Indistinguishability Obfuscation from Well-Founded Assumptions

By Aayush Jain, Huijia Lin, and Amit Sahai

Abstract
At least since the initial public proposal of public-key cryptography based on computational hardness conjectures, cryptographers have contemplated the possibility of a “one-way compiler” that translates computer programs into “incomprehensible” but equivalent forms. And yet, the search for such a “one-way compiler” remained elusive for decades. We examine a formalization of this concept with the notion of indistinguishability obfuscation (iO). Roughly speaking, iO requires that the compiled versions of any two equivalent programs (with the same size and running time) be indistinguishable from any efficient adversary. Finally, we show how to construct iO in such a way that we can prove the security of our iO scheme based on well-studied computational hardness conjectures in cryptography.

1. INTRODUCTION
Consider the polynomial \( f_1(x, y) \in \mathbb{Z}[x, y] \) that is computed as follows:

\[
f_1(x, y) = (x + y)^{16} - (x - y)^{16}
\]

Alternatively, contemplate the polynomial \( f_2(x, y) \in \mathbb{Z}[x, y] \) that is computed via:

\[
f_2(x, y) = 32x^{16}y + 1120x^{15}y^2 + 8736x^{14}y^3 + 22880x^9y^5
\]

\[
+ 22880x^7y^6 + 8736x^5y^7 + 1120x^3y^8 + 32xy^{15}
\]

A calculation shows that \( f_1 \) and \( f_2 \) are in fact the same polynomial, computed in two different ways. Indeed, the expressions \( f_1 \) and \( f_2 \) above are special cases of arithmetic circuits, which precisely represent “ways to compute a polynomial.”

What if we wanted to hide all implementation choices made when creating such an arithmetic circuit for a particular polynomial? An easy way to do that would be to first convert our polynomial into a canonical form and then implement the canonical form as an arithmetic circuit. Indeed, the description of \( f_2 \) above can be seen as a canonical representation of the polynomial as a sum of monomials with regard to a natural monomial ordering. However, as this example illustrates, canonical forms can be substantially more complex than other implementations of the same polynomial. For polynomials in \( n \) variables, the loss in efficiency can be exponential in \( n \). This would often make computing the canonical form—or indeed, even writing it down—infeasible.

1.1. A pseudo-canonical form
Given that computing, canonical forms can be infeasible, what is there to do? Here, following \( i \), we draw an analogy to the notion of pseudo-randomness. When truly random values are not available, we can instead aim to produce values that “look random” by means of a pseudo-random generator. That is, we require that no efficient algorithm can distinguish between truly random values and the output of our pseudo-random generator.

Now, for two arithmetic circuits \( g_1 \) and \( g_2 \) that compute the same underlying polynomial, a true canonical form \( \text{Canonical}(g_i) \) would be identical to the canonical form of \( \text{Canonical}(g) \). Instead, we would ask that a pseudo-canonical form \( \text{PseudoCanonical}(g_i) \) would simply be indistinguishable from the pseudo-canonical form \( \text{PseudoCanonical}(g) \), to all efficient algorithms that were given \( g_1 \) and \( g_2 \) as well. Observe that unless there are actual efficiently computable canonical forms for all arithmetic circuits—which we do not believe to be true—it must be that such a \( \text{PseudoCanonical} \) operator is randomized, and outputs a probability distribution over arithmetic circuits computing the same polynomial.

1.2. The computing lens
The classic theory of computation tells us that general computer programs can be converted into equivalent polynomials (albeit over finite fields, which we will focus on implicitly in the sequel). So the pseudo-canonicalization question posed above is equivalent to the pseudo-canonicalization question for general computer programs. Indeed, the question of hiding implementation details within a computer program has a long history, dating at least as far back as the groundbreaking 1976 work of Diffie and Hellman introducing the concept of public-key cryptography. Historically, this problem has been called “program obfuscation,” albeit it was typically discussed in an ill-defined form. Discussed in these vague terms, it was folklore that truly secure program obfuscation would have revolutionary applications to computing, especially for securing intellectual property. The work of Barak et al.\(^7\) gave a formal treatment of this problem and proved the impossibility of strong forms of general-
purpose program obfuscation. This work also formalized the pseudo-canonicalization problem discussed above via the notion of indistinguishability obfuscation \((iO)\). Writing now in the language of Boolean circuits, we define the problem below:

**Definition 1.** (Indistinguishability Obfuscator \((iO)\) for Circuits.) A probabilistic polynomial-time algorithm \((iO)\) is called a secure indistinguishability obfuscator for polynomial-sized circuits if the following holds:

- **Completeness:** For every \(\lambda \in \mathbb{N}\), every circuit \(C\) with input length \(n\), every input \(x \in \{0,1\}^n\), we have that
  \[
  \Pr\left[ \tilde{C}(x) = C(x) : \tilde{C} \leftarrow iO(1^\lambda, C) \right] = 1.
  \]

- **Indistinguishability:** For every two ensembles \(\{C_{\lambda,1}\}_{\lambda \in \mathbb{N}}\) and \(\{C_{\lambda,0}\}_{\lambda \in \mathbb{N}}\) of polynomial-sized circuits that have the same size, input length, and output length, and are functionally equivalent, that is, \(\forall x \in \{0,1\}^n\), \(C_{\lambda,0}(x) = C_{\lambda,1}(x)\), for every input \(x\), the distributions \(iO(1^\lambda, C_{\lambda,0})\) and \(iO(1^\lambda, C_{\lambda,1})\) are computationally indistinguishable, denoted as:
  \[
  \{iO(1^\lambda, C_{\lambda,0})\} \approx_r \{iO(1^\lambda, C_{\lambda,1})\}.
  \]

It means or every efficient polynomial-time algorithm \(D\), for every constant \(c > 0\), there exists a constant \(\lambda_0 \in \mathbb{N}\), such that for all \(l > \lambda_0\), we have:

\[
\Pr\left[ D(iO(1^\lambda, C_{\lambda,0})) = 1 \right] - \Pr\left[ D(iO(1^\lambda, C_{\lambda,1})) = 1 \right] \leq \frac{1}{\lambda^c}.
\]

As we discuss in Section 1.4, indeed \(iO\) as a formalization of pseudo-canonicalization lived up to the folklore promise of software obfuscation: there was, and still is, a large research community studying novel applications of \(iO\).

In contrast, demonstrating the feasibility of constructing \(iO\) proved far more challenging. Often one expects that theory will lag behind the practice, and given the folklore promise of software obfuscation, one might expect that over the years perhaps clever programmers had come up with heuristic approaches to software obfuscation that resisted the attack. The reality is the opposite. Indeed, in 2021 the third annual White Box Cryptography contest was held to evaluate heuristic methods for software obfuscation, and every one of the 97 submitted obfuscations was broken before the contest ended.\(^3\)

A large body of theoretical work, starting with the pioneering work of Garg et al.,\(^4\) has attempted to construct \(iO\) using mathematical tools. However, before the result\(^5\) by the authors of this article, all previous mathematical approaches to constructing \(iO\) relied on new, unproven mathematical assumptions, many of which turned out to be false (see Jain et al.\(^6\) for a list of prior constructions.)

We would like to build \(iO\) whose security rests upon cryptographic hardness assumptions that have stood the test of time, have a long history of study, and are widely believed to be true. The main result of our work\(^7\) is the construction of an \(iO\) scheme from three well-studied assumptions. We discuss this in more detail next.

