IO Construction: From XIO to IO

1 A Tool: Randomized Encodings

One tool we will use in the compiler from XIO to IO is efficient randomized encodings.

Definition 1 (Efficient Randomized Encodings, informal)**.** *An efficient $(T(\lambda), \epsilon(\lambda))$-randomized encoding scheme* $(\mathsf{RE.Encode}, \mathsf{RE.Eval}, \mathsf{RE.Sim})$ *is a tuple of probabilistic algorithms satisfying the following conditions for all circuits $C : \{0,1\}^n \to \{0,1\}^m$ and inputs $x \in \{0,1\}^n$:*

Correctness*: It holds with probability* 1 *that*

$$\mathsf{RE.Eval}(\mathsf{RE.Encode}(1^\lambda, C, x)) = C(x)$$

Security*: The distributions* $\mathsf{RE.Encode}(1^\lambda, C, x)$ *and* $\mathsf{RE.Sim}(1^\lambda, C, C(x))$ *cannot be distinguished with probability greater than $\epsilon(\lambda)$ by any $T(\lambda)$-time distinguisher.*[1] *Note that the distributions are taken just over the internal randomness of each algorithm.*

Efficiency*: Let ℓ denote the output length of* $\mathsf{RE.Encode}(1^\lambda, C, x)$*. For each $i \in [\ell]$, there is a randomized circuit*

$$D_i : \{0,1\}^{\lambda+|C|+n} \to \{0,1\}$$

such that

$$D_i(1^\lambda, C, x; r) = \mathsf{RE}(1^\lambda, C, x; r)_i$$

for all r, and the size of each circuit D_i is at most $\mathrm{poly}(n, \lambda)$*, independently of $|C|$ and m.*

Since we will treat the circuit C as fixed, we will sometimes denote $\mathsf{RE.Encode}(1^\lambda, C, x; r)$ *as* $\mathsf{RE.Encode}_C(x; r)$*.*

The crucial property for the XIO to IO compiler is efficiency, as that allows one to in some sense "compress" the size of the circuit if you only care about outputting one bit.

A randomized encoding for all polynomial-size circuits exists assuming one-way functions exist. The construction we use ("Yao's garbled circuits") is attributed to (oral presentations of) Yao [Yao86], but the security proof was first formalized by Lindell and Pinkas [LP09].

Theorem 2 ([Yao86, LP09])**.** *Assuming the existence of one-way functions, $\mathcal{P}$ has efficient randomized encodings.*

We give a quick sketch of the construction, but refer to [LP09] for more details.

Proof sketch. Since we assume the existence of one-way functions, we can assume the existence of a secret-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and pseudo-random functions (PRFs). The main idea is as follows. Suppose C is a circuit composed purely of NAND gates (without loss of generality). For each wire w in the circuit (including input and output wires), we will sample encryption keys $k_w^0, k_w^1 \leftarrow \mathsf{Gen}(1^\lambda)$, where the key k_w^σ corresponds to wire w holding logical value $\sigma \in \{0,1\}$. In our usage of garbled circuits, we will

[1]One could instead require that the simulator have access to only $C(x)$ (and not C), which would make the definition stronger. By considering the universal circuit, one could achieve this generically at the cost of some blow-up, but the definition we give suffices for our purposes.

actually sample k_w^0, k_w^1 using randomness from a PRF instead of fresh randomness each time. That is, we have some global PRF key k, and we define

$$k_w^\sigma = \mathsf{Gen}(1^\lambda; \mathsf{PRF}_k(w \| \sigma))$$

Fix some gate $g = (a, b, c)$ in the circuit, where a and b are inputs wires and c is an output wire. For each such $g \in C$, we will let T_g be the randomly permuted table consisting of the four values

$$\left\{ \mathsf{Enc}_{k_a^{\sigma_a}} \left(\mathsf{Enc}_{k_b^{\sigma_b}} \left(k_c^{\mathsf{NAND}(\sigma_a, \sigma_b)} \right) \right) \right\}_{\sigma_a, \sigma_b \in \{0,1\}}.$$

That is, given $k_a^{\sigma_a}$ and $k_b^{\sigma_b}$, one can use Dec to compute $k_c^{\mathsf{NAND}(\sigma_a, \sigma_b)}$. Intuitively, this allows one to work up the circuit and keep obtaining keys for wires with their corresponding logical values.

