Obfuscating Branching Programs Using Black-Box Pseudo-Free Groups[*]

Ran Canetti[†]
Boston U. and Tel Aviv U.

Vinod Vaikuntanathan[‡]
Toronto and MIT

August 14, 2013

Abstract

We show that the class of polynomial-size branching programs can be obfuscated according to a virtual black-box notion akin to that of Barak et.al., in an idealized black-box group model over pseudo-free groups. This class is known to lie between NC^1 and P and includes most interesting cryptographic algorithms. The construction is rather simple and is based on Kilian's randomization technique for Barrington's branching programs.

The black-box group model over pseudo-free groups is a strong idealization. In particular, in a pseudo-free group, the group operation can be efficiently performed, while finding surprising relations between group elements is intractable. A black-box representation of the group provides an ideal interface which permits prescribed group operations, and nothing else. Still, the algebraic structure and security requirements appear natural and potentially realizable. They are also unrelated to the specific function to be obfuscated.

Our modeling should be compared with the recent breakthrough obfuscation scheme of Garg et al. [FOCS 2013]: While the high level structure is similar, some important details differ. It should be stressed however that, unlike Garg et al., we do not provide a candidate concrete instantiation of our abstract structure.

1 Introduction

A program obfuscator $\mathcal{O}$, informally, is a compiler that changes a program P into a program $\mathcal{O}(P)$ which is equivalent to P in functionality, but leaks no more information about P than black-box access to P.

In spite of the immense popularity of program obfuscation in the programming and systems security communities, coming up with program obfuscation mechanisms that provide rigorous security guarantees has proven to be a hard problem. Indeed, such obfuscation schemes have been shown only for relatively few classes of programs, or functions, e.g. [Can97, LPS04, Wee05, HMLS07,

[*]A preliminary version of this work was submitted to Crypto 2007 and is available at [CV07].

[†]Supported by an NSF Algorithmic Foundations grant, an ISF grant, and the Check Point Institute for Information Security.

[‡]Supported by an NSERC Discovery Grant, DARPA award No. FA8750-11-2-0225, Connaught New Researcher Award and an Alfred P. Sloan Fellowship.

HRSV07, CD08, CRV10, BR13]. Furthermore, sweeping impossibility results show that a natural notion of security for program obfuscation, namely virtual black box (VBB) obfuscation, is impossible to obtain in general [BGI$^+$01, GK05].

Still, in an exciting new work, Garg et al. [GGH$^+$13] propose a general obfuscation mechanism and argue that it satisfies a weaker notion of security, called Indistinguishability Obfuscation [BGI$^+$01, GR07]. They also demonstrate that this notion actually suffices in many applications.

We take a slightly different approach and show general feasibility results for VBB program obfuscation in an idealized model of computation. Specifically, we show that in the "black-box group model", taken over a group that's pseudo-free except for a certain structure, a large class of branching programs can be VBB-obfuscated. While our obfuscation procedure is very simple and relies strongly on idealized security properties of the underlying group, the basic ingredients of our scheme show great similarity to corresponding ingredients in the [GGH$^+$13] scheme. (See the end of the introduction for a brief account of the history of this work.) We give background for and brief description of our scheme.

Pseudo-free Black-box Groups. The notion of a black-box group was introduced by Babai and Szemeredi [BS84]. Informally, a black-box group is a algebraic group adjoined with a "random representation of group elements" — namely a random function from group elements to a large enough binary domain, along with oracles that allow performing the group operation directly on the representations of the elements. The idea is to capture groups where the only meaningful computational operations that can be done with the representations of group elements is use them to do one of the group operations, namely group multiplication or inversion.

A free group is an infinite group defined by a set of generators $A = \{a_1, a_2, \ldots, a_n\}$. The elements of this group are all the words that use the symbols in A, along with their inverses (the inverses are also treated as formal symbols). This group is "free" in the sense that there is no relation among the a_i's that define the generating set. Pseudo-free groups, defined by Hohenberger [Hoh03] and later made precise by Rivest [Riv04] are, informally, groups which cannot be distinguished from a free group by any probabilistic polynomial-time adversary. The notion is naturally extendable to groups where some given set of relations are assumed to exist and the group remains otherwise free. (An example for such a group is the free *Abelian* group.) A Pseudo-Free Black-Box Group is a black-box group defined over a group that's assumed to be pseudo-free. Such a black-box group provides only very limited interface; for instance, it does not allow obtaining the representations of inverses, and it is infeasible to determine the order of the group. For formal definitions of these notions, see Section 2.