**Theorem 1.** Under the following assumptions\(^8,9,10\):

- the Learning Parity with Noise (LPN) assumption over general prime fields \(\mathbb{Z}_p\) with polynomially many LPN samples and error rate \(1/k^2\), where \(k\) is the dimension of the LPN secret, and \(\delta > 0\) is any constant;
- the existence of a Boolean Pseudo-Random Generator (PRG) in \(\text{NC}^0\) with stretch \(n^{1+\epsilon}\), where \(n\) is the length of the PRG seed, and \(\epsilon > 0\) is any constant;
- the Decision Linear (DLIN) assumption on symmetric bilinear groups of prime order.

Indistinguishability obfuscation for all polynomial-size circuits exists.

The three assumptions above (discussed further in Section 1.3) are based on computational problems with a long history of study, rooted in complexity, coding, and number theory. Further, they were introduced for building basic cryptographic primitives (such as public key encryption), and have been used for realizing a variety of cryptographic goals that have nothing to do with \(iO\).

**1.3. Assumptions in more detail**

We now describe each of the assumptions we need in more detail and briefly survey their history.

**The DLIN Assumption:** The Decisional Linear assumption (DLIN) is stated as follows: For an appropriate \(l\)-bit prime \(p\), two groups \(G\) and \(G_t\) are chosen of order \(p\) such that there exists an efficiently computable nontrivial symmetric bilinear map \(e : G \times G \rightarrow G_t\). A canonical generator \(g\) for \(G\) is also computed. Following the tradition of cryptography, we describe the groups above using multiplicative notation, even though they are cyclic. The DLIN assumption requires that the following computational indistinguishability holds:

\[
\{\langle g^s, g^r, g^{sr}, g^{s^2}, g^{r^2} \rangle | x, y, r, s \leftarrow \mathbb{Z}_p \} \approx_r \{\langle g^s, g^r, g^{sr}, g^{s^2}, g^{r^2} \rangle | x, y, r, s, z \leftarrow \mathbb{Z}_p \}
\]

This assumption was first introduced in the 2004 work of Boneh et al.\(^10\) and instantiated using appropriate elliptic curves. Since then DLIN and assumptions implied by DLIN have seen extensive use in a wide variety of applications throughout cryptography, such as Identity-Based Encryption, Attribute-Based Encryption, Functional Encryption for degree 2 polynomials, Non-Interactive Zero-Knowledge, etc.

**The Existence of PRGs in \(\text{NC}^0\):** The assumption of the existence of a Boolean Pseudo-Random Generator PRG in \(\text{NC}^0\) states that there exists a Boolean function \(G : \{0,1\}^n \rightarrow \{0,1\}^m\) where \(m = n^{1+\epsilon}\) for some constant \(\epsilon > 0\), and where each output bit computed by \(G\) depends on a constant number of

\(^a\) For technical reasons, we need to hardness of these assumptions to be such that no polynomial-time adversaries have beyond sub-exponentially small advantage in breaking the hardness of the underlying problems.
input bits, such that the following computational indistinguishability holds:

\[ \{G(\sigma) | \sigma \leftarrow \{0,1\}^n\} \approx \{ y | y \leftarrow \{0,1\}^m\} \]

Pseudorandom generators are a fundamental primitive in their own right and have vast applications throughout cryptography. PRGs in NC^0 are tightly connected to the fundamental topic of Constraint Satisfaction Problems (CSPs) in complexity theory and were first proposed for cryptographic use by Goldreich\textsuperscript{16} 20 years ago. The complexity theory and cryptography communities have jointly developed a rich body of literature on the cryptanalysis and theory of constant-locality Boolean PRGs.

LPN over large fields: The Learning Parity with Noise LPN assumption over finite fields \( \mathbb{Z}_p \) is a decoding problem. The standard LPN assumption with respect to subexponential size modulus \( p \), dimension \( \ell \), sample complexity \( n \), and a noise rate \( r = 1/\ell^\delta \) for some \( \delta \in (0, 1) \), states that the following computational indistinguishability holds:

\[ \{ A, s \cdot A + e \mod p | A \leftarrow \mathbb{Z}_p^{n \ell}, s \leftarrow \mathbb{Z}_p^{n}, e \leftarrow \mathcal{D}_{\mathbb{F}_p}^{\ell}\} \]
\[ \approx \{ A, u | A \leftarrow \mathbb{Z}_p^{n \ell}, u \leftarrow \mathbb{Z}_p^{n}\} \]

Above \( e \leftarrow \mathcal{D}_s \) is a generalized Bernoulli distribution, i.e., \( e \) is sampled randomly from \( \mathbb{Z}_p \) with probability \( 1/\ell^\delta \) and set to be 0 with probability \( 1 - 1/\ell^\delta \). We consider polynomial sample complexity \( n(\ell) \), and the modulus \( p \) is an arbitrary subexponential function in \( \ell \).

The origins of the LPN assumption date all the way back to the 1950s (e.g., Gilbert\textsuperscript{11}): it has long been known that random linear codes possessed remarkably strong minimum distance properties. However, since then, very little progress has been made inefficiently decoding random linear codes under random errors. The LPN over fields assumption above formalizes this, and was introduced over \( \mathbb{Z}_p \) for cryptographic uses in 1994,\textsuperscript{4} and formally defined for general finite fields and parameters in 2009,\textsuperscript{17} under the name “Assumption 2.”

While in Ishai et al.\textsuperscript{17} the assumption was used when the error rate was assumed to be a constant, in fact, polynomially low error (in fact \( \delta = 1/2 \)) has an even longer history in the LPN literature: it was used by Alekhnovich\textsuperscript{4} to construct public-key encryption with the field \( \mathbb{Z}_p \), and used to build public-key encryption over \( \mathbb{Z}_p \) in 2015.\textsuperscript{5} The exact parameter settings that we describe above, with both general fields and inverse polynomial error rate corresponding to an arbitrarily small constant \( \delta > 0 \) was explicitly posed by Boyle,\textsuperscript{1} in the context of building efficient secure two-party and multi-party protocols for arithmetic computations.

Recently, the LPN assumption has led to a wide variety of applications. A comprehensive review of known attacks on LPN over large fields, for the parameter settings we are interested in, was given in Boyle.\textsuperscript{11, 12} For our parameter setting, the running time of all known attacks is \( \Omega(2^n) \), for any choice of the constant \( \delta \in (0, 1) \) and for any polynomial number of samples \( n(\ell) \).

Finally, we mention that except for the DLIN assumption, the other two assumptions, LPN and PRG in NC^0, have search-to-decision reductions.

### 1.4. Applications of iO

The notion of iO occupies an intriguing and influential position in complexity theory and cryptography. Interestingly, if \( \text{NP} \subseteq \text{BPP} \), then iO exists for the class of all polynomial size circuits, because if \( \text{NP} \subseteq \text{BPP} \), then it is possible to efficiently compute a canonical form for any function computable by a polynomial-size circuit. This is in contrast to most powerful cryptographic objects, which typically imply the existence of one-way functions. A large body of work has shown that iO plus one-way functions imply a vast array of cryptographic objects, so much so that iO has been conjectured to be a “central hub”\textsuperscript{24} for cryptography. In addition, an impressive list of fascinating new cryptographic objects is only known under iO or related objects.