To complete the picture, we also need the input and output encodings. For simplicity of notation, let in_i denote the i-th input wire for $i \in [n]$, and let out_j denote the j-th output wire for $j \in [m]$.

The input encoding for bit $i \in [n]$ will consist of just $k_{\mathsf{in}_i}^{x_i}$, and the output encoding for bit $j \in [m]$ will consist of

$$\left(k_{\mathsf{out}_j}^0, k_{\mathsf{out}_j}^1 \right)$$

in this order, to allow one to know which key corresponds to logical 0 and which corresponds to logical 1. In full, $\mathsf{RE.Encode}(1^\lambda, C, x)$ will consist of all T_g (for gates $g \in C$), and all input and output encodings.

The algorithm $\mathsf{RE.Eval}$ works as follows. Starting from the input encodings, we choose some topological ordering of the gates, and for each gate, we will try decrypting all four possible (double) ciphertexts based on the keys we have from the input gates (inductively). We can generically modify our encryption protocol such that with high probability, exactly one of these (double) ciphertexts will give a recognizably valid decryption, allowing one to obtain exactly one key for the output wire.[2] Finally, once reaching the output tables, we can simply look at the output table and output the corresponding bit based on the key we have.

The simulator $\mathsf{RE.Sim}$ will work as follows. For the input wires, the simulator samples $k_{\mathsf{in}_i}^\sigma \leftarrow \mathsf{Gen}(1^\lambda)$ for $\sigma \in \{0, 1\}$ as usual, but for each g, the table T_g will now consist of four ciphertexts of k_c^0. That is, the simulated table will be a random permutation of

$$\left\{ \mathsf{Enc}_{k_a^{\sigma_a}} \left(\mathsf{Enc}_{k_b^{\sigma_b}} \left(k_c^0 \right) \right) \right\}_{\sigma_a, \sigma_b \in \{0,1\}}.$$

Lastly, the output encoding for the j-th bit is programmed to be

$$\left(k_{\mathsf{out}_j}^{C(x)_j}, k_{\mathsf{out}_j}^{1 - C(x)_j} \right)$$

to ensure that $k_{\mathsf{out}_j}^0$ maps to $C(x)_j$. To argue that this is indistinguishable from the randomized encoding requires a careful hybrid argument [LP09] but ultimately follows from security of the encryption scheme.

Since $\mathsf{RE.Encode}$ can generate the garbled table for each gate *in parallel*, it follows that each output bit of the randomized encoding in time independent of the number of gates in the circuit. □

[2] We note that there exist simple modifications to Yao's garbled circuits that allows one to evaluate with no error. For simplicity, we don't present them here.

2 Exponentially Inefficient IO (XIO)

A key step in the construction of indistinguishability obfuscators is to reduce the problem to constructing a rather weak version of obfuscation that is now called "exponentially efficient" IO, or XIO.[3] The reductions were first formulated as a reduction from constructing IO to constructing a functional encryption (FE) scheme, and were later refined and refactored into (a) a simple construction of XIO from functional encryption and (b) a construction of IO from XIO, which we will present now.

The key qualitative statement is this: it is trivial to IO a circuit $C : \{0,1\}^n \to \{0,1\}^m$ where the size of the obfuscated program is $O(2^n \cdot m)$: simply output the truth table. Somewhat surprisingly, essentially doing any better than this triviality gives us full-fledged IO (with a polynomial-time obfuscator and a polynomial-size obfuscated program).