Group-based Branching Programs. Branching Programs are a non-uniform model of computation introduced by Lee [Lee59] and studied by [BDFP86, Bar86, BT87, BT88]. In particular, bounded-width branching programs (which can be considered as programs over some symmetric group S_n) were introduced by [BDFP86]. The celebrated result of Barrington [Bar86] shows that width-5 branching programs (equivalently, branching programs over the symmetric group S_5) are equal in computational power to the circuit class NC^1. The model of branching programs was further generalized to branching programs over arbitrary groups (as opposed to just the symmetric groups) and other algebraic structures starting from the work of [BT87, BT88]. As for the computational power of this model, we know that if the group has a certain (easy to satisfy) non-Abelian structure, branching programs over such a group are at least as powerful as NC^1 (using an easy generalization of the arguments used by Barrington). On the other hand, polynomial-size branching programs are contained in P.

Our construction. We show how to obfuscate any family of NC^1 circuits in a black-box group based on a group that is pseudo-free except for some simple structure. Specifically, we assume existence of a group G that is pseudo-free except for the following structure. To clearly present the different elements of the construction, we describe G as a product of four groups $G = G_A \times G_P \times G_K \times G_B$, where the group operation for G is the natural extension of the group operations of the four components. The group G_A is a pseudo-free Abelian group, the groups, G_P and G_K are non-Abelian pseudo-free group, and G_B is a group that supports the Barrington transformation (e.g., G_B can be S_5). We use an abstract black box access mechanism to G where the following idealized interfaces are given: (1) It is possible to obtain representations of group elements of the form $(r_A, 1, 1, 1)$, $(1, r_P, 1, 1)$, $(1, 1, r_K, 1)$ and $(1, 1, 1, r_B)$, together with their inverses, where r_A, r_P, r_K, r_B are random elements of G_A, G_P, G_K, G_B, respectively. (2) It is possible to perform the group operation in G on representations of elements in G. (3) It is possible to test whether a given string is a representation of an element of G that has a given form. (Specifically, we test whether the underlying element is of the form $(1, a_P, r_K, r_B)$, where a_P is any element of G_P and r_K, r_B are random elements of G_K and G_B that were fixed ahead of time.) (4) Finally, it is possible to obtain representations of six fixed "special elements. These are the elements if the form $(1, 1, 1, g_1)...(1, 1, 1, g_6)$, where $g_1..g_6$ are the six special elements used in the Barrington construction and for which the special Barrington relations hold.

In this group, our obfuscation mechanism can be briefly summarized as follows. Our starting point is the universal function $U(\langle c \rangle, x) = c(x)$. That is, U interprets its first input as a description of a circuit c and then runs c on the second input, x. We assume that U is in NC^1, which means that it only handles appropriately constrained circuits. We first run Barringtons transformation on U in our generic group, where we use only the G_B part of G, and the identity elements in the other parts. Recall that the output of this transformation is a sequence of triples $\{(\mathsf{var}(j), \mathbf{g}_j^0, \mathbf{g}_j^1)\}_{j=1..m}$ where each $\mathbf{g}_j^i = (1, 1, 1, g_j^i)$, each $\mathsf{var}(j)$ specifies an index i of a bit z_i in the input z of U, and the program is evaluated by picking from each triplet the element $\mathbf{g}_j^{z_{\mathsf{var}(j)}}$ that corresponds to the value of that bit, multiplying all the elements in order, and checking whether the end product equals a special group element. (Here m is the length of the program.) Now, consider the following potential "obfuscated program": For each input bit location $\mathsf{var}(j)$ that corresponds to the description of c, output only the pair $(\mathsf{var}(j), \mathbf{g}_j^{z_{\mathsf{var}(j)}})$. For each input bit location $\mathsf{var}(j)$ that corresponds to the input x, output the entire triple $\{(\mathsf{var}(j), \mathbf{g}_j^0, \mathbf{g}_j^1)\}$. This information certainly suffices for evaluating c on any input x of the evaluators choice. However, it is still far form a valid obfuscation since it does not "hide" c at all.

To protect c, we randomize the output of Barrington's transformation in three steps. Here we use the other three components of G. The first step is similar to Kilian's randomization [K89], and uses the G_K component: For each $j = 1..m-1$ choose a random element r_j in G_K. and replace each triple $(\mathsf{var}(j), \mathbf{g}_j^0, \mathbf{g}_j^1)$ with $(\mathsf{var}(j), (1, 1, r_j, g_j^0), (1, 1, r_j, g_j^1))$. In addition, the special target group element α from Barringtons construction is replaced by $(1, 1, r, \alpha)$, where $r = r_1 r_2 ... r_m$. The evaluator is given the ability to test whether it holds a representation of an element of the form $(1, a_P, r, \alpha)$ where a_P is any element in G_P. Intuitively, this step ensures that the only way to obtain meaningful information from the program is by multiplying one element from each triple, in order. Note that the pseudofree group structure allows us to avoid the need to use inverses in the Kilian randomizers.

The second step is to randomize the G_P coordinates of the group elements. That is, we multiply each one of the $2m$ group elements in the branching program resulting from the previous step by a new, unique element of the form $(1, r_P, 1, 1)$ where r_P is a random element in G_P. This step ensures that *partial* evaluations of the obfuscated program look like independent random values.