We highlight a small subset of these implications (see Jain et al.\textsuperscript{18} and the references therein): **multi-party non-interactive key exchange in the plain model, zero-knowledge succinct noninteractive arguments (ZK-SNARGs)** for any NP language, **multilinear maps** with bounded polynomial multilinear degrees, witness encryption for any NP language, **functional encryption** supporting all polynomial-sized circuits, and **fully homomorphic encryption schemes (FHE)** for unbounded-depth polynomial-size circuits.

### 1.5. Open problems

Our workplaces iO on firm foundations with respect to the assumptions is based on, thereby answering the main feasibility question for the primitive (until we resolve the P vs. NP question). However, there are many important open questions that remain to be answered. One of the main questions is whether there is a construction that is very simple to describe and efficient to implement in practice. Our construction is quite complex, relies on a number of steps, and is recursive in nature. As a consequence, the actual asymptotic complexity is some large polynomial. Simplifying and finding more efficient paths to building iO from well-studied assumptions is one of the most important problems that must be solved, toward the vision of using iO as a universal tool.

### 2. OVERVIEW: HOW TO CONSTRUCT iO?

In the remaining of the article, we describe a very high level overview of the main technical ideas implying iO. For simplicity of exposition, we choose the simplest path to iO that we are aware of. This overview is based on a combination of ideas from Jain et al.\textsuperscript{14, 19} which, however, would require one additional assumption to the three stated above (See Theorem 1)—namely, the Learning With Errors (LWE)\textsuperscript{23} assumption. Since the exact technical reasons for needing LWE is not important for this overview, and this assumption is actually unnecessary,\textsuperscript{19} we do not discuss it in detail here.

#### 2.1. Preliminaries

Let’s start with introducing some basic notation. Let \( \text{size}(X) \)
indicates the length of the binary description of an object \( X \) (e.g., a string, a circuit, or truth table). Throughout, we consider Boolean functions or circuits or algorithms mapping \( n \)-bit binary strings to \( m \)-bit binary strings, for some \( n, m \in \mathbb{Z}^+ \). Let \( \text{time}(A, x) \) denote the running time of an algorithm (or circuit) \( A \) on an input \( x \) (in the case of a circuit \( C \), \( \text{time}(C, x) \) is the same as \( \text{size}(C) \)). We say that an algorithm or circuit is efficient if its running time is bounded by a (fixed) polynomial in the length of the input, that is, \( \text{time}(C, x) = \text{size}(x)^k \) for some positive integer \( k \in \mathbb{Z}^+ \). When we only care about the existence of a constant and the concrete value is not important, we write \( \Theta(1) \) in place of that constant, e.g., \( \text{time}(C, x) = \text{size}(x)^\Theta(1) \) (following the big-O notation in complexity theory).

Restating our goal, we want to design an efficient randomized algorithm, called the obfuscator \( \mathcal{O} \), that given a Boolean circuit \( C: \{0,1\}^n \to \{0,1\} \) with size \( s \leq n^{\Theta(1)} \), referred to as the original circuit, outputs another Boolean circuit \( \hat{C}: \{0,1\}^n \to \{0,1\} \), called the obfuscated circuit, such that the following three properties hold:

- **Correctness:** the obfuscated circuit \( \hat{C} \) is functionally equivalent to the original circuit \( C \), denoted as \( \hat{C} \equiv C \), meaning that for every \( x \in \{0,1\}^n \), \( \hat{C}(x) = C(x) \). Correctness must hold no matter what random coins the obfuscator \( \mathcal{O} \) used.
- **Efficiency:** the obfuscator is efficient, meaning \( \mathcal{O} \) runs in time polynomial in the size of the original circuit, namely, \( \text{time}(C, C) = \text{size}(C)^\Theta(1) \).
- **Security:** the obfuscated circuit \( \hat{C} \) hides the implementation details in the original circuit \( C \). That is, for every two, equally-sized and functionally-equivalent circuits \( C_0 \) and \( C_1 \) (i.e., \( \text{size}(C_0) = \text{size}(C_1) \) and \( C_0 \equiv C_1 \)), the obfuscated circuits \( \hat{C}_0 \) and \( \hat{C}_1 \) are computationally hard to distinguish (as defined in Definition 1; see also Jain et al.\(^{13}\)).

Next, we give an informal overview of how to construct \( i\mathcal{O} \) from well-studied assumptions.

### 2.2. Simplifying the task of \( i\mathcal{O} \)

Perhaps the simplest starting point is the following: If there is no restriction on the time the obfuscator \( \mathcal{O} \) can take, then there is an extremely intuitive obfuscator: the obfuscated circuit is the truth table of the original circuit. The truth table \( \mathcal{TT}_C \) of a Boolean circuit \( C \) is an array indexed by inputs, where \( \mathcal{TT}_C[x] = C(x) \). It can be computed in time \( 2^n \cdot s \) if the input length \( n \) is \( \|n\| \) and the circuit size is \( s \). Perfect security comes from the fact that for any two functionally equivalent circuits \( C_0 \equiv C_1 \), their truth tables are identical \( \mathcal{TT}_{C_0} = \mathcal{TT}_{C_1} \) and hence impossible to distinguish.

To put simply, a truth table is a canonical form of all circuits producing it. While outputting the truth table satisfies the correctness and security requirement of \( i\mathcal{O} \), the obvious flaw with this is that the obfuscator is far from efficient: The running time of an \( i\mathcal{O} \) scheme should be \( s^{\Theta(1)} \), rather than \( 2^n \cdot s \). The input length \( n \) of a Boolean circuit can be close to its size \( s \), and hence the time to compute a truth table is exponentially large! This inefficiency is likely inherent, as efficient methods of finding canonical forms of circuits imply the collapse of the polynomial hierarchy, which is widely conjectured to be false.

Therefore, to improve efficiency, we seek canonical forms that fool computationally limited adversaries. Naturally, we start with a more humble goal:

**Can we improve efficiency slightly to, say** \( 2^{(n+\varepsilon)} \cdot s^{\Theta(1)} \) **for some small \( \varepsilon > 0 \)? What does \( i\mathcal{O} \) with such non-trivial efficiency imply?**

#### Simplification 1: Obfuscation with non-trivial exponential efficiency.

\( i\mathcal{O} \) with non-trivial efficiency was studied in Lin et al.\(^{21}\). Surprisingly, they showed that a very modest improvement on efficiency – captured in the notion of exponential-efficiency \( i\mathcal{O} \) or \( xi\mathcal{O} \) for short – is actually enough to construct completely efficient (polynomial time) \( i\mathcal{O} \).

- **\( xi\mathcal{O} \)** is an obfuscator \( \mathcal{O} \) whose running time is still “trivial” (\( 2^n \cdot s^{\Theta(1)} \)), but outputs an obfuscated circuit \( \hat{C} \) of “non-trivial” size \( 2^{(n+\varepsilon)} \cdot s^{\Theta(1)} \) for some \( \varepsilon > 0 \).

We can think of \( xi\mathcal{O} \) (as well as \( i\mathcal{O} \)) as a special kind of encryption method, where the obfuscated circuit \( \hat{C} \) is a “ciphertext” of the original circuit \( C \), also denoted as \( \text{spCT}(C) = \hat{C} \), satisfying that

- the ciphertext \( \text{spCT}(C) \) hides all information about \( C \), except that it lets anyone with access to it learn the truth table of \( C \). This is unlike normal encryption which reveals no information about the encrypted message.
- The size of the ciphertext \( \text{spCT}(C) \) is \( 2^{(n+\varepsilon)} \) for some \( 0 < \varepsilon < 1 \). So it can be viewed as a (slightly) compressed version of the truth table (that reveals no other information of \( C \) to computationally limited adversaries).