We formulate two different versions of XIO: fast-XIO and slow-XIO. The former, fast-XIO, requires both that the obfuscator runs in time $|\mathsf{TT}_C|^{1-\epsilon}$ as well as that the size of the obfuscated circuit is $|\mathsf{TT}_C|^{1-\epsilon}$ where TT_C is the truth table of the circuit C and $\epsilon > 0$ is some constant. The latter, slow-XIO, allows the obfuscator to run for a long time, that is, polynomial in $|\mathsf{TT}_C|$, but the obfuscated circuit has to be shorter than the truth table. Both imply IO, with different additional assumptions: fast-XIO plus OWF gives us IO, whereas slow-XIO requires LWE to give us IO (as far as we know).

2.1 XIO: Two Flavors

Definition 3. *For an absolute constant $\alpha > 0$, a probabilistic algorithm* XIO *is a $(T(\lambda), \epsilon(\lambda))$-secure α-exponentially efficient obfuscator (or α-XIO) if it takes as input a* Boolean *circuit $C : \{0,1\}^n \to \{0,1\}$ and outputs an obfuscated circuit $\widehat{C}$ such that:*

Functionality: *For all $x \in \{0,1\}^n$, $C(x) = \widehat{C}(x)$.*

Security: *For all n, all $\mathsf{poly}(2^n) \cdot T(\lambda)$-time distinguishers D, and all pairs of functionally equivalent circuits $C_0, C_1 : \{0,1\}^n \to \{0,1\}$ of the same size,*

$$\left| \Pr[D(1^\lambda, \mathsf{XIO}(C_0)) = 1] - \Pr[D(1^\lambda, \mathsf{XIO}(C_1)) = 1] \right| \leq \epsilon(\lambda)$$

Efficiency (size): *There is a fixed polynomial function* poly *such that the size of the obfuscated program $\widehat{C}$ is*

$$\mathsf{poly}(\lambda, |C|) \cdot 2^{(1-\alpha)n} .$$

Slow-XIO: *There is a fixed polynomial function* poly *such that the runtime of the obfuscator* XIO *is* $\mathsf{poly}(\lambda, 2^n)$.[4]

We will call this variant slow-XIO. *Additionally, one could require the following.*

Fast-XIO: *There is a fixed polynomial function* poly *such that the runtime of the obfuscator* XIO *is*

$$\mathsf{poly}(\lambda, |C|) \cdot 2^{(1-\alpha)n} .$$

This variant will be referred to as fast-XIO. *Note that this requirement subsumes the efficiency condition above.*

[3]The term "exponentially efficient", in our view, should have been called "exponentially inefficient", but that is a matter of personal taste.

[4]We do not explicitly mention polynomial dependence on $|C|$ here, as $|C| \leq 2^n$, without loss of generality. Furthermore, note that the dependence in n is $\text{poly}(2^n) = 2^{O(n)}$, not $2^{\text{poly}(n)}$.

Remark We note that restricting our definition to *Boolean* circuits is intentional as we wish to parameterize efficiency guarantees on a single parameter, namely the *input length* of the circuit, rather than both the input and output lengths. It is not hard to see that any circuit $C' : \{0,1\}^n \to \{0,1\}^m$ can be equivalently computed as $C(x,i) = C'(x)_i$ where $C : \{0,1\}^n \times \{0,1\}^{\log m} \to \{0,1\}$.

Remark In the rest of this exposition, we will focus on constructing IO from fast-XIO (and one-way functions).

3 IO from Fast-XIO and OWF

We present a construction of IO from (sub-exponentially secure) fast-XIO and OWF.

Preparing for the Compiler: Locally Computable XIO. Anticipating our compiler, we will show how to take any XIO scheme and make it computable in parallel. That is, take any XIO scheme for a circuit $C : \{0,1\}^n \to \{0,1\}$ where both the time to compute the obfuscated program as well as the size of the obfuscated program are upper-bounded by

$$\mathsf{size}_n = (\lambda \cdot |C|)^\beta \cdot 2^{n(1-\alpha)} .$$

where $\beta > 0$ and $0 < \alpha \leq 1$ are constants associated to the XIO scheme.

Something that will come up later is the complexity of computing *each bit* of the obfuscated program. A priori, there is no way to bound this beyond the trivial, namely size_n. We would like to construct a better XIO scheme XIO' where the size of the obfuscated program as well as the time to compute it are slightly larger, by a factor polynomial in the security parameter, but the complexity of computing each bit of the obfuscated program is a mere $\mathsf{poly}(\lambda)$. These requirements immediately suggest a randomized encoding scheme, and indeed that is what we will use.

