the path only for those bit positions that change when incrementing by 1. This is because for the rest of the bit positions the share the same prefix with the input node and we can just output those input signatures along with those newly computed ones provided the input is valid. This observation is in fact crucial to prove the security of our construction. We need to ensure that the Next circuit must have the ability to check the validity of every signature but it has access only

[2] We actually need a stronger property that the left half of the PRG output being injective and we observe that the construction of PRG from one-way permutations has such a property.

to those prefix punctured keys corresponding to the bit positions that change when incrementing by 1.

We satisfy these two "conflicting" properties by decoupling the process of checking the input signatures and the process of generating the next node on the path. In order to check the input signatures we propagate $\mathsf{PRG}_0(S_{i,x})$ for every $i \in [\kappa]$ and to generate the signatures on the next node on the path we propagate an encrypted version of $S_{j,x+1}$ under $\mathsf{PRG}_1(S_{j,x})$ only for those bits j that change when incrementing x.

2 Preliminaries

2.1 PPAD

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 that a search problem (I_1, R_1) polynomial time reduces to another search 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 $(P(x_1), w_2) \in R_2$, $R_1(x_1, Q(w_2)) = 1$.

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 witnesses w such that $R(x, w) = 1$ is non-empty. The class of total search problems is denoted by TFNP. PPAD [Pap94] is a subset of TFNP and is defined by its complete problem called as END-OF-LINE (abbreviated as EOL).

Definition 3 ([Pap94]) 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* $R_{\mathsf{EOL}}((x_s, \mathsf{Succ}, \mathsf{Pred}), w) = 1$ *iff* $\big(\mathsf{Pred}(\mathsf{Succ}(w)) \neq w\big) \vee \big(\mathsf{Succ}(\mathsf{Pred}(w)) \neq w \wedge w \neq x_s\big)$.

Definition 4 ([Pap94]) *The complexity class* PPAD *is the set of all search problems* (I, R) *such that* $(I, R) \in$ TFNP *and* (I, R) *polynomial time reduces to* EOL.

A related problem to EOL is the SINK-OF-VERIFIABLE-LINE (abbreviated as SVL) which is defined as follows:

Definition 5 ([AKV04, BPR15]) 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* $\big(\mathsf{Ver}(w, T) = 1\big)$.

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. It is easy to observe that for every *valid* SVL instance the set of witness w is not empty. But SVL may not be total since there is no known efficient procedure to test whether the instance is valid or not. But it was shown in [AKV04, BPR15] that SVL polynomial time reduces to EOL.

Lemma 6 ([AKV04, BPR15]) SVL *polynomial time reduces to* EOL.

2.2 Cryptographic Definitions

κ denotes the security parameter. A function $\mu(\cdot) : \mathbb{N} \to \mathbb{R}^+$ is said to be negligible if for all polynomials $\mathsf{poly}(\cdot)$, $\mu(\kappa) < \frac{1}{\mathsf{poly}(\kappa)}$ for large enough κ. For a probabilistic algorithm $\mathcal{A}$, we denote by $\mathcal{A}(x;r)$ the output of $\mathcal{A}$ on input x with the content of the random tape being r. We will omit r when it is implicit from the context. We denote $y \leftarrow \mathcal{A}(x)$ as the process of sampling y from the output distribution of $\mathcal{A}(x)$ with a uniform random tape. For a finite set S, we denote $x \xleftarrow{\$} S$ as the process of sampling x uniformly from the set S. We model non-uniform adversaries $\mathcal{A} = \{\mathcal{A}_\kappa\}$ as circuits such that for all κ, $\mathcal{A}_\kappa$ is of size $p(\kappa)$ where $p(\cdot)$ is a polynomial. We will drop the subscript κ from the adversary's description when it is clear from the context. We will also assume that all algorithms are given the unary representation of security parameter 1^κ as input and will not mention this explicitly when it is clear from the context. We will use PPT to denote Probabilistic Polynomial Time algorithm. We denote $[\kappa]$ to be the set $\{1, \cdots, k\}$. We will use $\mathsf{negl}(\cdot)$ to denote an unspecified negligible function and $\mathsf{poly}(\cdot)$ to denote an unspecified polynomial.

A binary string $x \in \{0,1\}^\kappa$ is represented as $x_1 \cdots x_\kappa$. x_1 is the most significant (or the highest order bit) and x_κ is the least significant (or the lowest order bit). The i-bit prefix $x_1 \cdots x_i$ of the binary string x is denoted by $x_{[i]}$. We use $x \| y$ to denote concatenation of binary strings x and y. We say that a binary string y is a prefix of x if and only if there exists a string $z \in \{0,1\}^*$ such that $x = y \| z$.

Injective Pseudo Random Generator. We give the definition of an injective Pseudo Random Generator PRG.

Definition 7 *An injective pseudo random generator* PRG *is a deterministic polynomial time algorithm with the following properties:*

- ***Expansion:*** *There exists a polynomial $\ell(\cdot)$ (called as the expansion factor) such that for all κ and $x \in \{0,1\}^\kappa$, $|\mathsf{PRG}(x)| = \ell(\kappa)$.*
- ***Pseudo randomness:*** *For all κ and for all poly sized adversaries $\mathcal{A}$,*

$$|\Pr[\mathcal{A}(\mathsf{PRG}(U_\kappa)) = 1] - \Pr[\mathcal{A}(U_{\ell(\kappa)}) = 1]| \leq \mathsf{negl}(\kappa)$$

  where U_i denotes the uniform distribution on $\{0,1\}^i$.
- ***Injectivity:*** *For every κ and for all $x, x' \in \{0,1\}^\kappa$ such that $x \neq x'$, $\mathsf{PRG}(x) \neq \mathsf{PRG}(x')$.*

We in fact need an additional property from an injective PRG. Let us consider PRG where the expansion factor (or the output length) is given by $2 \cdot \ell(\cdot)$. Let us denote the first $\ell(\cdot)$ bits of the output of the PRG by the function PRG_0 and the next $\ell(\cdot)$ bits of the output of the PRG by PRG_1.

Definition 8 *A pseudo random generator* PRG *is said to be left half injective if for every κ and for all $x, x' \in \{0,1\}^\kappa$ such that $x \neq x'$. $\mathsf{PRG}_0(x) \neq \mathsf{PRG}_0(x')$.*

Note that left half injective PRG is also an injective PRG. We note that the standard construction of pseudo random generator for arbitrary polynomial stretch from one-way permutations is left half injective. For completeness, we state the construction:

Lemma 9 *Assuming the existence of one-way permutations, there exists a pseudo random generator that is left half injective.*

Proof Let $f : \{0,1\}^\kappa \to \{0,1\}^\kappa$ be a one-way permutation with hardcore predicate $B : \{0,1\}^\kappa \to \{0,1\}$ [GL89]. Let G be an algorithm defined as follows: On input $x \in \{0,1\}^\kappa$, $G(x) = f^n(x) \| B(x) \| B(f(x)) \cdots B(f^{n-1}(x))$ where $n = 2\ell(\kappa) - \kappa$. Clearly, $|G(x)| = 2\ell(\kappa)$. The pseudo randomness property of $G(\cdot)$ follows from the security of hardcore bit. The left half injectivity property follows from the observation that f^n is a permutation. ■

Puncturable Pseudo Random Function. We recall the notion of puncturable pseudo random function from [SW14]. The construction of pseudo random function given in [GGM86] satisfies the following definition [BW13, KPTZ13, BGI14].

Definition 10 *A puncturable pseudo random function $\mathcal{PRF}$ is a tuple of PPT algorithms* $(\mathsf{KeyGen}_{\mathcal{PRF}}, \mathsf{PRF}, \mathsf{Punc})$ *with the following properties:*

- ***Efficiently Computable:*** *For all κ and for all $S \leftarrow \mathsf{KeyGen}_{\mathcal{PRF}}(1^\kappa)$, $\mathsf{PRF}_S : \{0,1\}^{\mathsf{poly}(\kappa)} \to \{0,1\}^\kappa$ is polynomial time computable.*

- ***Functionality is preserved under puncturing:*** *For all κ, for all $y \in \{0,1\}^\kappa$ and $\forall x \neq y$,*

$$\Pr[\mathsf{PRF}_{S\{y\}}(x) = \mathsf{PRF}_S(x)] = 1$$

  where $S \leftarrow \mathsf{KeyGen}_{\mathcal{PRF}}(1^\kappa)$ and $S\{y\} \leftarrow \mathsf{Punc}(S, y)$.

- ***Pseudo randomness at punctured points:*** *For all κ, for all $y \in \{0,1\}^\kappa$, and for all poly sized adversaries $\mathcal{A}$*

$$|\Pr[\mathcal{A}(\mathsf{PRF}_S(y), S\{y\}) = 1] - \Pr[\mathcal{A}(U_\kappa, S\{y\}) = 1]| \leq \mathsf{negl}(\kappa)$$

  where $S \leftarrow \mathsf{KeyGen}_{\mathcal{PRF}}(1^\kappa)$, $S\{y\} \leftarrow \mathsf{Punc}(S, y)$ and U_κ denotes the uniform distribution over $\{0,1\}^\kappa$.

Indistinguishability Obfuscator. We now define Indistinguishability obfuscator from [BGI$^+$12, GGH$^+$13b].

Definition 11 *A PPT algorithm $i\mathcal{O}$ is an indistinguishability obfuscator for a family of circuits $\{C_\kappa\}_\kappa$ that satisfies the following properties:*

- ***Correctness:*** *For all κ and for all $\mathcal{C} \in C_\kappa$ and for all x,*

$$\Pr[i\mathcal{O}(\mathcal{C})(x) = \mathcal{C}(x)] = 1$$

  where the probability is over the random choices of $i\mathcal{O}$.

- ***Security:*** *For all $\mathcal{C}_0, \mathcal{C}_1 \in C_\kappa$ such that for all x, $\mathcal{C}_0(x) = \mathcal{C}_1(x)$ and for all poly sized adversaries $\mathcal{A}$,*

$$|\Pr[\mathcal{A}(i\mathcal{O}(\mathcal{C}_0)) = 1] - \Pr[\mathcal{A}(i\mathcal{O}(\mathcal{C}_1)) = 1]| \leq \mathsf{negl}(\kappa)$$

Functional Encryption. We recall the notion of functional encryption with selective indistinguishability based security [BSW11, O'N10].