The such special encryption scheme is known as functional encryption, which controls precisely which information of the encrypted message is revealed, and hides all other information. This notion is tightly connected with \( xi\mathcal{O} \) and \( i\mathcal{O} \), and, in fact, the implication of \( xi\mathcal{O} \) to \( i\mathcal{O} \) goes via the notion of functional encryption.

When viewing \( \text{spCT}(C) \) as a compressed version of the truth table. It becomes clear why even slight compression is powerful enough to imply \( i\mathcal{O} \): The idea is keeping compressing iteratively until the size of the special ciphertext becomes polynomial. The works of Ananth and Jain,\(^3\) Bitansky and Vaikuntanathan,\(^8\) and Lin et al.\(^{21}\) turn this high-level idea into an actual proof that \( xi\mathcal{O} \) implies \( i\mathcal{O} \) and allows us to focus on constructing \( xi\mathcal{O} \), or equivalently the special encryption described above.

#### Simplification 2: It suffices to obfuscate simple circuits.

\( b \) This is the step where the additional assumption of Learning With Errors (LWE) is used. However, as we showed in Jain et al.\(^{13}\) this assumption is unnecessary for \( i\mathcal{O} \).

\( c \) This implication, however, comes with a quantitative weakening in the security.
Unfortunately, despite the efficiency relaxation, it is still unclear how to obfuscate general Boolean circuits, which can be complex. Naturally, we ask:

Can we obfuscate simple sub-classes of circuits? What does xi0 for simple circuits imply?

It turns out that it suffices to focus on an extremely simple class of circuits $C = \text{NC}^0$, where $\text{NC}^0$ is the set of all circuits with constant output locality, meaning every output bit depends on a constant number of input bits. To do so, we will rely on two cryptographic tools, randomized encodings and Pseudo-Random Generators (PRGs).

**Randomized encoding in NC^0.** A Randomized Encoding (RE) scheme consists of two efficient algorithms (RE, Decode). It gives a way to represent a complex circuit $C(\cdot)$, by a much simpler randomized circuit $\text{RE}_c(\cdot; \cdot) := \text{RE}(C(\cdot); \cdot)$, such that,

- Correctness: for every input $x$, the output $p_x$ of $\text{RE}(C, x; r)$ produced using uniformly random coins $r$ encodes the correct output: In other words, there exists an efficient decoding algorithm Decode such that $\text{Decode}(p) = C(x)$.
- Security: $p_x$ reveals no information of $C$ beyond the output of $x$ to efficient adversaries.
- Simplicity: RE is a simple circuit by some measure of simplicity. Classic works showed that RE can be simply a NC^0 circuit, when assuming the existence of PRGs in NC^0 like we are.

The correctness and security of the randomized encoding suggest that, instead of directly obfuscating a general circuit $C$, we can alternatively obfuscate a circuit $D$ that on input $x$ outputs an encoding $p_x$, which reveals only $C(x)$. The potential benefit is that $D$ depending on RE and $C$ should be simply NC^0 circuit. Hence, it would suffice to construct xi0 for simple NC^0 circuits!

To make the above idea goes through, there are, however, a few wrinkles to be ironed out. The issue is that the security of randomized encoding only holds if an encoding $p_x$ is generated using fresh random coins. There are concrete attacks that learn information of $C$ beyond the outputs, when $p_x$ is generated using non-uniform random coins, or when two encoding $p_x$ and $p_y$ are generated using correlated random coins. If the truth table of the circuit $D$ contains an encoding $p_x$ for every input $x$ (i.e., $\text{TT}_x[p] = p_x$) the random coins for generating these encodings must be embedded in $D$, that is,

$$D = D_{c, \text{RE}, r_1, \cdots, r_n}, \quad \text{s.t.} \quad D(x) = \text{RE}(C, x; r_x)$$

Such a circuit $D$ has size at least $2^n$. In particular, we cannot hope to “compress” the random coins $r_1, r_2, \cdots, r_n$ into $2^{n-\epsilon}$ space, which is the target size of the obfuscated circuit. To resolve this problem, we will use a Pseudo-Random Generator.

**Pseudo-random generator in NC^0.** Recall that a pseudo-random generator is a Boolean function $\text{PRG}: \{0, 1\}^n \rightarrow \{0, 1\}^m$ that takes as input a uniformly random seed $\text{sd} \in \{0, 1\}^s$, and produces a polynomially longer output $r \in \{0, 1\}^m$, where $m = n^{1+\epsilon}$ for some constant $\epsilon > 0$, such that, $r$ is indistinguishable to a uniformly random $m$-bit string to computationally limited adversaries. There are several proposals of pseudo-random generators that can be computed by an NC^0 function.

Equipped with a pseudo-random generator in NC^0, we can replace uniformly random coins $r_1, r_2, \cdots, r_n$ with pseudorandom coins expanded from a much shorter seed $\text{sd}$ of length roughly $2^{n(1+\epsilon)} = 2^{n+\epsilon}$ for some $\epsilon > 0$. This gives a variant of the circuit $D$ above:

$$D' = D_{c, \text{RE}, \text{PRG}, \text{sd}}, \quad \text{s.t.} \quad D'(x) = \text{RE}(C, x; (r_x = \text{PRG}_x(sd))),$$

where $\text{PRG}(\text{sd})$ expands to output $r'_1, \cdots, r'_n$ and $r_x = \text{PRG}(\text{sd})$ is the $x$th chunk of the output. Thanks to the fact that both PRG and RE are in NC^0, so is $D'$.

Moving forward, it suffices to devise a way to encrypt $C$, $\text{sd}$ into a ciphertext $\text{ciphertext} = \text{spCT}(C, \text{sd})$, from which we can expand out the truth table of $D'$, while hiding all other information of $C$ and $\text{sd}$.

$$\text{spCT}(C, \text{sd}) \rightarrow \text{Expand} \rightarrow \text{TT}' \rightarrow (C, \text{sd}) \rightarrow \text{PRG} \rightarrow \text{TT}'$$

Note that the special encryption only needs to hide $(C, \text{sd})$, instead of the entire description of circuit $D$ since RE and PRG are public algorithms. Moreover, the truth table of $D'$ can be computed from $(C, \text{sd})$ via a function $G_{\text{RE}, \text{PRG}}$ in NC^0.

### 2.3. Special encryption for NC^0 mappings

In our works, we constructed the needed special encryption for general NC^0 mappings $G: \{0, 1\} \rightarrow \{0, 1\}^m$, where ciphertext $\text{spCT}(X)$ reveals the output of $G$ on $X$ while hiding all other information of $X$, such that size$(\text{spCT}(X)) = \text{size}(X) + m^{1-\epsilon}$ for some $\epsilon > 0$. In this subsection, we describe the first half of the ideas behind our construction, which connects the special encryption with bilinear pairing groups and a new object called a structured-seed pseudo-random generator. The second half of the ideas are explained in Section 2.5, describing the construction of a structured-seed pseudo-random generator.