We transform the obfuscator circuit $\mathsf{XIO}(\cdot\,;\,\cdot)$ whose size and output length are both upper bounded by size_n, into the circuit

$$\mathsf{XIO}'(C; r, r') = \mathsf{RE.Encode}_{\mathsf{XIO}}\big(C, r; r'\big)$$

where, on the left, r and r' are treated as the randomness for XIO' and on the right, one outputs a randomized encoding (RE) of the computation of XIO on input (C, r), using r' as the randomness for RE. The output length of XIO' is

$$\lambda^{\beta'} \cdot \mathsf{size}_n \leq (\lambda \cdot |C|)^{\beta+\beta'} \cdot 2^{n(1-\alpha)}$$

for some constant $\beta' > 0$. In other words, the obfuscated program becomes slightly larger. However, now, each output bit of XIO', i.e. each bit of the obfuscated program, can be computed using a circuit of size $\lambda^{\beta''}$ for some constant $\beta'' > 0$ as well.

Let $C' \leftarrow \mathsf{XIO}'(C; r, r')$. To evaluate C' on input x:

- first RE-decode C' into $C \in \mathsf{XIO}(C; r)$;
- then evaluate C on x.

By the simulation security of the randomized encoding, since $\mathsf{XIO}(C_0) \approx_c \mathsf{XIO}(C_1)$ for two circuits $C_0 \equiv C_1$, we have $C_0' \approx_c C_1'$ as well.

Remark To be completely precise, we need the XIO circuit to be uniform in the sense that given i, one can output the i-th gate of the circuit quickly, i.e. in time independent of the size of the circuit. For the sake of conciseness of presentation, we elide this important detail here.