A functional encryption $\mathcal{FE}$ is a tuple of PPT algorithms ($\mathsf{FE.Setup}, \mathsf{FE.Enc}, \mathsf{FE.KeyGen}, \mathsf{FE.Dec}$) with the message space $\{0,1\}^*$ having the following syntax:

- $\mathsf{FE.Setup}(1^\kappa)$: Takes as input the unary encoding of the security parameter κ and outputs a public key PK and a master secret key MSK.
- $\mathsf{FE.Enc}_{PK}(m)$: Takes as input a message $m \in \{0,1\}^*$ and outputs an encryption C of m under the public key PK.
- $\mathsf{FE.KeyGen}(MSK, f)$: Takes as input the master secret key MSK and a function f (given as a circuit) as input and outputs the function key FSK_f.
- $\mathsf{FE.Dec}(\mathsf{FSK}_f, C)$: Takes as input the function key FSK_f and the ciphertext C and outputs a string y.

Definition 12 (Correctness) *The functional encryption scheme $\mathcal{FE}$ is correct if for all κ and for all messages $m \in \{0,1\}^*$,*

$$\Pr\left[y = f(m) \middle| \begin{array}{l} (PK, MSK) \leftarrow \mathsf{FE.Setup}(1^\kappa) \\ C \leftarrow \mathsf{FE.Enc}_{PK}(m) \\ \mathsf{FSK}_f \leftarrow \mathsf{FE.KeyGen}(MSK, f) \\ y \leftarrow \mathsf{FE.Dec}(\mathsf{FSK}_f, C) \end{array} \right] = 1$$

Definition 13 (Selective Security) *For all κ and for all poly sized adversaries $\mathcal{A}$,*

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

where $\mathsf{Expt}_{1^\kappa,b,\mathcal{A}}$ is defined below:

- ***Challenge Message Queries:*** *The adversary $\mathcal{A}$ outputs two messages m_0, m_1 such that $|m_0| = |m_1|$ to the challenger.*
- *The challenger samples $(PK, MSK) \leftarrow \mathsf{FE.Setup}(1^\kappa)$ and generates the challenge ciphertext $C \leftarrow \mathsf{FE.Enc}_{PK}(m_b)$. It then sends (PK, C) to $\mathcal{A}$.*
- ***Function Queries****: $\mathcal{A}$ submits function queries f to the challenger. The challenger responds with $\mathsf{FSK}_f \leftarrow \mathsf{FE.KeyGen}(MSK, f)$.*
- *If $\mathcal{A}$ makes a query f to functional key generation oracle such that $f(m_0) \neq f(m_1)$, output of the experiment is $\perp$. Otherwise, the output is b' which is the output of $\mathcal{A}$.*

Remark 14 *We say that the functional encryption scheme $\mathcal{FE}$ is **single-key, selectively secure** if the adversary $\mathcal{A}$ in $\mathsf{Expt}_{1^\kappa,b,\mathcal{A}}$ is allowed to query the functional key generation oracle $\mathsf{FE.KeyGen}(MSK, \cdot)$ on a single function f.*

Definition 15 (Compactness, [AJS15, BV15a, AJ15]) *The functional encryption scheme $\mathcal{FE}$ is said to be compact if for all $\kappa \in \mathbb{N}$ and for all $m \in \{0,1\}^*$ the running time of the encryption algorithm $\mathsf{FE.Enc}$ is $\mathsf{poly}(\kappa, |m|)$.*

Bitansky et al. in [BV15b] and Ananth et al. in [AJS15] show a generic transformation from any collusion-resistant $\mathcal{FE}$ for general circuits where the ciphertext size is independent of the number of collusions (but may depend arbitrarily on the circuit parameters) to a compact $\mathcal{FE}$ for general circuits. The property that the ciphertext size does not depend on the number of collusion is referred as *collusion-succinctness.*

Lemma 16 ([BV15b, AJS15]) *Assuming the existence of selectively secure collusion-resistant functional encryption with collusion-succinct ciphertexts, there exists a selectively secure compact functional encryption scheme.*

Symmetric Key Encryption. A Symmetric-Key Encryption scheme $\mathcal{SKE}$ is a tuple of algorithms $(\mathsf{SK.KeyGen}, \mathsf{SK.Enc}, \mathsf{SK.Dec})$ with the following syntax:

- $\mathsf{SK.KeyGen}(1^\kappa)$: Takes as input an unary encoding of the security parameter κ and outputs a symmetric key SK.
- $\mathsf{SK.Enc}_{SK}(m)$: Takes as input a message $m \in \{0,1\}^*$ and outputs an encryption C of the message m under the symmetric key SK.
- $\mathsf{SK.Dec}_{SK}(C)$: Takes as input a ciphertext C and outputs a message m'.

We say that $\mathcal{SKE}$ is *correct* if for all κ and for all messages $m \in \{0,1\}^*$, $\Pr[\mathsf{SK.Dec}_{SK}(C) = m] = 1$ where $SK \leftarrow \mathsf{SK.KeyGen}(1^\kappa)$ and $C \leftarrow \mathsf{SK.Enc}_{SK}(m)$.

Definition 17 *For all κ and for all polysized adversaries $\mathcal{A}$,*

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

where $\mathsf{Expt}_{1^\kappa,b,\mathcal{A}}$ is defined below:

- ***Challenge Message Queries:*** *The adversary $\mathcal{A}$ outputs two messages m_0 and m_1 such that $|m_0| = |m_1|$ for all $i \in [n]$.*
- *The challenger samples $SK \leftarrow \mathsf{SK.KeyGen}(1^\kappa)$ and generates the challenge ciphertext C where $C \leftarrow \mathsf{SK.Enc}_{SK}(m_b)$. It then sends C to $\mathcal{A}$.*
- *Output is b' which is the output of $\mathcal{A}$.*

Prefix Puncturable Pseudo Random Functions. We now define the notion of prefix puncturable pseudo random function PPRF which is satisfied by the construction of the pseudo random function in [GGM86].

Definition 18 *A prefix puncturable pseudo random function $\mathcal{PPRF}$ is a tuple of PPT algorithms* $(\mathsf{KeyGen}_{\mathcal{PPRF}}, \mathsf{PrefixPunc})$ *satisfying the following properties:*

- ***Functionality is preserved under* repeated *puncturing:*** *For all κ, for all $y \in \cup_{k=0}^{\mathsf{poly}(\kappa)}\{0,1\}^k$ and for all $x \in \{0,1\}^{\mathsf{poly}(\kappa)}$ such that there exists a $z \in \{0,1\}^*$ s.t. $x = y\|z$,*

$$\Pr[\mathsf{PrefixPunc}(\mathsf{PrefixPunc}(S,y),z) = \mathsf{PrefixPunc}(S,x)] = 1$$

where $S \leftarrow \mathsf{KeyGen}_{\mathcal{PPRF}}$.

- ***Pseudorandomness at punctured prefix:*** *For all κ, for all $x \in \{0,1\}^{\mathsf{poly}(\kappa)}$, and for all poly sized adversaries $\mathcal{A}$*

$$|\Pr[\mathcal{A}(\mathsf{PrefixPunc}(S,x),\mathsf{Keys})=1]-\Pr[\mathcal{A}(U_\kappa,\mathsf{Keys})=1]|\leq \mathsf{negl}(\kappa)$$

where $S \leftarrow \mathsf{KeyGen}_{\mathcal{PRF}}(1^\kappa)$ and $\mathsf{Keys}=\{\mathsf{PrefixPunc}(S,x_{[i-1]}\|(1-x_i))\}_{i\in[\mathsf{poly}(\kappa)]}$.

3 Hardness from Indistinguishability Obfuscation

In this section, we prove that SVL is hard on average assuming polynomial hardness of indistinguishability obfuscation, injective PRGs and puncturable pseudo random functions. Coupled with the fact that SVL reduces to EOL (Lemma 7) we have the following theorem.

Theorem 19 *Assume the existence of one-way permutations and indistinguishability obfuscation against polynomial time adversaries then we have that EOL problem is hard for polynomial time algorithms.*

3.1 Hard on Average SVL Instances

In this section, we describe an efficient sampler that provides hard on average instances $(x_s, \mathsf{Succ}, \mathsf{Ver}, 1^\kappa)$ of SVL. Here x_s is the source node and Succ is the successor circuit. We define 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 k^{th} node from x_s. That is, $\mathsf{Ver}(x,k)=1$ iff $x=\mathsf{Succ}^{k-1}(x_s)$. For the generated instances, we argue that it is hard to find the 1^κ node in the path.

The formal description of hard on average SVL instance sampler is provided in Figure 3. Internally this sampler generates an obfuscation of the Next circuit provided in Figure 2. Next we describe the SVL instances which we consider informally.

The instance we generate defines a line graph. The nodes in the graph are of the form: $(x,\sigma_1,\cdots,\sigma_\kappa)$ where $x\in\{0,1\}^\kappa$. The nodes satisfy the following relation: for all $i\in[\kappa]$, $\mathsf{PRF}_{S_i}(x_{[i]})=\sigma_i$ and in that case we say that $(x,\sigma_1,\cdots,\sigma_\kappa)$ is *valid*. The node $(x,\sigma_1,\cdots,\sigma_\kappa)$ is connected to $(x+1,\sigma'_1,\cdots,\sigma'_\kappa)$ through an outgoing edge and is connected to $(x-1,\sigma''_1,\cdots,\sigma''_\kappa)$ through an incoming edge where $\sigma'_1,\cdots,\sigma'_\kappa$ and $\sigma''_1,\cdots,\sigma''_\kappa$ satisfy the above described PRF relationship. The source node is given by $(0^\kappa,\mathsf{PRF}_{S_1}(0),\cdots,\mathsf{PRF}_{S_\kappa}(0^\kappa))$.