**Connection with bilinear pairing groups.** As a starting point, suppose that the function $G$ is really simple; Simple enough so that it can be computed by a degree-$2$ polynomial mapping $Q: \mathbb{Z}_p \rightarrow \mathbb{Z}_p^m$ over the finite field $\mathbb{Z}_p$ for some prime modulus $p$. Namely,

$$\forall x \in \{0, 1\}^n, \quad G(x) = Q(x),$$

where,

$$Q(x) = \sum_{j, k} \alpha_{j, k} x_j x_k + \sum_i \beta_i x_i + \gamma \mod p.$$

Then, the special encryption can be implemented using bilinear pairing groups, as shown in Ananth and Sahai, and Lin.

**Bilinear pairing groups.** At a high-level, pairing groups
allows for computing quadratic polynomials over secret encoded values, and reveal only whether the output is zero or not. More specifically, it consists of cyclic groups $G_i, G_p$, and $G$, with generators $g_i, g_p,$ and $g$, respectively, and all of order $p$. $G_i$ and $G_p$ are referred to as the source groups and $G$, the target group. (In some instantiations, the two source groups are the same group, which is called symmetric pairing groups.) They support the following operations:

- **Encode:** For every group $G_i$, one can compute $g_i^x$ for $x \in \mathbb{Z}_p$. The group element $g_i^x$ is viewed as an encoding of $x$ in the group $G_i$.
- **Group Operation:** In each group $G_i$, one can perform the group operation to get $g_i^{x_1} \cdot g_i^{x_2} = g_i^{x_1 \cdot x_2}$, corresponding to “homomorphic” addition modulo $p$ in the exponent. Following the tradition of cryptography, we write the group operation multiplicatively.
- **Bilinear Pairing:** Given two source group elements $g_i^{x_1}$ and $g_i^{x_2}$, one can efficiently compute a target group element $g_T^{x_1 \cdot x_2}$, using the so-called pairing operation $e(g_i^{x_1}, g_i^{x_2}) = g_T^{x_1 \cdot x_2}$. This corresponds to “homomorphic” multiplication modulo $p$ in the exponent. However, after multiplication, we obtain an element in the target group that cannot be paired anymore.
- **Zero Testing:** Given a group element $g_i^x$, one can always tell if the encoded value is $x = 0$, by comparing the group element with the identity in $G_i$. Similarly, one can do equality testing to see if $g_i^x = g_i^y$ for any $c$.

Combining the above abilities gives a “rudimentary” special encryption scheme that supports the evaluation of degree-2 polynomials. A ciphertext of $X \in \mathbb{Z}_p$ includes encodings of every element $X_i$ in both source groups, $\{(g_i^{X_1}, \cdots, g_i^{X_n}), (g_z^{X_1}, \cdots, g_z^{X_n})\}$. Given these, one can “homomorphically” compute a quadratic mapping $Q$ to obtain encoding of the output $y = Q(X)$ in the target group $(g_T^{X_1}, \cdots, g_T^{X_n})$ (without knowing the encoded input $X$ at all); finally, if the output $y$ is a bit string, one can learn $y$ in the clear via zero-testing. In summary,

$$\{(g_i^{X_1}, \cdots, g_i^{X_n}), (g_z^{X_1}, \cdots, g_z^{X_n})\} \xrightarrow{\text{Expand}} Q(X), \text{if } Q(X) \in \{0,1\}^m.$$  

This fulfills the correctness of the special encryption.

**What about security?** The ciphertext must hide all information about $X$ beyond what’s revealed by $Q(X)$, for which we resort to the security of pairing groups. For simplicity of this overview, let’s gain some security intuition by assuming the strongest hardness assumptions pertaining to the pairing groups, known as the generic group model. Think of encoding as black boxes and the only way of extracting information (of the encoded values) is by performing (a combination of) the above four operations. In this model, given $g^a$ for a secret and random $x \in \mathbb{Z}_p$, no efficient adversary can learn $x$. Extending further, if $X$ were random (in $\mathbb{Z}_p$) subject to $Q(X) = y$, then the encoding would reveal only $y$ and hide all other information of $X$. The only issue is that $X$ in our example is not random – it is binary. Nevertheless, the works of Ananth and Sahai, and Lin designed clever ways of randomizing an arbitrary input $X$, to a random input $\bar{X}$ subject to the only condition that an appropriate quadratic mapping $\bar{Q}$ reveals the right output $\bar{Q}(\bar{X}) = Q(X)$. Hence, security is attained. We refer the reader to Ananth and Sahai, and Lin for details about how such randomization works.

**Challenges beyond Degree 2.** Unfortunately, the mapping $G_{\text{SE-PRG}}$ we care about can only be computed by polynomials of a degree much larger than 2. It is known that a Boolean function with output locality $l$ can be computed by a multilinear degree $l$ polynomial over $\mathbb{Z}_p$. However, the locality of Boolean PRG we need is at least $5$ and known randomized encodings have locality at least 4.

**Key idea: The preprocessing model.** To overcome this challenge, our first idea is to use preprocessing of inputs to help reduce the degree of computation. Instead of directly computing $G(X)$ in one shot, we separate it into two steps: First, the input is preprocessed $X_e = \text{pre}(X, r)$ in a randomized way using fresh random coins $r$, then the output is computed from the preprocessed input $y = Q(X_e)$. The idea is that the preprocessing can perform complex transformations on the input in order to help the computation later. The only constraint is that the preprocessing should not increase the size of its input too much, that is, $\text{size}(X_e) \sim \text{size}(X) + m^{1-\delta}$, for some $\delta > 0$. As such, it suffices to encrypt the preprocessed input $\text{spCT}(X_e)$, from which one can recover the desired output $y = G(X_e)$, by evaluating a (hopefully) simpler function $Q$. Unfortunately, because of the restriction on the size of $X_e$, it is unclear how preprocessing alone can help.

Our second idea is to further relax the preprocessing model to allow the preprocessed input to contain a public part and a secret part $X_e = (P, S)$. Importantly, the public part $P$ should reveal no information about $X$ to computationally limited adversaries. (In contrast, $P, S$ together reveals $X$ completely.) Moreover, we allow the second stage computation $Q(P, S)$ to have an arbitrary constant degree in $P$ and only restricts its degree on $S$ to 2, that is,

$$Q_i(P, S) = \sum_{j,k} \alpha_{j,k}(P) \cdot S_j \cdot S_k + \sum_k \beta_k(P) \cdot S_k + \gamma(P),$$

where $\alpha_{j,k}, b_k, g$ are constant degree polynomials over $\mathbb{Z}_p$. It turns out that the techniques alluded to above for special encryption for degree 2 computations can be extended, so that given ciphertext $\text{spCT}(S)$ and $P$ in the clear, one can homomorphically compute $Q$ and thereby learn the output $G(X)$.

In Jain et al. 18,19 we show how to compute any NC° Boolean function $G$ in such a preprocessing model, assuming the Learning Parity with Noises assumption over general fields, which completes the puzzle of iO. When applied to specific cryptographic tools, our techniques give interesting new objects. For instance, it converts any PRG in NC into what we call structured-seed PRG. Given a preprocessed seed $(P, S) = \text{PRG}(\text{sd}; \text{re})$, the structured-seed PRG expands out a polynomially longer output $r = \text{sprng}(P, S)$, where the computation has only degree-2 in the private seed $S$, and the output $r$ is pseudo-random given the public seed $P$. In the next section, we describe how to do preprocessing in the context
of constructing structured-seed PRG. The same ideas can be extended to handle general NC$^0$ functions.