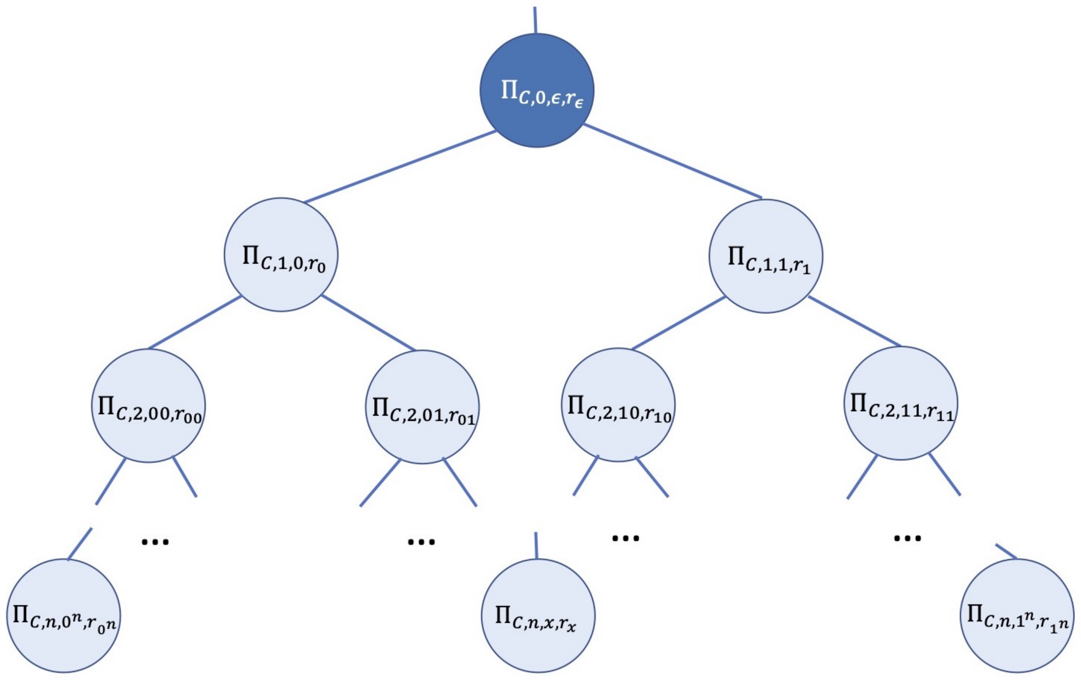

Figure 1: The construction of IO from XIO. Each Π is a program obfuscated using fast-XIO. The program Π_{C,i,x,r_x} outputs two programs $\Pi_{C,i+1,x0,r_{x0}}$ and $\Pi_{C,i+1,x1,r_{x1}}$. The IO obfuscator itself outputs $\Pi_{C,0,\varepsilon,r_\varepsilon}$. The IO evaluator, on input $x \in \{0,1\}^n$, generates programs $\Pi_{C,i,x_i,\cdot}$ iteratively starting from $\Pi_{C,0,\varepsilon,r_\varepsilon}$. Finally, computing the program $\Pi_{C,n,x,\cdot}$ on the empty input results in $C(x)$.

The Compiler. We will henceforth assume that we have a locally computable XIO scheme XIO.

At a high level, the compiler (implicitly!) generates a depth-n tree of circuits (i.e. with 2^n leaves) where each program $\Pi_{C,i,\mathbf{x},r}$ is parameterized by the circuit C in question, an index $i \in \{0,1,\ldots,n\}$, a string $\mathbf{x} \in \{0,1\}^i$ and a random string $r \in \{0,1\}^\lambda$.

- For $i = n$, the circuit $\Pi_{C,n,\mathbf{x},r}$ takes no input, computes C on input $\mathbf{x}$ and outputs $C(\mathbf{x})$.
- For $0 \le i < n$, the circuit $\Pi_{C,i,\mathbf{x},r}$ works as follows.

  Let $\mathsf{size}_{i+1} = |\mathsf{XIO}(\Pi_{C,i+1,\mathbf{x},r})|$ denote the size of the obfuscated program at level $i+1$. $\Pi_{C,i,\mathbf{x},r}$ takes as input a pair $(b,j) \in \{0,1\} \times \{0,1\}^{\log \mathsf{size}_{i+1}}$, and

  1. Computes
  $$(r_0, s_0, r_1, s_1) \leftarrow G(r)$$
  where $G : \{0,1\}^\lambda \to \{0,1\}^{4\lambda}$ is a PRG; and
  2. Outputs the j^{th} bit of the obfuscated program
  $$\mathsf{XIO}(\Pi_{C,i+1,\mathbf{x}\|b,r_b}; s_b)\ .$$
- Output $C' \leftarrow \mathsf{XIO}(\Pi_{C,0,\varepsilon,r})$ for a random string $r \leftarrow \{0,1\}^\lambda$. Here, ε is the empty string.

Let $\mathbf{x} = x_1 x_2 \ldots x_n$ be the bits of $\mathbf{x}$. To evaluate the obfuscated program on input $\mathbf{x}$, run C' on input $(x_1, \star)$ to get the obfuscated program $\Pi_{C,1,x_1,\cdot}$. Evaluate that on input $(x_2, \star)$ to get $\Pi_{C,2,x_1x_2,\cdot}$. So on, until you get $\Pi_{C,n,\mathbf{x},\cdot}$, evaluating which on an empty input, results in $C(\mathbf{x})$. Correctness should be clear.

Analysis: Size. Let us analyze the size of $\Pi_{C,0,\varepsilon,r}$ by induction. Note that we want this to be the polynomial in the size of the circuit and the security parameter. Let us start with some notation.

- Let the XIO parameters be (α, β, γ). That is, the size of an obfuscated circuit C on n bits of input is at most
$$(\lambda \cdot |C|)^\beta \cdot 2^{n(1-\alpha)}$$
and each bit of the obfuscated circuit can be computed with a circuit of size $\lambda^\gamma + n$.
- Let $\overline{\Pi}_{C,i,\mathbf{x},r}$ denote the XIO obfuscation of the circuit $\Pi_{C,i,\mathbf{x},r}$.
- Let $\overline{\mathsf{size}}_i$ (resp. size_i) denote the size of the circuits $\overline{\Pi}_{C,i,\mathbf{x},r}$ (resp. $\Pi_{C,i,\mathbf{x},r}$) for $\mathbf{x} \in \{0,1\}^i$.