At a very high level successor circuit of our SVL instances provides a method for moving forward from one node to the next. The successor circuit in our instances corresponds to an obfuscation of the Next circuit. This circuit on input a node of the form $(x,\sigma_1,\cdots,\sigma_\kappa)$ checks for the validity of the input. If it is valid, it outputs the next node $(x+1,\sigma'_1\cdots\sigma'_\kappa)$ where $\sigma'_i=\mathsf{PRF}_{S_i}((x+1)_{[i]})$ in the path. On an invalid input, it outputs $\perp$.

For the hard SVL instances we additionally need to provide a verification circuit. The verification circuit just uses the successor circuit in a very natural manner. The verification circuit on input $(x,\sigma_1,\cdots,\sigma_\kappa,j)$ outputs 1 if and only if $x=j-1$ and $\mathsf{Next}_{S_1,\cdots,S_\kappa}(x,\sigma_1,\cdots,\sigma_\kappa)\neq\perp$.

3.2 Proof of Hardness

We start by showing that our sampler generates SVL instances, satisfying constraints of Definition 6. It suffices to show that the verification circuit Ver on input $((x,\sigma_1,\cdots,\sigma_\kappa),j)$ outputs 1 if and only if $(x,\sigma_1,\cdots,\sigma_\kappa)=\mathsf{Succ}^{j-1}(x_s)$. This follows immediately from our construction. Our verification

Input: $(x, \sigma_1, \cdots, \sigma_\kappa)$
Hardcoded Parameters: $S_1, \cdots, S_\kappa$

1. For any $i \in [\kappa]$, if $\sigma_i = \mathsf{PRF}_{S_i}(x_{[i]})$ the output $\perp$.
2. If $i = 1^\kappa$, then output SOLVED.
3. Else output $(x+1, \sigma'_1, \cdots, \sigma'_\kappa)$, where for all $i \in [\kappa]$ compute $\sigma'_i = \mathsf{PRF}_{S_j}((x+1)_{[i]})$.

Padding: This circuit is padded so that total size of the circuit is $p(\kappa)$, for some polynomial $p(\cdot)$ specified later.

Figure 2: $\mathsf{Next}_{S_1,\cdots,S_\kappa}$

- **Sampled Ingredients:** Sample $\{S_i\}_{i\in[\kappa]} \leftarrow \mathsf{KeyGen}_{\mathcal{PRF}}(1^\kappa)$. For all $i \in [\kappa]$, S_i is a seed for a PRF mapping i bits to κ bits. That is, $\mathsf{PRF}_{S_i} : \{0,1\}^i \to \{0,1\}^\kappa$.
- **Source Node**: The source node $x_s = (0^\kappa, \mathsf{PRF}_{S_1}(0), \cdots, \mathsf{PRF}_{S_\kappa}(0^\kappa))$.
- **Successor Circuit**: The successor circuit is given by $i\mathcal{O}(\mathsf{Next}_{S_1,\cdots,S_\kappa})$ where the circuit $\mathsf{Next}_{S_1,\cdots,S_\kappa}$ is described in Figure 2.
- **Verification Circuit**: The verification circuit, given by Ver, on input $((x, \sigma_1 \cdots \sigma_\kappa), j)$ checks if $x = j-1$ and $i\mathcal{O}(\mathsf{Next}_{S_1,\cdots,S_\kappa})((x, \sigma_1 \cdots \sigma_\kappa)) \neq \perp$.

Figure 3: Sampler for hard on average instances of SVL based on hardness of $i\mathcal{O}$

circuit Ver outputs 1 if and only if $x = j-1$ and $\mathsf{Next}_{S_1,\cdots,S_\kappa}(x, \sigma_1, \cdots, \sigma_\kappa) \neq \perp$. The later requirement means implies that $(x, \sigma_1, \cdots, \sigma_\kappa)$ must be valid. Therefore, by design, we have that $(x, \sigma_1, \cdots, \sigma_\kappa) = \mathsf{Succ}^{j-1}(x_s)$.

Next we show that no polynomial time adversary can output a valid value $(1^\kappa, \sigma_1, \cdots, \sigma_\kappa)$. We will show this via a hybrid argument. Starting with providing SVL instance as in distribution from Figure 3, namely hybrid Hyb_0, we move to a hybrid $\mathsf{Hyb}_{4,\delta(u_0)}$ where $\delta(u_0) \leq \kappa$ is a number specified later. In the final hybrid, the successor circuit returns $\perp$ on all inputs of the form $(1^\kappa, \cdot, \cdots, \cdot)$. Observe that the advantage of any adversary in solving the SVL instance in this final hybrid is 0. This proves our claim. We make this change by going through a polynomial (in fact linear) in the security parameter number of intermediate hybrids.

Circuit Next^*. In our proof we will extensively use the circuit $\mathsf{Next}^*_{S_1,\cdots,S_\kappa,u,u'}$ which is a modification of $\mathsf{Next}_{S_1,\cdots,S_\kappa}$ again padded to size $p(\kappa)$. The circuit $\mathsf{Next}^*_{S_1,\cdots,S_\kappa,u,u'}$ is identical to $\mathsf{Next}_{S_1,\cdots,S_\kappa}$ except that on inputs $(x, \cdot, \cdots, \cdot)$ such that $u \leq x \leq u'$ it outputs $\perp$.

Our hybrids. Next we describe our hybrids.

- Hyb_0: This hybrid corresponds to SVL instance generation as given in Figure 3.

- Hyb_1: In the hybrid we change how the successor circuit is generated. In particular, the successor circuit is generated as $i\mathcal{O}(\mathsf{Next}^1_{S_1,\cdots,S_\kappa,v})$ instead of $i\mathcal{O}(\mathsf{Next}_{S_1,\cdots,S_\kappa})$. The circuit $\mathsf{Next}^1_{S_1,\cdots,S_\kappa,v}$ is identical to $\mathsf{Next}_{S_1,\cdots,S_\kappa}$ except that on input $(x,\cdot,\cdots,\cdot)$ such that $\mathsf{PRG}(x) = v$, it outputs $\perp$. (Again this circuit is padded to size $p(\kappa)$.) The value v itself is chosen uniformly from $\{0,1\}^{2\kappa}$.

  Since v is chosen uniformly at random, with overwhelming probability we have that for all $x \in \{0,1\}^\kappa$ $\mathsf{PRG}(x) \neq v$. Hence the circuits $\mathsf{Next}_{S_1,\cdots,S_\kappa}$ and $\mathsf{Next}^1_{S_1,\cdots,S_\kappa,v}$ are functionally equivalent with overwhelming probability. Therefore, indistinguishability obfuscation implies computational indistinguishability between Hyb_0 and Hyb_1.

- Hyb_2: In this hybrid we change how the value v, hard-coded in $\mathsf{Next}^1_{S_1,\cdots,S_\kappa,v}$ is generated. In particular, instead of sampling v uniformly at random from $\{0,1\}^{2\kappa}$, we generate v as $\mathsf{PRG}(u_0)$ where u_0 is sampled uniformly from $\{0,1\}^\kappa$. Here PRG is a length doubling injective pseudorandom generator.

  The indistinguishability between Hyb_1 and Hyb_2 follows from the pseudorandomness property of the PRG.

- Hyb_3: In this hybrid we change how the successor circuit is generated. Recall that in hybrid Hyb_2, the successor circuit was the obfuscated circuit $i\mathcal{O}(\mathsf{Next}^1_{S_1,\cdots,S_\kappa,\mathsf{PRG}(u_0)})$. In Hyb_3 we change it to the obfuscated circuit $i\mathcal{O}(\mathsf{Next}^*_{S_1,\cdots,S_\kappa,u_0,u_0})$.

  Note that the circuits $\mathsf{Next}^1_{S_1,\cdots,S_\kappa,\mathsf{PRG}(u_0)}$ and $\mathsf{Next}^*_{S_1,\cdots,S_\kappa,u_0,u_0}$ are functionally equivalent because of injectivity of the PRG. Hence, indistinguishability obfuscation implies computational indistinguishability between Hyb_2 and Hyb_3.

- $\mathsf{Hyb}_{4,j}$: In hybrid $\mathsf{Hyb}_{4,j}$ for $j \in \{0,\ldots,\delta(u_0)\}$, the successor circuit is generated as an obfuscation of the circuit $\mathsf{Next}^*_{S_1,\cdots,S_\kappa,u_0,u_j}$ where u_j values are described below. Here $\delta(u_0)$ is the number of 0 bits in the binary representation of u_0.

  Defining u_j values. For any string $u \in \{0,1\}^\kappa$, let $f(u)$ denote the index of the lowest order bit of u that is 0 (with the index of the highest order bit being 1). More formally, $f(u)$ is the largest j such that $u = u_{[j]}||1^{\kappa-j}$. For example, if $u = \overbrace{100}^{3}11$ then $f(u) = 3$. Also let $\delta(u)$ be the number of 0 bits in the binary representation of u.

  Starting with a value $u_0 \in \{0,1\}^\kappa$ we define a sequence of values such that u_{j+1} is the value u_j with the $f(u_j)^{th}$ bit set to 1. More formally, $u_{j+1} = u_j + 2^{\kappa-f(u_j)}$. It is easy to see that for all $j \in \{0,\ldots,\delta(u_0)\}$ we have that $\delta(u_{j+1}) < \delta(u_j)$ and $u_{\delta(u_0)} = 1^\kappa$. The sequence of u_i values starting with $u_0 = 0010$ are illustrated in Figure 4.

  Indistinguishability Argument. Observe that the hybrid $\mathsf{Hyb}_{4,0}$ is identical to hybrid Hyb_3 and $\mathsf{Hyb}_{4,\delta(u_0)}$ is such that the successor outputs $\perp$ on all inputs of the form $(1^\kappa,\cdot,\ldots,\cdot)$. Consequently, no adversary can solve the SVL instance in this final hybrid with probability better than 0. Hence, in order to complete the proof, it suffices to argue that the hybrids

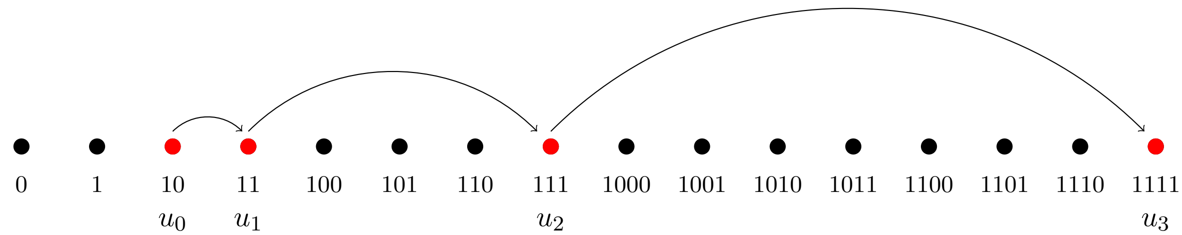

Figure 4: Illustration of the steps starting with $u_0 = 0010$.