### 2.4. Definition of structured-seed PRG

**Definition 2.** (Syntax of Structured-Seed Pseudorandom Generators (sPRG).) Let $i > 1$. A structured seed Boolean PRG (sPRG) with polynomial stretch $i$, is defined by the following PPT algorithms:

- $\text{IdSamp}(1^n, p)$: The index sampling algorithm takes as input the input length parameter $n$, and a prime $p$. It samples a function index $I$.
- $\text{SdSamp}(I)$ jointly samples two strings, a public seed and a private seed, $sd = (P, S)$ which are both vectors of dimension $\ell_{sd} = O(n)$ over $\mathbb{Z}_p^*$.
- $\text{Eval}(I, sd)$ computes a string in $\{0, 1\}^m$.

Looking ahead, the prime $p$, that we choose is set to the order of the bilinear group which has a bit length of $n^{\Omega(1)}$.

**Definition 3.** (Security of sPRG.) A structured-seed Boolean PRG, sPRG, satisfies security if for any constant $r > 0$, any function $p : \mathbb{N} \rightarrow \mathbb{Z}$ that takes as input a number $k \in \mathbb{N}$ and outputs a $k'$ bit prime $p(k)$, any $n \in \mathbb{N}$, with probability $1 - o(1)$ over $\text{IdSamp}(1^n, p(n)) \rightarrow I$ it holds that the following distributions are computationally indistinguishable.

$$\{I, P, \text{Eval}(I, P, S)\} \approx \{I, P, r\}$$

where $I$ is sampled by $\text{IdSamp}(1^n, p)$, $sd$ by $\text{SdSamp}(I)$, and $r$ sampled randomly from $\{0, 1\}^{m(n)}$.

**Definition 4.** (Complexity and Degree of sPRG.) Let $r > 0$, $d_i, d_s \in \mathbb{N}$ be any constant, and $p : \mathbb{N} \rightarrow \mathbb{Z}$ be any function that maps an integer $k$ into a $k'$ bit prime $p(k)$. An sPRG has degree $d_i$ in public seed $P$ and degree $d_s$ in $S$ over $\mathbb{Z}_p^*$, denoted as, $\text{sPRG} \in (\deg d_i, \deg d_s)$, if for every $I$ in the support of $\text{IdSamp}(1^n, p(n))$, there exists efficiently generateable polynomials $g_{i,1}, \ldots, g_{i,m}$ over $\mathbb{Z}_p$ such that:

- $\text{Eval}(I, (P, S)) = (g_{i,1}(P, S), \ldots, g_{i,m}(P, S))$, and
- The maximum degree of each $g_{i,j}$ over $P$ is $d_i$, and the maximum degree of $g_{i,j}$ over $S$ is $d_s$.

We remark that the above definition generalizes the standard notion of families of PRGs in two aspects: 1) the seed consists of a public part and a private part, jointly sampled and arbitrarily correlated, and 2) the seed may not be uniform. Therefore, we obtain the standard notion as a special case.

**Definition 5.** (Pseudo-Random Generators, Degree, and Locality.) A (uniform-seed) Boolean PRG (PRG) is an sPRG with a seed sampling algorithm $\text{SdSamp}(I)$ that outputs a public seed $P$ that is an empty string and a uniformly random private seed $S \sim \{0, 1\}^n$. Let $d, c \in \mathbb{N}$. The PRG has multilinear degree $d$ if for every $n \in \mathbb{N}$ and $f$ in the support of $\text{IdSamp}(1^n)$, we have that $\text{Eval}(I, sd)$ can be written as an $n(m(n))$-tuple of degree-$d$ polynomials over $\mathbb{Z}$ in $S$. It has constant locality $c$ if for every $n \in \mathbb{N}$ and $I$ in the support of $\text{IdSamp}(1^n)$, every output bit of $\text{Eval}(I, sd)$ depends on at most $c$ bits of $S$.

In what follows next we will give an overview of our construction of sPRG from the LPN assumption and the existence of PRG in NC$^0$. For ease of exposition, we will actually use Goldreich’s PRG candidate\cite{Goldreich} which is the most well-known conjectured PRG in NC$^0$. See Jain et al.\cite{Jain} for the construction based on any PRG in NC$^0$.

**Definition 6.** (Goldreich’s PRG.) A Goldreich PRG of locality $c$ mapping $n$ bits to $m$ bits is described using a predicate $f : \{0, 1\}^c \rightarrow \{0, 1\}$ and a hypergraph $H = \{Q_1, \ldots, Q_m\}$ where each $Q_i$ is a randomly chosen ordered subset of $[n]$ of size $c$. The index $I$ consists of $f$ and $H$. Further, on input $x \in \{0, 1\}^n$, PRG.Eval($I, x) = y$, where $y = (y_1, \ldots, y_m)$. Here each $y_i = f(x_{i_1}, \ldots, x_{i_c})$ where $Q_i = (i_1, \ldots, i_c)$.

### 2.5. sPRG overview

We start with a Goldreich PRG $\text{PRG} = (\text{IdSamp}, \text{Eval})$ with stretch $i$, associated with a $d$-local predicate $f : \{0, 1\}^d \rightarrow \{0, 1\}$ and a randomly sampled hypergraph $H$, i.e., $I = (f, H)$.

Observe that on any input $\sigma \in \{0, 1\}^*$, $y = \text{PRG.Eval}(I, s)$ can be computed by degree $d$ multilinear polynomials over $\mathbb{Z}$ as $f$ is a predicate evaluated on $d$ bits. Our high-level strategy is to somehow “preprocess” $s$ into two vectors $(P, S)$ of small dimension (preferably $O(n)$, but anything sublinear in $m$ works) such that $y = \{0, 1\}^n$ can be computed in $(\deg d, \deg 2)$. Thereby this will have the effect of transferring the complexity of computation to the public input. To achieve this, we pre-process $s$ into appropriate public and private seeds $(P, S)$ and leverage the LPN assumption over $\mathbb{Z}_p^*$ (which is the input to sPRG.IdSamp) to show that the seed is hidden.

Our first idea toward this is that we can “encrypt” the seed $s$ using LPN samples over $\mathbb{Z}_p^*$ as follows:

**Sample:** $A \leftarrow \mathbb{Z}_p^{m}, s \leftarrow \mathbb{Z}_p^{\ell d}, e \leftarrow \mathbb{D}_p^{\ell m}$ (p)

**Add to function index I:** $A$ to $P$

**Add to public seed P:** $b = sa + e + s$

where $\ell$ is set to be the dimension for the LPN samples. It is set so that $\ell d = n$. $D_\ell(p)$ is a distribution that samples randomly from $p$ with probability $r$, and 0 otherwise. Finally, $r = \ell d$.

It follows directly from LPN assumption that $(A, b)$ is pseudorandom and hides $s$. Furthermore, due to the sparsity of LPN noises, the vector $s + e$ differs from $s$ only at a $\ell^{-d}$ fraction of components – thus it is a sparsely erroneous version of the seed. For $i \in [m]$, let $f_i(s)$ be the locality $d$, degree $d$ polynomial over $\mathbb{Z}$ such that $y_i = f_i(s)$. Then, for $i \in [m]$ consider the following polynomial:

$$h_i(b, \overline{s}^{\overline{d}}) = f_i(b - sa)$$

Above we first interpret $f_i$ over $\mathbb{Z}$ into the field $\mathbb{Z}_p^*$ by simply reducing coefficients mod $p$, and then compute as given. Observe that $f_i$ is a degree $d$ polynomial in $b$ and $s$, therefore its degree over $b$ is $d$ and over $\overline{s}^{\overline{d} - 1}$ is two. Thus $h_i$ is a $(\deg d, \deg 2)$ polynomial. Observe that $h_i(b, \overline{s}^{\overline{d}}) = f_i(\sigma + e)$.