The base case is the circuits at the leaves. The unobfuscated circuit $\overline{\Pi}_{C,n,\mathbf{x},r}$, where $\mathbf{x} \in \{0,1\}^n$, computes C on input $\mathbf{x}$ and outputs the result. Thus,

$$\mathsf{size}_n = |C|, \quad |\mathsf{TT}_n| = 1, \quad \text{and} \quad \overline{\mathsf{size}}_n = (\lambda \cdot \mathsf{size}_n)^\beta \cdot |\mathsf{TT}_n|^{1-\alpha} \le (\lambda \cdot |C|)^\beta$$

For the induction, assume that the size of the obfuscated circuit at depth $i+1$ is $\overline{\mathsf{size}}_{i+1}$. The program $\Pi_{C,i,\mathbf{x},r}$ at depth i computes

- The pseudorandom generator G on input $r \in \{0,1\}^\lambda$ to get a string of length 4λ. Call it (r_0, s_0, r_1, s_1). This takes a circuit of size λ^δ for some constant $\delta > 0$; and

- Runs the XIO obfuscator on input (b, j) and randomness s_b to produce the j^{th} bit of the obfuscated circuit $\overline{\Pi}_{C,i+1,\mathbf{x}\|b,r_b}$. This takes a circuit of size λ^γ, by the local computability property of our XIO scheme. Note that this is where our earlier preparation ends up being useful.

So, the size of $\Pi_{C,i,\mathbf{x},r}$ is

$$\mathsf{size}_i \le \lambda^\delta + \lambda^\gamma + |C|$$

and thus, the size of $\overline{\Pi}_{C,i,\mathbf{x},r}$ is

$$\overline{\mathsf{size}}_i = (\lambda^\delta + \lambda^\gamma + |C|)^\beta \cdot |\mathsf{TT}_i|^{1-\alpha} \le (\lambda^\delta + \lambda^\gamma + |C|)^\beta \cdot (2 \cdot \overline{\mathsf{size}}_{i+1})^{1-\alpha}$$

It is not hard to see (through a tedious calculation) that this is at most

$$2^{1/\alpha} \cdot (\lambda^\delta + \lambda^\gamma + |C|)^{\beta/\alpha} + \overline{\mathsf{size}}_{i+1} = \mathsf{poly}(\lambda, |C|) + \overline{\mathsf{size}}_{i+1}$$

The point is that the dependence on $\overline{\mathsf{size}}_{i+1}$ is *additive*. Thus, the size of $\Pi_{C,0,\varepsilon,r}$ is at most

$$n \cdot 2^{1/\alpha} \cdot (\lambda^\delta + \lambda^\gamma + |C|)^{\beta/\alpha} + \overline{\mathsf{size}}_n = n \cdot 2^{1/\alpha} \cdot (\lambda^\delta + \lambda^\gamma + |C|)^{\beta/\alpha} + \mathrm{poly}(\lambda, |C|) = \mathrm{poly}(\lambda, |C|)$$

as required.

The Tedious Calculation. The essential idea is to analyze the recursion

$$T(i) = c \cdot T(i-1)^{1-\alpha}$$

to find $T(n)$, where $T(1) = 1$, say, and c is a constant. Now, $T(2) = c$ and $T(3) = c^{1+(1-\alpha)}$, $T(4) = c^{1+(1-\alpha)+(1-\alpha)^2}$, and so on, which converges to $c(n) \approx c^{1/\alpha}$.

Analysis: Security. The proof proceeds by a *very* delicate hybrid argument that proceeds level by level. Let $C_0 \equiv C_1$ be two circuits of the same size and functionality.