$\mathsf{Hyb}_{4,j}$ and $\mathsf{Hyb}_{4,j+1}$ are computationally indistinguishable for each $j \in \{0, \ldots \delta(u_0) - 1\}$. We argue this next.

Indistinguishability between $\mathsf{Hyb}_{4,j}$ and $\mathsf{Hyb}_{4,j+1}$. We prove indistinguishability via sequence of sub-hybrids, where in each successive hybrid we make a small change to the successor circuit. We let f_j as the shorthand for $f(u_j)$ and $t_j = u_{j[f_j]} + 1$.

- $\mathsf{Hyb}_{4,j,1}$: Let $S'_{f_j} = \mathsf{Punc}(S_{f_j}, t_j)$ and let $\sigma^\star = \mathsf{PRF}_{S_{f_j}}(t_j)$. Replace the successor circuit from an obfuscation of $\mathsf{Next}^*_{S_1,\cdots,S_\kappa,u_0,u_j}$ to an obfuscation of $\mathsf{Next}^2_{S_1,\cdots,S'_{f_j},\cdots,S_\kappa,u_0,u_j,\sigma^\star}$.

  $\mathsf{Next}^2_{S_1,\cdots,S'_{f_j},\cdots,S_\kappa,u_0,u_j,\sigma^\star}$ is identical to $\mathsf{Next}^*_{S_1,\cdots,S_\kappa,u_0,u_j}$ except that it cannot compute $\mathsf{PRF}_{S_{f_j}}(t_j)$ because it is provided with a punctured key S'_{f_j}. However, the exact same value $\sigma^\star$ is hard-wired in it which it uses whenever needed instead of computing $\mathsf{PRF}_{S_{f_j}}(t_j)$. (The circuit is appropriately padded to size $p(\kappa)$.)

  Computational indistinguishability between hybrids $\mathsf{Hyb}_{4,j}$ and $\mathsf{Hyb}_{4,j,1}$ follows from the security of the indistinguishability obfuscation scheme.

- $\mathsf{Hyb}_{4,j,2}$: The successor circuit is still an obfuscation of $\mathsf{Next}^2_{S_1,\cdots,S'_{f_j},\cdots,S_\kappa,u_0,u_j,\sigma^\star}$ with $S'_{f_j} = \mathsf{Punc}(S_{f_j}, t_j)$ just as in hybrid $\mathsf{Hyb}_{4,j,1}$. However, instead of generating $\sigma^\star = \mathsf{PRF}_{S_{f_j}}(t_j)$, we sample $\sigma^\star \leftarrow \{0,1\}^\kappa$. (The circuit is appropriately padded to size $p(\kappa)$.)

  Computational indistinguishability between hybrids $\mathsf{Hyb}_{4,j,1}$ and $\mathsf{Hyb}_{4,j,2}$ follows from the pseudorandom at punctured point property of the puncturable PRF.

  Important Observations. Let's understand the role of $\sigma^\star$ in the circuit obfuscated above.

  Observe that the successor circuit on input $(x, \sigma_1, \ldots, \sigma_{f_j}, \ldots, \sigma_\kappa)$ checks if $\sigma_{f_j} = \sigma^\star$ for all $x \in \{u_j + 1, \ldots, u_{j+1}\}$. Furthermore, if this check passes then $\sigma^\star$ is output for all $x \in \{u_j + 1, \ldots, u_{j+1} - 1\}$.

  Next note that $\sigma^\star$ is not used for any other purpose. This relies on the fact that in this hybrid the successor is set to output $\perp$ when $x = u_j$. At this point it is instructive to recall that $x = u_j$ is the only other input on which the successor circuit in $\mathsf{Hyb}_{4,j-1}$ was expected to output this value without being providing the same value in the input.

- $\mathsf{Hyb}_{4,j,3}$: In this circuit instead of generating the successor circuit as an obfuscation of $\mathsf{Next}^2_{S_1,\cdots,S'_{f_j},\cdots,S_\kappa,u_0,u_j,\sigma^\star}$ we generate it as an obfuscation of $\mathsf{Next}^3_{S_1,\cdots,S'_{f_j},\cdots,S_\kappa,u_0,u_j,\tau^\star}$ where

$\tau^* = \mathsf{PRG}(\sigma^\star)$. Note that Next^2 used $\sigma^\star$ to only check if $\sigma_{f_j} = \sigma^\star$ for certain choices of inputs. In these cases, Next^3 is modified to check if $\mathsf{PRG}(\sigma_{f_j}) = \tau^\star$ and outputs the provided input value σ_{f_j} if the check passes. (The circuit is appropriately padded to size $p(\kappa)$.)

Note that by the injectivity property of the PRG we have that the circuits considered here are functionally equivalent and hence computational indistinguishability between hybrids $\mathsf{Hyb}_{4,j,2}$ and $\mathsf{Hyb}_{4,j,3}$ follows from the security of indistinguishability obfuscation.

- $\mathsf{Hyb}_{4,j,4}$: In this hybrid we still obfuscate $\mathsf{Next}^3_{S_1,\cdots,S'_{f_j},\cdots,S_\kappa,u_0,u_j,\tau^\star}$ but instead of generating $\tau^* = \mathsf{PRG}(\sigma^\star)$, we generate $\tau^\star$ as a random string in $\{0,1\}^{2\kappa}$. (The circuit is appropriately padded to size $p(\kappa)$.)

  Indistinguishability between $\mathsf{Hyb}_{4,j,3}$ and $\mathsf{Hyb}_{4,j,4}$ follows from the security of the PRG.

- $\mathsf{Hyb}_{4,j,5}$: In this hybrid we change the successor circuit to now be an obfuscation of $\mathsf{Next}^3_{S_1,\cdots,S'_{f_j},\cdots,S_\kappa,u_0,u_{j+1},\tau^\star}$.(The circuit is appropriately padded to size $p(\kappa)$.)

  Observe that with overwhelming probability the circuits obfuscated in hybrids $\mathsf{Hyb}_{4,j,4}$ and $\mathsf{Hyb}_{4,j,5}$ are functionally equivalent and hence computational indistinguishability follows from indistinguishability obfuscation.

- $\mathsf{Hyb}_{4,j,6}$: Instead of using the puncture key S'_{f_j} we use the unpunctured key S_{f_j}. More specifically, we generate the sampler as an obfuscation of $\mathsf{Next}^*_{S_1,\cdots,S_\kappa,u_0,u_{j+1}}$.

  Again computational indistinguishability follows by indistinguishability obfuscation.

We set $p(\cdot)$ to be the least polynomial such that all circuits considered in the construction and the proof are of size at most $p(\kappa)$. Note that hybrid $\mathsf{Hyb}_{4,j,6}$ is same as $\mathsf{Hyb}_{4,j}$. This concludes the proof.

4 Hardness Result based on Functional Encryption

In this section we show that SVL is hard on average assuming polynomially hard functional encryption and one-way permutations. Coupled with the fact that SVL reduces to EOL (Lemma 7) we have the following theorem.

Theorem 20 *Assume the existence of one-way permutations and functional encryption against polynomial time adversaries then we have that* EOL *problem is hard for polynomial time algorithms.*

Recall that hard SVL instance based on $i\mathcal{O}$, required κ puncturable PRF keys. The ability to puncture the keys was used to prove the hardness from polynomially hard $i\mathcal{O}$. Basing hardness on polynomially hard functional encryption requires us to still maintain κ keys. However, now we need to use prefix-puncturing (see Definition 19) which is more delicate and needs to be handled carefully. Consequently the construction ends up being complicated. However, the special mechanism of prefix-puncturing that we use is crucial to understanding our construction. So towards simplifying exposition, we start by abstracting out the details of this puncturing and present a special tree structure and some properties about it next.

4.1 Special Tree Key Structure

Let $x_{[i]}$ denote the first i (higher order) bits of x i.e $x_1 \cdots x_i$. Now note that any $y \in \{0,1\}^i$ can be identified with a node in a binary tree for which nodes at depth i correspond to strings $\{0,1\}^i$. Note that the root of the tree corresponds to the empty string ϕ. As previously mentioned our construction needs κ PPRF keys, namely $S_1, \ldots S_\kappa$. The key S_i works on inputs of length i. We use $S_{i,x}$ to denote the key S_i prefix punctured at a string $x \in \{0,1\}^{\leq i}$.

Looking ahead, in our hard-on-average instances of SVL each $x \in \{0,1\}^\kappa$ will be attached with *associated signature* values $\sigma_1, \ldots, \sigma_\kappa$ where for each $i \in [\kappa]$ we have that $\sigma_i = \mathsf{PrefixPunc}(S_i, x_{[i]})$. Furthermore in our construction given x and the associated signature values, we will need to verify these values and provide the associated signature values for $x+1$, but this has to be done in a circuitous manner because of several security reasons. We do not delve into the security arguments right away, but focus on describing the prefix-puncturing that we need to perform.

We next describe the set V_x^i where $x \in \{0,1\}^{\leq i}$, which contains suitable prefix-puncturings of the key S_i. Intuitively, we want this set to contain all keys that will allow us to perform the task of *checking the validity* of the i^{th} associated signature on any input of the form $x\|y$ where $y \in \{0,1\}^{\kappa-|x|}$ as well as *computing* the i^{th} associated signature for $(x\|y)+1$. Furthermore, it should suffice to generate $\mathsf{V}_{x\|y}^i$ for all y. For any node $x \in \{0,1\}^{\leq i}$, this very naturally translates to the keys $S_{i,x}$ and $S_{i,x+1}$. A careful reader might have noticed that instead of $S_{i,x+1}$, it in fact suffices to just have $S_{i,(x+1)\|0^{i-|x|}}$. As it turns out we must only include $S_{i,(x+1)\|0^{i-|x|}}$. Including $S_{i,x+1}$ prevents the Derivability Lemma (Lemma 23) from going through.