The main point of this is this is that if we set the polynomial
map \( G^{(1)}(x_1, \ldots, x_{l+1}) \) by letting each \( G_{i}^{(1)} = h_{i} \), and setting the private seed \( S = x^{\frac{n}{2}} \):

\[
G^{(1)}(P, S) = \text{PRG.Eval}(\sigma + e)
\]

The reason \( G^{(1)} \) is interesting because \( e \) is sparse. With probability, \( 1 - 2^{-O(n)} \), it is oblivious of the location of \( \ell \) locations. As a consequence of this, for any given \( i \in [m] \), \( f_i(s) = f(s + e) \) with all but \( O(\ell^{-d}) \) probability as \( f_i \) is a local function depending on \( d \) randomly chosen inputs. Since in the hypergraph \( H \) of the Goldreich PRG, each set \( Q \), is chosen independently, every output is independently error-prone with probability \( O(\ell^{-d}) \). Because of this due to Chernoff style concentration bounds, out of \( m \) outputs, with probability \( 1 - 2^{-O(n)} \), all but \( T = O(m\ell^{-d}) \) outputs are error-prone.

This gives us a nice candidate for sPRG that satisfies almost all properties! The dimension of \( S \) and \( P \) is \( O(n) \) which is sublinear in \( m \), and it can be computed by a degree \((d, 2d)\) polynomial \( G^{(1)} \). We would be done if we could somehow force the output to be correct on all the \( m \) coordinates. For the rest of the overview, we refer to the indices \( i \in [m] \), such that \( f_i(s) = f_i(s + e) \), as bad indices/outputs.

To correct errors, we further modify the polynomial and include more pre-processed information in the private seeds. We describe a sequence of ideas that lead to the final correction method, starting with two wrong ideas that illustrate the difficulties we will overcome.

- The first wrong idea is correcting by adding the difference \( \text{Corr} = y - y'e \) between the correct and erroneous outputs, \( y = \text{Eval}(s) \) and \( y'e = \text{Eval}(s + e) \); we refer to \( \text{Corr} \) as the correction vector. To obtain the correct output, evaluation can compute the polynomial map \( G^{(1)}(b, \langle x^{\frac{n}{2}} \rangle) + \text{Corr} \). The problem is that \( \text{Corr} \) must be included in the seed, but it is as long as the output and would destroy expansion. Thus, we have to make use of the fact that \( \text{Corr} \) is sparse.
- To fix expansion, the second wrong idea is adding correction only for bad outputs, so that the seed only stores non-zero entries in \( \text{Corr} \). Recall that \( \text{Corr} \) is sparse with at most \( T \) non-zero elements. More precisely, the \( j \)'th output can be computed as \( G^{(1)}(b, \langle x^{\frac{n}{2}} \rangle) + \text{Corr}_j \) if output \( j \) is bad and without adding \( \text{Corr} \) otherwise. This fixes expansion, but now the evaluation polynomial depends on the location of bad outputs. If these locations are included in public information, this would leak information about the location of LPN noises, and jeopardize security. If on the other hand the locations are included in the private seed, then it is unclear how to maintain the requirement that the polynomial map computes only a degree two polynomial in the private seed.

These two wrong ideas illustrate the tension between the expansion and security of our sPRG. Our construction takes care of both, by compressing the correction vector \( \text{Corr} \) to be polynomially shorter than the output and stored in the seed, and expanding it back during evaluation in a way that is oblivious of the location of bad output bits. This is possible thanks to the sparsity of the correction vector and the allowed degree two computation on the private seed. We first illustrate our idea with the help of a simple case.

**Simple Case 1:** Much fewer than \( \sqrt{m} \) bad outputs. Suppose hypothetically that the number of bad outputs is bounded by \( z \) which is much smaller than \( \sqrt{m} \). Thus, if we convert \( \text{Corr} \) into a \( \sqrt{m} \times \sqrt{m} \) matrix, \( \text{Corr} \) has low-rank \( z \). We can then factorize \( \text{Corr} \) into two matrices \( U \) and \( V \) of dimensions \( \sqrt{m} \times z \) and \( z \times \sqrt{m} \), respectively, such that \( \text{Corr} = UV \), and compute the correct output as follows: \( \forall j \in [m] \),

\[
G^{(1)}(b, \langle x^{\frac{n}{2}} \rangle, U, V)) = G^{(1)}(b, \langle x^{\frac{n}{2}} \rangle) + (UV)_{k_j, l_j},
\]

where \( (k_j, l_j) \) is the corresponding index of the output bit \( j \), in the \( \sqrt{m} \times \sqrt{m} \) matrix. When \( z < \sqrt{m} \), the matrices \( U \) and \( V \) have \( 2z \sqrt{m} \) field elements, which are polynomially smaller than \( m = \ell \). As such, \( G^{(2)} \) is expanding. Moreover, observe that \( G^{(2)} \) has only degree 2 in the private seed and is completely oblivious to where the bad outputs are.

While the idea above works for fewer than \( \sqrt{m} \) bad outputs, it does not work for the case we are in. We have \( T = O(m\ell^{-d}) \) bad outputs. Nevertheless, we show that a similar idea works for this case.

**T bad outputs.** The above method however cannot handle more than \( \sqrt{m} \) bad outputs, whereas the actual number of bad outputs can be up to \( T = O(m\ell^{-d}) \), much larger than \( \sqrt{m} \) since \( d \) is an arbitrarily small constant. Consider another hypothetical case where the bad outputs are evenly spread in the following sense: suppose that if we divide the matrix \( \text{Corr} \) into \( m/\ell^d \) blocks, each of dimension \( \ell^d \times \ell^d \), there are at most \( \ell \) bad outputs in each block where \( r > 0 \) is a really small constant (say \( d/10 \)). In this case, we can “compress” each block of \( \text{Corr} \) separately using the idea from case 1. More specifically, for every block \( i \in [m/\ell^d] \), we factor it into \( UV_i \) with dimensions \( \ell^{2d} \times \ell^d \) and \( \ell^d \times \ell^d \), respectively, and correct bad outputs as follows: \( \forall j \in [m] \),

\[
G^{(1)}(b, \langle x^{\frac{n}{2}} \rangle, U_{ij}, V_{ij})) = G^{(1)}(b, \langle x^{\frac{n}{2}} \rangle) + (U_{ij} V_{ij})_{i, j},
\]

where \( i \) is the block that output \( j \) belongs to, and \( (k_{ij}, l_{ij}) \in [\ell^{2d}] \times [\ell^{2d}] \) is its index within this block. We observe that \( G^{(2)} \) is expanding, since each matrix \( U \) or \( V \) has \( \ell^{2d} \) field elements, and the total number of elements is \( \ell^{4d} \sqrt{m} > \frac{m}{d} \); which is polynomially smaller than \( m \) as long as \( d \) is positive and \( m \) is polynomially related to \( \ell \). Moreover, \( G^{(2)} \) is oblivious to the location of bad outputs just as in case 1.