First, consider the 2^n programs $\overline{\Pi}_{C_b,n,\mathbf{x},r_\mathbf{x}}$ for $b \in \{0,1\}$; that is, leaves of the tree computing either C_0 or C_1. By a straightforward hybrid argument, losing a factor of 2^n, we have

$$\left(\overline{\Pi}_{C_0,n,\mathbf{x},r_\mathbf{x}} \leftarrow \mathsf{XIO}(\Pi_{C_0,n,\mathbf{x},r_\mathbf{x}})\right)_{\mathbf{x}\in\{0,1\}^n} \approx_c \left(\overline{\Pi}_{C_1,n,\mathbf{x},r_\mathbf{x}} \leftarrow \mathsf{XIO}(\Pi_{C_1,n,\mathbf{x},r_\mathbf{x}})\right)_{\mathbf{x}\in\{0,1\}^n}$$

where $\mathbf{r_x}$ are randomly chosen. This is the base case. Now, assuming this, we need to show that the 2^{n-1} programs one level up are also computationally indistinguishable. Towards this, the following lemma is useful.

Lemma 4 (PIO Lemma)**.** *Given a probabilistic circuit $C(x; r)$, consider the circuit*

$$D_{C,K}(x) = C(x; \mathsf{PRF}(K, x))\,.$$

where PRF *is a puncturable PRF. For any two probabilistic circuits C_0 and C_1 such that for every x, $C_0(x) \approx_\epsilon C_1(x)$, we have*

$$\mathcal{O}(D_{C_0,K}) \approx_{O(2^n\epsilon)} \mathcal{O}(D_{C_1,K})\,.$$

Notice that the lemma loses a factor of $O(2^n\epsilon)$ in the distinguishing advantage, i.e. an adversary that distinguishes the obfuscated program with advantage ϵ' will give us an adversary that distinguishes between the output distributions of the original circuits C_0 and C_1 for some x, with advantage $\epsilon'/2^n \ll \epsilon'$. This suggests that the proof should proceed via a hybrid argument with $O(2^n)$ hybrids, which it indeed does.

Proof. We go through hybrids, one input at a time. Consider a hybrid circuit

$$C^{(i)}(x;r) = \begin{cases} C_1(x;r) & \text{if } x < i \\ C_0(x;r) & \text{otherwise} \end{cases}$$

so that $C^{(0)} \equiv C_0$ and $C^{(2^n)} \equiv C_1$. Consider

$$D^{(i)}_{C_0,C_1,K} = D_{C^{(i)},K} .$$

We will show that for all $i < 2^n$,

$$\mathcal{O}(D^{(i)}_{C_0,C_1,K}) \approx_c \mathcal{O}(D^{(i+1)}_{C_0,C_1,K})$$

The difference between the programs is that on input i, the left program computes $C_0(i;\mathsf{PRF}(K,i))$ and the right program computes $C_1(i;\mathsf{PRF}(K,i))$. First, puncture the PRF at i and hardcode $\mathsf{PRF}(K,i)$, where indistinguishability follows from IO security. Next, change the hardcoded PRF value to random, where indistinguishability follows from PRF security. Finally, use the indistinguishability of the samples $C_0(i;r)$ and $C_1(i;r)$ to switch between C_0 and C_1. (Slightly more formally, one would apply these substitutions, swap $C_0(i;r)$ with $C_1(i;r)$, and then apply these substitutions backwards to show indistinguishability between $\mathcal{O}(D^{(i)}_{C_0,C_1,K})$ and $\mathcal{O}(D^{(i+1)}_{C_0,C_1,K})$.)

□

Now, back to the proof of security of the compiler. Since we know that the level-n programs are computationally indistinguishable and each level-$(n-1)$ program outputs two level-n programs, we can use the above lemma to argue that the level-$(n-1)$ programs are indistinguishable as well. Continuing along this way, we get that the root programs are computationally indistinguishable. We lose a factor of

$$2^n \cdot 2^{n-1} \cdot \ldots 2 \approx 2^{n^2}$$

in security along the way. However, setting the security parameter λ to be significantly larger than n^2 and assuming the subexponential security of the underlying primitives, i.e. the XIO scheme and the puncturable PRF, does the job for us.

References

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

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