Recall that the key S_i corresponds to a PPRF key for inputs of length i. Therefore, for $x\|y$ such that $|x| = i$, the key S_i can be prefix-punctured only for the prefix $x = (x\|y)_{[i]}$. This raises the following question. Should we include $S_{i,x}$ and $S_{i,x+1}$ in all $\mathsf{V}_{x\|y}^i$? As we will see later, in our construction, we carefully decouple the checking of associated signatures from the generation of new associated signatures. An important consequence, relevant here is that, even though the checks need to be performed for all $x\|y$, a new i^{th} associated signature needs to be generated for only one choice of y, namely $1^{\kappa-|x|}$ (the all 1 string of length $\kappa - |x|$). This design choice also allows us to set $\mathsf{V}_{x\|y}^i$ for all other choices of y to be $\emptyset$. In terms of the binary tree structure one can think of this as V_x^i getting passed only along the rightmost path in the subtree rooted at x. At a very high level, this allows us to argue that the key S_i (proved formally in Lemma 23) can be punctured at a special point by removing keys fron V_x^i for only a polynomial number of choices of x and i. This is crucial for ensuring that our proof of security has only a polynomial number of hybrids.

Next note that dropping keys from $V_{x\|y}^i$ (such that $|x| = i$) hinders the checking of associated signatures provided along with inputs $x\|y$ where $y \neq 1^{\kappa-i}$. We tackle this issue by introducing a vestigial set $\mathsf{W}_{x\|y}^i$ corresponding to each $\mathsf{V}_{x\|y}^i$. This vestigial set contains remnants of the keys that were dropped from V_x^i. We craft these remnants to be such that they suffice for performing the necessary checks. In particular, we set these remnants to be the left half of an left half injective PRG evaluation on the dropped key. More formally, V_x^i and V_x are defined as follows. In the following, for any $i \in [\kappa]$ we treat $1^i + 1$ as 1^i, and $\phi + 1$ as ϕ. Here 1^i is a string of i 1s and ϕ is

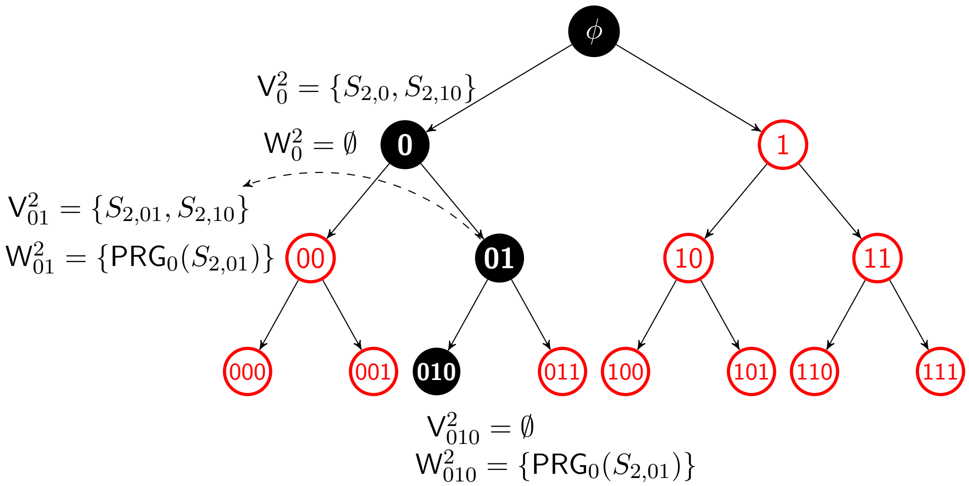

Figure 5: Example of values contained in V_x^2 for $x \in \{0,1\}^{\leq 3}$.

the empty string.

$$\mathsf{V}_x = \bigcup_{i \in [\kappa]} \mathsf{V}_x^i \qquad \mathsf{V}_x^i = \begin{cases} \{S_{i,x_{[i]}}, S_{i,x_{[i]}+1}\} & \text{if } |x| > i \text{ and } x = x_{[i]} \| 1^{|x|-i} \\ \{S_{i,x}, S_{i,(x+1)\|0^{i-|x|}}\} & \text{if } |x| \leq i \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{W}_x = \bigcup_{i \in [\kappa]} \mathsf{W}_x^i \qquad \mathsf{W}_x^i = \begin{cases} \{\mathsf{PRG}_0(S_{i,x_{[i]}})\} & \text{if } |x| \geq i \\ \emptyset & \text{otherwise} \end{cases}$$

For the empty string $x = \phi$, these sets can be initialized as follows.

$$\mathsf{V}_\phi = \bigcup_{i \in [\kappa]} \mathsf{V}_\phi^i \qquad \mathsf{V}_\phi^i = \{S_i\}$$

$$\mathsf{W}_\phi = \bigcup_{i \in [\kappa]} \mathsf{W}_\phi^i \qquad \mathsf{W}_\phi^i = \emptyset$$

Illustration with an example. Finally we explain what sets $\mathsf{V}_x^2, \mathsf{W}_x^2$ contain when x is a prefix of 010 in Figure 5. At the root node we have $\mathsf{V}_\phi^2 = \{S_2\}$ and $\mathsf{W}_\phi = \emptyset$. The set V_0^2 contains $S_{2,0}$ and $S_{2,10}$ and the set W_0^2 is still empty. Next note that V_{01}^2 contains $S_{2,01}, S_{2,10}$ and W_{01}^2 contains $\mathsf{PRG}_0(S_{2,01})$. Finally set $\mathsf{V}_{010}^2 = \emptyset$ and W_{010}^2 continues to contain $\mathsf{PRG}_0(S_{2,01})$.

Properties of the special tree key structure. We now prove several properties about the special tree key structure. Intuitively speaking the crux of the lemmas is the claim V-set for can a node can be used to derive its children. Furthermore each element in V-set for any node can only be derived from the V-set of nodes in exactly two different paths.

Lemma 21 (Computability Lemma) *There exists an* explicit *efficient procedure that given* $\mathsf{V}_x, \mathsf{W}_x$ *computes* $\mathsf{V}_{x\|0}, \mathsf{W}_{x\|0}$ *and* $\mathsf{V}_{x\|1}, \mathsf{W}_{x\|1}$.

Proof We start by noting that it suffices to show that for each i, given $\mathsf{V}^i_x, \mathsf{W}^i_x$ one can compute $\mathsf{V}^i_{x\|0}, \mathsf{W}^i_{x\|0}$ and $\mathsf{V}^i_{x\|1}, \mathsf{W}^i_{x\|1}$. We argue this next. Observe that two cases arise either $|x| < i$ or $|x| \geq i$. We deal with the two cases:

- $|x| < i$: In this case V^i_x is $\{S_{i,x}, S_{i,(x+1)\|0^{i-|x|}}\}$ and these values can be used to compute $S_{i,x\|0}$, $S_{i,x\|1}$, $S_{i,(x\|0)+1} = S_{i,x\|1}$ and $S_{i,((x\|1)+1)\|0^{i-|x|-1}} = S_{i,(x+1)\|0\|0^{i-|x|-1}} = S_{i,(x+1)\|0^{i-|x|}}$. Observe by case by case inspection that these values are sufficient for computing $\mathsf{V}^i_{x\|0}, \mathsf{W}^i_{x\|0}$ and $\mathsf{V}^i_{x\|1}, \mathsf{W}^i_{x\|1}$ in all cases.

- $|x| \geq i$: Note that according to the constraints placed on x by the definition, if $\mathsf{V}^i_x = \emptyset$ then both $\mathsf{V}^i_{x\|0}$ and $\mathsf{V}^i_{x\|1}$ must be $\emptyset$ as well. On the other hand if $V^i_x \neq \emptyset$ then $\mathsf{V}^i_{x\|0}$ is still $\emptyset$ while $\mathsf{V}^i_{x\|1} = \mathsf{V}^i_x$. Additionally, $W^i_{x\|0} = W^i_{x\|1} = W^i_x$.

This concludes the proof. ∎

Lemma 22 (Derivability Lemma) *For every $i \in [\kappa], x \in \{0,1\}^i$ and $x \neq 1^i$ we have that, $S_{i,x+1}$ can be derived from keys in V^i_y if and only if y is a prefix of $x\|1^{\kappa-i}$ or $(x+1)\|1^{\kappa-i}$. Additionally, $S_{i,0^i}$ can be derived from keys in V_y if and only if y is a prefix of $0^i\|1^{\kappa-i}$.*

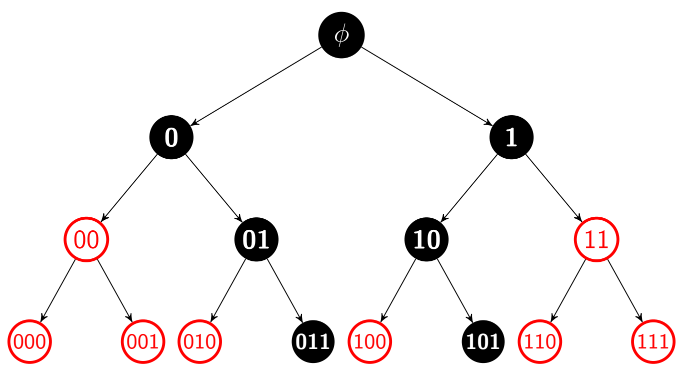

Figure 6: Black nodes represent the choices of $x \in \{0,1\}^{\leq 3}$ such that V^2_x can be used to derive $S_{2,10}$.

Proof We start by noting that for any $y \in \{0,1\}^{>i} \cap \{0,1\}^{\leq \kappa}$, by definition of V-sets we have that $\mathsf{V}^i_y = \mathsf{V}^i_{y_{[i]}}$ or $\mathsf{V}^i_y = \emptyset$. Hence it suffices to prove the above lemma for $y \in \{0,1\}^{\leq i}$.