This completely solves our problem except that we need to ensure that the bad outputs are well spread out in the manner described above. Our main observation here is that this is ensured due to the fact that in a Goldreich’s PRG candidate the input dependence hypergraph \( Q_{t_1}, ..., Q_{t_n} \) is randomly chosen. Therefore, once we fix the location of the non-zero errors inside \( e \) (where with high probability \( O(n\ell^{-d}) \))

---

\( ^e \) Any injective mapping from a vector to a matrix that is efficient to compute and invert will do.
locations are non-zero), in every block of \(\ell^2\) output bits, each entry \(j\) is independently non-zero with probability \(O(\ell^{-d})\). Thus, in expectation, each block has a constant number of bad output bits. More so, due to the Chernoff bound, it can be seen that with probability \(1 - 2^{-O(\ell)}\), each has at most \(\ell^4\) non-zero elements. Thus, our construction can be summarized as follows:

Step 1: Assign outputs. We partition the outputs into \(B\) buckets, via a mapping \(f_{\text{out}}: [m] \rightarrow [B]\). The number of buckets is set to \(B = m/\ell^d\) and the number of elements in each bucket is set to be \(\ell^d\) so that they exactly form a partition of \(m\). The mapping \(f_{\text{out}}\) simply divides \(m\) by \(B\) and outputs the remainder. Since the error \(\epsilon\) is chosen to be from the LPN error distribution and the hypergraph \(H\) of the PRG is randomly chosen, by a Chernoff-style argument, we can show that in each bucket out of \(\ell^d\) output bits, at most \(\ell\) of them are bad, except with probability \(1 - 2^{-\Omega(\ell^d)}\). We will set \(\ell = \ell\) for a tiny constant \(r > 0\).

Step 2: Compress the buckets. Next, we organize each bucket \(i\) into a matrix \(M_i\) of dimension \(\ell^d \times \ell^d\) and then compute its factorization \(M_i = U_i V_i\), where \(U_i, V_i\) are matrices of dimensions \(\ell^d \times t\) and \(t \times \ell^d\), respectively. To form matrix \(M_i\), we use another mapping \(f_{\text{ind}}: [m] \rightarrow [\ell^d]\) to assign each output bit \(j\) to an index \((k, l)\) in the matrix of the bucket \(i\) it is assigned to. This assignment must guarantee that no two output bits in the same bucket (assigned according to \(f_{\text{out}}\)) have the same index. One such way to compute \(f_{\text{ind}}(j)\) is to divide \(j \in [m]\) by \(B\). The remainder is set as \(f_{\text{ind}}(j)\), and the quotient is divided further by \(\ell^d\). The quotient and the remainder from this division are set as the resulting indices \((k, l)\). Once we have this, \((M_i)_{k,l}\) is set to \(\text{Corr}\) if there is \(j\) such that \(f_{\text{ind}}(j) = i\) and \(f_{\text{out}}(j) = (k, l)\). Since every matrix \(M_i\) has at most \(t\) non-zero entries, we can factor them and compute the correct output as:

\[
G^{(2)}(b, (\tilde{x}^{\text{out}}[j]), (U_i V_i)_{k,l}[j]) = \sum_{j} G^{(1)}(b, (\tilde{x}^{\text{out}}[j]), (U_i V_i)_{k,l}[j])
\]

\(G^{(2)}\) is expanding, because the number of field elements in \(U_i\)'s and \(V_i\)'s are much smaller than \(m\), namely: \(2^{\ell^d}B = O(m^{\ell^d+\epsilon}) \ll m\). We set \(I = (I, f_{\text{out}}, f_{\text{ind}})\).

Step 3: Zerize if uneven buckets. Finally, to deal with the low probability event that some bucket contains more than \(t\) bad outputs, we introduce a new variable called flag. If this occurs, our SPRG sets \(P\) and \(S\) as all zero vectors. In this case, the evaluation always outputs 0. This gives us our candidate SPRG. For security, observe that the polynomial map \(G^{(2)}\) is independent of the location of LPN noises. With probability \(1 - 2^{-\Omega(\ell^d)}\), the evaluation results in output \(y\). Therefore, by the LPN over \(Z_p\) assumption, the seed \(s\) of PRG is hidden and the security of PRG ensures that the output is pseudorandom when it is not all zero (which occurs with a subexponentially small probability). We now proceed to the formal construction and proof.

This completes the overview of our construction of structured seed PRG, as well as \(I\); see Jain et al., for the formal constructions and security proofs. In conclusion, \(I\) is an extremely powerful cryptographic tool and our works have established its feasibility based on well-studied assumptions.

References

1. WhitDeContest21—CHES 2021 Challenge. 2021. https://test2021whibox.io.
2. Alekhnovich, M. More on average case vs approximation complexity. In 44th FOCS (Oct. 2003). IEEE Computer Society Press, New York, 298–307.
3. Ananth, P., Jain, A. Indistinguishability obfuscation from compact functional encryption. In CRYPTO 2015, Part I. J. R. Gennaro and M.J. Robshaw, eds. Volume 9215 of LNCS (Aug. 2015). Springer, Heidelberg, 308–326.
4. Ananth, P., Sahai, A. Projective arithmetic-functional encryption and indistinguishability obfuscation from degree-5 multilinear maps. In EUROCRYPT 2017. Part I. J.-S. Coron and J.B. Nielsen, eds. Volume 10201 of LNCS (Apr-May 2017). Springer, Heidelberg, 152–181.
5. Applebaum, B., Avron, J., Brzuska, C. Arithmetic cryptography: Extended abstract. In ITCS 2015. T. Roughgarden, ed. (Jan. 2015). ACM, New York, 143–151.
6. Applebaum, B., Ishai, Y., Kushilevitz, E. Cryptography in NC1. In FOCS (2004). IEEE Computer Society, New York, 168–175.
7. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., et al. On the (im)possibility of obfuscating programs. In CRYPTO 2002. J. Kilian, ed. Volume 2139 of LNCS (Aug. 2001). Springer, Heidelberg, 1–18.
8. Bitansky, N., Vakilianathan, V. Indistinguishability obfuscation from functional encryption. In 56th FOCS. V. Guruswami, ed. (Oct. 2015). IEEE Computer Society Press, New York, 171–190.
9. Blum, A., Furst, M.L., Kearns, M., Lipton, R.J. Cryptographic primitives based on hard learning problems. In CRYPTO’93. D.R. Stinson, ed. Volume 773 of LNCS (Aug. 1994). Springer, Heidelberg, 278–291.
10. Boneh, D., Boyen, X., Shacham, H. Short group signatures. In CRYPTO 2004. M. Franklin, ed. Volume 3152 of LNCS (Aug 2004). Springer, Heidelberg, 41–55.
11. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y. Compressing vector OLE. In ACM CCS 2018. D. Lee, M. Mansanet, M. Backes and X. Wang, eds. (Oct. 2018). ACM Press, New York, 896–912.
12. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., et al. Correlated pseudorandom functions from variable-density LPN. In 61st FOCS (Nov. 2020). IEEE Computer Society Press, New York, 1098–1108.
13. Diffe, W., Hellman, M.E. New directions in cryptography. IEEE Trans. Inform. Theory 22, 6 (1976), 644–654.