We first prove that if y is a prefix of x or $(x+1)$ then we can derive $S_{i,x+1}$ from V^i_y. Two cases arise:

- Observe that if y is a prefix of x then we must have that either y is a prefix of $x+1$ or $x+1 = (y+1)\|0^{i-|y|}$. Next note that by definition of V-sets we have that $\mathsf{V}^i_y = \{S_{i,y}, S_{i,(y+1)\|0^{i-|y|}}\}$, and one of these values can be used to compute $S_{i,x+1}$.

- On the other hand if y is a prefix of $x+1$ then again by definition of V-sets we have that $\mathsf{V}^i_y = \{S_{i,y}, S_{i,(y+1)\|0^{i-|y|}}\}$, and $S_{i,y}$ can be used to compute $S_{i,x+1}$.

Next we show that no other $y \in \{0,1\}^{\leq i}$ allows for such a derivation. Note that by definition of V-sets we have that $V^i_y = \{S_{i,y}, S_{i,(y+1)\|0^{i-|y|}}\}$. We will argue that neither $S_{i,y}$ nor $S_{i,(y+1)\|0^{i-|y|}}$ can be used to derive $S_{i,x+1}$.

- We are given that y is not a prefix of $x+1$. This implies that $S_{i,y}$ cannot be used to derive $S_{i,x+1}$.
- Now we need to argue that $S_{i,(y+1)\|0^{i-|y|}}$ cannot be used to compute $S_{i,x+1}$. For this, it suffices to argue that $x+1 \neq (y+1)\|0^{i-|y|}$. If $x+1 = (y+1)\|0^{i-|y|}$ then y must be prefix of x. However, we are given that this is not the case. This proves our claim.

The argument for the value $S_{i,0^i}$ follows analogously. This concludes the proof. ■

4.2 Hard on Average SVL Instances

In this section, we describe our construction for hard on average instance of SVL. In particular, we describe our sampler that samples hard on average instances $(x_s, \mathsf{Succ}, \mathsf{Ver}, 1^\kappa)$. Here x_s is the source node and Succ is the successor circuit. We define 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 k^{th} node from x_s. That is, $\mathsf{Ver}(x,k) = 1$ iff $x = \mathsf{Succ}^{k-1}(x_s)$. For the generated instances, we argue that it is hard to find the 1^κ node in the path.

In our construction we use a selectively secure functional encryption scheme ($\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec}$), a prefix-puncturable PRF (Definition 19), a semantically secure symmetric key encryption ($\mathsf{SK.KeyGen}, \mathsf{SK.Enc}, \mathsf{SK.Dec}$) and injective PRGs having the left half injectivity property Definition 9. PRG_0 and PRG_1 denote the left and the right part of the output of this PRG.

The formal description of hard on average SVL instance sampler is provided in Figure 7. Internally this sampler generates the successor circuit to include functional encryption secret keys for circuits provided in Figure 8. Next we informally describe the SVL instances considered.

A sampled instance implicitly defines a line graph where each node in the graph is of the form $(x, \sigma_1, \cdots, \sigma_\kappa)$ where $\sigma_i = \mathsf{PrefixPunc}(S_i, x_{[i]})$ for all $i \in [\kappa]$. We say a node is *valid* if the above condition holds. The node $(x, \sigma_1, \cdots, \sigma_\kappa)$ is connected to $(x+1, \sigma'_1, \cdots, \sigma'_\kappa)$ by an outgoing edge and to $(x-1, \sigma''_1, \cdots, \sigma''_\kappa)$ by an incoming edge. The successor circuit on input $(x, \sigma_1, \cdots, \sigma_\kappa)$ checks for the validity of the node and if the node is valid it outputs $(x+1, \sigma'_1, \cdots, \sigma'_\kappa)$. The verification circuit on input $(x, \sigma_1, \cdots, \sigma_\kappa, j)$ outputs if and only if $x = j-1$ and $(x, \sigma_1, \cdots, \sigma_\kappa)$ is valid.

We now explain how the successor circuit works. The successor circuit is described by a sequence of $\kappa+1$ secret keys $\mathsf{FSK}_1, \cdots, \mathsf{FSK}_{\kappa+1}$ for appropriate functions. There keys are generated corresponding to independent instances of functional encryption. Along with the keys the successor circuit also contains a ciphertext c_ϕ that encrypts the empty string, ϕ, under PK_1 alsong with the key values V_ϕ and W_ϕ. Intuitively, the function key FSK_i corresponds to a function F_i that takes as input a binary string x of length i and outputs an encryption of $x\|0$ and $x\|1$ under PK_{i+1}. Additionally these ciphertexts, in addition to $x\|0$ and $x\|1$, also contain key values $\mathsf{V}_{x\|0}, \mathsf{W}_{x\|0}$ and $\mathsf{V}_{x\|1}, \mathsf{W}_{x\|1}$ respectively. Recall from Section 4.1 that the keys in these sets are used to test validity of signatures provides as input and to generate the new ones.

The successor circuit on an input of the form $(x, \sigma_1, \cdots, \sigma_\kappa)$ does the following. It first obtains an encryption of x along with key values V_x and W_x under the public key $PK_{\kappa+1}$. This is done as follows. Start with c_ϕ and decrypt it using key FSK_1 to obtain encryptions of 0 and 1. Choose one of them based on which one is a prefix of x and continue the process. Repeating this process κ times results in the desired ciphertext. Next decrypt the obtained ciphertext using $\mathsf{FSK}_{\kappa+1}$ and it provides some information essential for checking validity of provided input signatures and additional

- **Sampled Ingredients**:
  1. Sample $\{S_i\}_{i\in[\kappa]}$ and K_ϕ from $\mathsf{KeyGen}_{\mathcal{PPRF}}(1^\kappa)$. Here S_i's is a key that works for i bit inputs, namely $\mathsf{PPRF}_{S_i} : \{0,1\}^i \to \{0,1\}^\kappa$ for all $i \in [\kappa]$. Similarly, K_ϕ works on inputs of length 2κ. Initialize $\mathsf{V}^i_\phi = S_i$, $\mathsf{V}_\phi = \bigcup_{i\in[\kappa]} \mathsf{V}^i_\phi$ and $\mathsf{W}_\phi = \emptyset$.
  2. Sample $(PK_i, MSK_i) \leftarrow \mathsf{FE.Setup}(1^\kappa)$ for all $1 \le i \le \kappa + 1$.
  3. Sample $sk \leftarrow \mathsf{SK.KeyGen}(1^\kappa)$ and let $\Pi \leftarrow \mathsf{SK.Enc}_{sk}(\pi)$ and $\Lambda \leftarrow \mathsf{SK.Enc}_{sk}(\lambda)$ where $\pi = 0^{\ell(\kappa)}$ and $\lambda = 0^{\ell(\kappa)}$. Here $\ell(\cdot)$ and $\ell'(\cdot)$ are appropriate length functions specified later.
  4. Sample $v \leftarrow \{0,1\}^{2\kappa}$.

- **Functional encryption ciphertext and keys to simulate obfuscation**:
  1. For each $i \in [\kappa]$ generate $\mathsf{FSK}_i \leftarrow \mathsf{FE.KeyGen}(MSK_i, F_{i,PK_{i+1},\Pi})$ and $\mathsf{FSK}_{\kappa+1} \leftarrow \mathsf{FE.KeyGen}(MSK_{\kappa+1}, G_{v,\Pi})$, where $F_{i,PK_{i+1},\Pi}$ and $G_{v,\Lambda}$ are circuits described in Figure 8.
  2. Let $c_\phi = \mathsf{FE.Enc}_{PK_1}(\phi, \mathsf{V}_\phi, \mathsf{W}_\phi, 0^\kappa, 0)$

- **Source node**: The source node x_s is given by $(0^\kappa, \sigma_1, \cdots, \sigma_\kappa)$ where $\sigma_i = \mathsf{PPRF}_{S_i}(0^i)$ for all $i \in [\kappa]$.

- **Successor Circuit**: The successor circuit Succ in our setting takes as input $x, \sigma_1, \ldots, \sigma_\kappa$ and outputs $x+1, \sigma'_1, \ldots, \sigma'_\kappa$ if the associated signatures $\sigma_1, \cdots, \sigma_\kappa$ are valid. It proceeds as follows:
  1. For $i \in [\kappa]$ compute $c_{x_{[i-1]}\|0}, c_{x_{[i-1]}\|1} := \mathsf{FE.Dec}(\mathsf{FSK}_i, c_{x_{[i-1]}})$.
  2. Obtain $d_x = ((\alpha_1, \ldots, \alpha_\kappa), (\beta_j, \ldots, \beta_\kappa))$ as output of $\mathsf{FE.Dec}(\mathsf{FSK}_{\kappa+1}, c_x)$. Here $j = f(x)$. Recall from Section 3.2 that $f(x)$ is the largest j such that $x = x_{[j]}\|1^{\kappa-j}$.
  3. Output $\perp$ if $\mathsf{PRG}_0(\sigma_i) \neq \alpha_i$ for any $i \in [\kappa]$.
  4. If $x = 1^\kappa$, output SOLVED.
  5. For each $i \in [j-1]$ set $\sigma'_i = \sigma_i$.
  6. For each $i \in \{j, \ldots, \kappa\}$ set $\gamma_i = \mathsf{PRG}_1(\sigma_i)$ and σ'_i as $\mathsf{SK.Dec}_{\gamma_j,\cdots,\gamma_\kappa}(\beta_i)$, decrypting β_i encrypted under $\gamma_j, \ldots \gamma_\kappa$.
  7. Output $(x+1, \sigma'_1, \cdots, \sigma'_\kappa)$.

- **Verification Circuit**: The verification circuit Ver on input $x, \sigma_1, \ldots, \sigma_\kappa, j$ outputs 1 if Succ on input $x, \sigma_1, \ldots, \sigma_\kappa$ doesn't output $\perp$ and $x = j-1$ and 0 otherwise.

Figure 7: Hard on average instance for SVL based on hardness of FE.

information to generate the signatures for the next node. More details are provided in Figures 7 and 8.

$F_{i,PK_{i+1},\Lambda}$

Hardcoded Values: i, PK_{i+1}, Π.
Input: ($x \in \{0,1\}^{i-1}$, V_x,W_x, K_x, sk, mode)

1. If $(\mathsf{mode} = 0)$ then output $\mathsf{FE.Enc}_{PK_{i+1}}(x\|0, \mathsf{V}_{x\|0}, \mathsf{W}_{x\|0}, K_{x\|0}, sk, \mathsf{mode}; K'_{x\|0})$ and $\mathsf{FE.Enc}_{PK_{i+1}}(x\|1, \mathsf{V}_{x\|1}, \mathsf{W}_{x\|1}, K_{x\|1}, sk, \mathsf{mode}; K'_{x\|1})$, where for $b \in \{0,1\}$, $K_{x\|b} = \mathsf{PrefixPunc}(K_x, b\|0)$ and $K'_{x\|b} = \mathsf{PrefixPunc}(K_x, b\|1)$ and $(\mathsf{V}_{x\|0}, \mathsf{W}_{x\|0})$, $(\mathsf{V}_{x\|1}, \mathsf{W}_{x\|1})$ are computed using the efficient procedure from the Computability Lemma (Lemma 22).
2. Else recover $(x||0, c_{x\|0})$ and $(x\|1, c_{x\|1})$ from $\mathsf{SK.Dec}_{sk}(\Pi)$ and output $c_{x\|0}$ and $c_{x\|1}$.

$G_{v,\Lambda}$

Hardcoded Values: v, Λ
Input: $x \in \{0,1\}^\kappa, \mathsf{V}_x, \mathsf{W}_x, K_x, sk, \mathsf{mode}$

1. If $(\mathsf{PRG}(x) = v)$ then output $\perp$.
2. If $\mathsf{mode} = 0$, (Below $j = f(x)$. Recall from Section 3.2 that $f(x)$ is the largest j such that $x = x_{[j]}\|1^{\kappa-j}$.)
   (a) For each $i \in [\kappa]$, set $\alpha_i = \mathsf{PRG}_0(\sigma_i)$ (obtained from W_x^i for $i \leq j$ and from V_x^i for $i > j$).
   (b) For each $i \in \{j, \ldots, \kappa\}$ set $\gamma_i = \mathsf{PRG}_1(\sigma_i)$ and $\beta_i = \mathsf{SK.Enc}_{\gamma_j,\cdots,\gamma_\kappa}(S_{i,x_{[i]}+1})$, encrypting $S_{i,x_{[i]}+1}$ under $\gamma_j, \ldots \gamma_\kappa$. (Using randomness obtained by expanding K_x sufficiently.)
   (c) Output $((\alpha_1, \ldots, \alpha_\kappa), (\beta_j, \ldots, \beta_\kappa))$
3. Else recover (x, d_x) from $\mathsf{SK.Dec}_{sk}(\Lambda)$ and output d_x.

Figure 8: Circuits for which functional encryption secret keys are given out.

Correctness. The correctness of our construction follows for the correctness of the underlying primitives. Here we note that since $v \xleftarrow{\$} \{0,1\}^{2\kappa}$, therefore the probability that v is in the image of the PRG is negligible and hence this step of $G_{v,\Lambda}$ is not triggered. Similarly since the provided ciphertext c_ϕ is set to be in mode 0 the other mode is never triggered.

Given that the above case do not arise we have that $\alpha_i = \mathsf{PRG}_0(S_{i,x_{[i]}})$. Recall that the successor check if $\mathsf{PRG}_0(\sigma_i) = \alpha_i$. Now from the left half injectivity property of the PRG we have that these checks pass if and only if $\sigma_i = S_{i,x_{[i]}}$. Hence, if successor does not output $\perp$ then the input $x, \sigma_1, \cdots, \sigma_\kappa$ must be valid. Additionally from the correctness of $\mathsf{SK.Dec}$ we have that the successor recovers the correct values $\sigma'_i = S_{i,x+1_{[i]}}$ corresponding to prefixes of $x+1$ which are different from the prefixes of x. This provides the full set of associated signatures on $x+1$.

The correctness of the verification circuit Ver directly follows from the correctness of the successor circuit.

4.3 Security Proof

In this section, we will prove that the SVL instances generated in our construction are hard to solve. The proof of security mimics the proof for the setting of $i\mathcal{O}$ with two crucial differences. Specifically:

1. In our setting the honest distribution has the PRG check in-built and so we do not need to introduce it as was done in going from Hyb_0 to Hyb_1 as done in Section 3.2. In particular, this check is built into the circuit $G_{v,\Lambda}$.

2. Since we are using functional encryption, the puncturing, checking of input signatures and generating on new ones has to be done differently and a bit more carefully. We highlight these differences as we go along the proof.

Hybrid Structure. We describe our proof in a way analogous to the proof from Section 3.2. In particular, we show that no polynomial time adversary can output a valid value $(1^\kappa, \sigma_1, \cdots, \sigma_\kappa)$. We will show this via a hybrid argument. Starting with providing SVL instance as in distribution from Figure 7, namely hybrid Hyb_0, we move to a hybrid $\mathsf{Hyb}_{4,\delta}$, where δ is defined below. In the final hybrid, the successor circuit returns $\perp$ on all inputs of the form $(1^\kappa, \cdot, \cdots, \cdot)$. Observe that the advantage of any adversary in solving the SVL instance in this final hybrid is 0. This proves our claim. We make this change by going through a polynomial in the security parameter number of intermediate hybrids. Next we describe our hybrids.

- Hyb_0 : This hybrid corresponds to SVL instance generation as given in Figure 7.

- Hyb_1 : In this hybrid we change how the value v, hard-coded in $G_{v,\Lambda}$ is generated. In particular, instead of sampling v uniformly at random from $\{0,1\}^{2\kappa}$, we generate v as $\mathsf{PRG}(u_0)$ where u_0 is sampled uniformly from $\{0,1\}^\kappa$. Here PRG is a length doubling injective pseudorandom generator.

  Computational indistinguishability between Hyb_0 and Hyb_1 follows from the pseudorandomness property of the PRG.

  Recalling notation for u_j. Recall from Section 3.2, for any string $u \in \{0,1\}^\kappa$, let $f(u)$ denote the index of the lowest order bit of u that is 0 (with the index of the highest order bit being 1). More formally, $f(u)$ is the largest j such that $u = u_{[j]}||1^{\kappa-j}$. For example, if $u = \overbrace{100}^{3}11$ then $f(u) = 3$. Also let $\delta(u)$ be the number of 0 bits in the binary representation of u.

  Starting with a value $u_0 \in \{0,1\}^\kappa$ we define a sequence of values such that u_{j+1} is the value u_j with the $f(u_j)^{th}$ bit set to 1. More formally, $u_{j+1} = u_j + 2^{\kappa - f(u_j)}$. We will use the shorthand δ to denote $\delta(u_0)$.

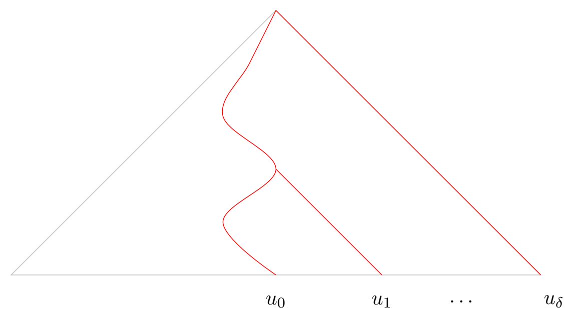

Figure 9: Starting with u_0, u_{j+1} is obtained by setting the lowest order 0-bit in u_j to 1.

New π, λ values. As in Section 3.2 we process the hybrids according to U_j values. However, here we need to set the stage before these can be done. For this we define some additional notation. For any $x \in \{0,1\}^{\leq\kappa}$, let $c^\star_x$ denote the ciphertext and $d^\star_x$ the clear output in execution of the successor in the Hyb_1 when Steps 1 and 2 of the successor circuit are executed on input x. We let P be the set of all prefixes of $u_0, u_1, \ldots u_\delta$ including the empty string ϕ. Note that $|P| \leq (\kappa+1)^2$. Additionally we define Q as follows. For every $x \in P$, let y be the value with the last bit of x flipped. We add y to Q if $y \notin P$. We set:

$$\pi^\star = ||_{x\in P\cup Q}\ (x, c^\star_x)$$
$$\lambda^\star = ||_{x\in P\cap\{0,1\}^\kappa}\ (x, d^\star_x)$$

We set $\ell(\kappa)$ and $\ell'(\kappa)$ to be the polynomials that describe an upper bound on the lengths of $\pi^\star$ and $\lambda^\star$ over all choices of $u_0 \in \{0,1\}^\kappa$.

- Hyb_2: In this hybrid we change how the hardcoded values Π and Λ are generated. Unlike hybrids Hyb_1 and Hyb_2 where these values were generated as encryptions of $0^{\ell(\kappa)}$ and $0^{\ell'(\kappa)}$, in this hybrid we generate them as encryptions $\pi^\star$ and $\lambda^\star$ describe above, respectively.

  Computational indistinguishability between Hyb_1 and Hyb_2 follows from the semantic security of the encryption scheme.

- Hyb_3: In this hybrid, for $x \in P$ we change the the $c^\star_x$ values embedded in $\pi^\star$. Recall that in hybrid Hyb_2 for each x, $c^\star_x$ is generated as $\mathsf{FE.Enc}_{PK_{|x|+1}}(x, \mathsf{V}_x, \mathsf{W}_x, K_x, 0^\kappa, 0; K'_x)$. We change the $c^\star_x$ to be now generated as $\mathsf{FE.Enc}_{PK_{|x|+1}}(x, 0^{2\kappa}, 0^{2\kappa}, 0^\kappa, sk, 1; \omega_x)$ using fresh randomness ω_x.[3]

  Computational indistinguishability between Hyb_2 and Hyb_3 follows by a sequence of sub-hybrids. We define an ordering on elements of P as follows. For $x, y \in P$ we say that $x < y$ if either $|x| < |y|$, or $|x| = |y|$ and $x < y$ [4]. Next we define the hybrid $\mathsf{Hyb}_{2,y}$ to be a modification of Hyb_2 where for all $x \in P$ such that $x \leq y$ we have that $c^\star_x$ is generated as $\mathsf{FE.Enc}_{PK_{|x|+1}}(x, 0^{2\kappa}, 0^{2\kappa}, 0^\kappa, sk, 1; \omega_x)$ using fresh randomness ω_x.

  Note that it suffices to argue that $\mathsf{Hyb}_{2,x'}$ and $\mathsf{Hyb}_{2,x}$ are indistinguishable for any two adjacent values x' and x in P such that $x' < x$. We argue this via a two step hybrid argument.

[3] Note that we do not change ciphertexts corresponding to $x \in Q$.
[4] Note that ϕ is the smallest value in P by this ordering.

1. $\mathsf{Hyb}_{2,x,1}$: In this hybrid we change $c^\star_x$ to $\mathsf{FE.Enc}_{PK_{|x|+1}}(x, \mathsf{V}_x, \mathsf{W}_x, K_x, 0^\kappa, 0; \omega_x)$ using fresh randomness ω_x.

   Note that for all prefixes x'' of x we have that $x'' < x$. Therefore for all such x'' we have that $c_{x''} = \mathsf{FE.Enc}_{PK_{|x''|+1}}(x'', 0^{2\kappa}, 0^{2\kappa}, 0^\kappa, sk, 1; \omega_{x''})$. This fact along with the pseudorandom at punctured point property implies computational indistinguishability from the previous hybrid, namely $\mathsf{Hyb}_{2,x}$.

2. $\mathsf{Hyb}_{2,x,2}$: In this hybrid we change $c^\star_x$ to $\mathsf{FE.Enc}_{PK_{|x|+1}}(x, 0^{2\kappa}, 0^{2\kappa}, 0^\kappa, sk, 1; \omega_x)$ using fresh randomness ω_x.

   The computational indistinguishability of this hybrid from $\mathsf{Hyb}_{2,x,2}$ relies on the selective security of the functional encryption scheme with public-key $PK_{|x|+1}$. Note that we can invoke security of functional encryption as the change in the messages being encrypted does not change the output of the decryption using key $\mathsf{FSK}_{|x|+1}$.

- $\mathsf{Hyb}_{4,j+1}$: In hybrid $\mathsf{Hyb}_{4,j+1}$ for $j \in \{0, \ldots, \delta - 1\}$, we make two changes with respect to $\mathsf{Hyb}_{4,j}$. Just like in Section 3.2 we let f_j as the shorthand for $f(u_j)$ and $t_j = u_{j[f_j]} + 1$.

  - We change the set $\mathsf{W}^{f_j}_{t_j}$ to be a uniformly random string $z \leftarrow \{0,1\}^{2\kappa}$ rather than containing the value $\mathsf{PRG}_0(S_{f_j,t_j})$. Note that this change needs to be made at two places. Namely we set $\mathsf{W}^{f_j}_{t_j\|0} = \{z\}$ in $c^\star_{t_j\|0}$ and from there on this value will be percolated to all its descendents as well. Additionally we set α_{f_j} included in $d^\star_{u_{j+1}}$ to be z.
  - We now generate encryptions $\beta_{f_j}, \ldots, \beta_\kappa$ included in $d^\star_{u_{j+1}}$ with encryption of 0^κ.

  Note that as a consequence of this change the successor circuit now starts to output $\perp$ additionally on all inputs in $\{u_j + 1, \ldots, u_{j+1}\}$. This is because for every input σ_{f_j} we have that $z \neq \mathsf{PRG}_0(\sigma_{f_j})$ with overwhelming probability. Since in hybrid $\mathsf{Hyb}_{4,j}$ the successor was already outputting $\perp$ on inputs $\{u_0, \ldots, u_j\}$ we have that the successor outputs $\perp$ on all inputs in $\{u_0, \ldots, u_{j+1}\}$.

  Now we argue computational indistinguishability between $\mathsf{Hyb}_{4,j}$ and $\mathsf{Hyb}_{4,j+1}$.

  - $\mathsf{Hyb}_{4,j,1}$: In this hybrid instead we replace the key S_{f_j,t_j} with a random string $S' \leftarrow \{0,1\}^\kappa$.

    Computational indistinguishability follows from the pseudorandom at punctured point property. This argument relies on that fact that no V set that hasn't been removed can be used to obtain S_{f_j,t_j}. This follows from the following two facts.

    - $\mathsf{V}^{f_j}_y$ values have been removed whenever y is a prefix of u_j or u_{j+1}. Note that it follows from the Derivability Lemma (Lemma 23) that these were the only V-sets that could be used to derive S_{f_j,t_j}.
    - Additionally σ_{f_j} is encrypted in β_{f_j} and this value is included in $d^\star_{u_j}$. But this has already been replaced with an encryption of 0^κ except $d^\star_{u_0}$ which is set to $\perp$.
  - $\mathsf{Hyb}_{4,j,2}$: In this hybrid instead we replace the $\beta_{f_j}, \ldots, \beta_\kappa$ in $d^\star_{u_{j+1}}$ to be generated using fresh randomness.

    Computational indistinguishability follows from the pseudorandom at punctured point property.

– $\mathsf{Hyb}_{4,j,3}$: In this hybrid, we change $\mathsf{PRG}_0(S')$ and $\mathsf{PRG}_1(S')$ to be random strings z, z'. Change of $\mathsf{PRG}_0(S')$ to z implies that the set $\mathsf{W}_{t_j}^{f_j}$ is $\{z\}$. Similarly γ_{f_j} will be z'.

  Indistinguishability between $\mathsf{Hyb}_{4,j}$ and $\mathsf{Hyb}_{4,j,1}$ follows from the pseudorandomness property of the PRG.

– $\mathsf{Hyb}_{4,j,4}$: In this hybrid we set encryption of all $\beta_{f_j}, \ldots, \beta_\kappa$ in $d^\star_{u_{j+1}}$ with encryption of 0^κ.

  Next by semantic security we have this hybrid is computationally indistinguishable from the previous. Here we rely on the fact that one of the keys γ_{f_j} has been replaced with random.

Note that hybrid $\mathsf{Hyb}_{4,j,4}$ is same as hybrid $\mathsf{Hyb}_{4,j+1}$.

Concluding the proof. Observe that the hybrid $\mathsf{Hyb}_{4,0}$ is identical to hybrid Hyb_3 and $\mathsf{Hyb}_{4,\delta}$ is such that the successor outputs $\perp$ on all inputs of the form $(1^\kappa, \cdot, \ldots, \cdot)$. Consequently, no adversary can solve the SVL instance in this final hybrid with probability better than 0.

Acknowledgements

The first author would like to thank Sidharth Telang for useful discussions on related topics. Research supported in part from DARPA Safeware Award W911NF15C0210, AFOSR Award FA9550-15-1-0274, and NSF CRII Award 1464397. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense, the National Science Foundation, or the U.S. Government.

References

[AB15] Benny Applebaum and Zvika Brakerski. Obfuscating circuits via composite-order graded encoding. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 528–556, Warsaw, Poland, March 23–25, 2015. Springer, Heidelberg, Germany.

[AJ15] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 308–326, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

[AJS15] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Achieving compactness generically: Indistinguishability obfuscation from non-compact functional encryption. *IACR Cryptology ePrint Archive*, 2015:730, 2015.

[AKV04] Tim Abbot, Daniel Kane, and Paul Valiant. On Algorithms for Nash Equilibria, 2004. `http://web.mit.edu/tabbott/Public/final.pdf`.

[BCC+14] Nir Bitansky, Ran Canetti, Henry Cohn, Shafi Goldwasser, Yael Tauman Kalai, Omer Paneth, and Alon Rosen. The impossibility of obfuscation with auxiliary input or a universal simulator. In *CRYPTO*, pages 71–89, 2014.

[BGI+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, pages 501–519, 2014.

[BGK+14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 221–238, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.

[BPR15] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a nash equilibrium. In *FOCS*, 2015.

[BR14] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 1–25, San Diego, CA, USA, February 24–26, 2014. Springer, Heidelberg, Germany.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, pages 253–273, 2011.

[BV15a] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In *FOCS*, 2015.

[BV15b] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. *IACR Cryptology ePrint Archive*, 2015:163, 2015.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, pages 280–300, 2013.

[CDT09] Xi Chen, Xiaotie Deng, and Shang-Hua Teng. Settling the complexity of computing two-player nash equilibria. *J. ACM*, 56(3), 2009.

[DGP09] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. *Commun. ACM*, 52(2):89–97, 2009.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 1–17, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[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 *54th FOCS*, pages 40–49, Berkeley, CA, USA, October 26–29, 2013. IEEE Computer Society Press.

[GGHZ16] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. Fully secure functional encryption from multilinear maps. In *TCC*, 2016.

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

[GK05] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *FOCS*, pages 553–562, 2005.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564, 2013.

[GL89] Oded Goldreich and Leonid A. Levin. A hard-core predicate for all one-way functions. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 25–32, 1989.

[GLSW15] Craig Gentry, Allison B. Lewko, Amit Sahai, and Brent Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. In *FOCS*, 2015.

[GVW12] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 162–179, 2012.

[Jer12] Emil Jerábek. Integer factoring and modular square roots. *CoRR*, abs/1207.5220, 2012.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 669–684, 2013.

[MP91] Nimrod Megiddo and Christos H. Papadimitriou. On total functions, existence theorems and computational complexity. *Theor. Comput. Sci.*, 81(2):317–324, 1991.

[Nas51] John Nash. Non-cooperative games. *The Annals of Mathematics*, 54(2):286–295, 1951.

[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. *J. Comput. Syst. Sci.*, 48(3):498–532, 1994.

[PST14] Rafael Pass, Karn Seth, and Sidharth Telang. Indistinguishability obfuscation from semantically-secure multilinear encodings. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 500–517, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 475–484, 2014.

[Wat15] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 678–697, 2015.

[Zim15] Joe Zimmerman. How to obfuscate programs directly. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 439–467, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.