The third step is to randomize the G_A coordinates of the group elements so as to force the evaluator of the program to use each bit of the input x in a consistent way across all the triples. That is, for each $i = 1..|x|$ let $j_1, ..., j_k$ be the triples that correspond to the ith bit in x. Then, for each bit $b \in \{0, 1\}$, the obfuscator chooses random $s_1, ..., s_k$ in G_A such that $s_1 \cdot ... \cdot s_k = 1$, and for each $l = 1...k$ sets the G_A component of $g^b_{j_l}$ to be s_l. We show:

Main Theorem 1 (Informal) *The above construction is a VBB obfuscator for any family of NC^1 circuits, in the black box group model for G.*

Interpreting Our Result. We suggest some interpretations of our result. Perhaps the most immediate and natural interpretation is that our abstract construction provides a "blueprint", or a direction for coming up with potential concrete obfuscation schemes. This interpretation should be compared with the breakthrough construction of Gentry et al. [GGH+13]. Indeed, while at high level their scheme appears to have similar structure, the details are a somewhat different. In particular, they use in a crucial way the fact that their underlying mathematical structure allows even adversarial parties to evaluate only a bounded number of group operations. In contrast, our scheme works even when an unbounded number of group operations can be performed.

Alternatively, the scheme can be taken as a basis for hardware-based obfuscation, where the black box group operations are implemented in hardware.

Either way, it is worth stressing that the ideal structure we assume has relatively simple interface and is independent of the functions to be obfuscated.

Finally, we note that our construction bypasses the impossibility result of [BGI+01]. This is due to the fact that we use an idealized model, and thus our obfuscated program is not a fully specified algorithms, and so the [BGI+01] "diagonalization" technique does not apply. Still, any concrete instantiation of our obfuscation using a real pseudo-free group cannot satisfy the virtual black-box property in the standard model. Finding a meaningful notion of security for program obfuscation that captures the full power of such general construction without being susceptible to diagonalization-based impossibility results such as those of [BGI+01] is an interesting question.

History of this work. An early version of this work was submitted to Crypto 2007. Shortly afterwards, we found that the construction there was lacking. We then formulated the solutions of the additional Abelian and non-Abelian randomizations, but these solutions were never formalized. Given the recent exciting advancements in program obfuscation mechanisms [GGH+13], we have decided to complete writing up this early work and post it in hope of obtaining better understanding of the necessary ingredients towards general obfuscation. In particular, the G_A and G_P parts of the current solution are direct formalizations of our ideas from 2007. We are grateful to the authors of [GGH+13] for prompting us to dig up our old ideas and make them rigorous. For completeness, the 2007 version of this work is available at [CV07].

2 Mathematical Preliminaries

Computing with an abstract group family $G = \{G_n\}_{n \in \mathbb{N}}$ requires representing the group elements as strings, and performing group operations with such a representation. We will work with two types of representations of group elements.

Canonical Representation: Let $\ell = \ell(n)$ be a polynomial function. A canonical representation is simply an injective function $\Psi_n : G_n \to \{0,1\}^{\ell(n)}$. For any group element $g \in G_n$, $\Psi(g) \in \{0,1\}^n$ is called the canonical representation of g.

When we say statements of the form "choose a group $G \ldots$" in a computational context, what we really mean is "choose a group G together with a canonical representation $\Psi \ldots$". When we talk about computing on group elements $g \in G$, what we mean is computing on $\Psi(g)$. For example, a canonical representation of $a \in \mathbb{Z}_N^*$ is just its bit representation using a $\lceil \log N \rceil$-bit string.

The more important object for us is a:

Black-box Representation: Let $r = r(n)$ and $\ell = \ell(n)$ be polynomial functions. The black-box representation of a group G_n, denoted $[G_n]$, is a *random injective function* $R_n : G_n \to \{0,1\}^{\ell(n)}$ chosen from the space of all such functions, together with algorithms P, Samp and Test, defined as below. (When clear from the context, we omit the subscript n.)

- The randomized function $\mathsf{Samp} = \mathsf{Samp}_{R,D}$ is parameterized by a distribution D over G. It picks an element $g \in G$ according to distribution D and returns the pair (x, y) where $x \leftarrow R(g)$ and $y \leftarrow R(g^{-1})$.
- The function $\mathsf{P} = \mathsf{P}_R$ takes as input $x, y \in \{0,1\}^{\ell(n)}$, and outputs the representation of the product of the group elements associated to x and y. That is:
  - if there are group elements $f, g \in G$ such that $x = R(f)$ and $y = R(g)$, then output $z = R(f \cdot g)$.
  - output $\perp$ otherwise.
- The function $\mathsf{Test}_{L,R}$ is a procedure parameterized by a binary relation L on elements in G. For strings $x, y \in \{0,1\}^{\ell(n)}$, $\mathsf{Test}_{L,R}(x, y) = 1$ if x and y represent group elements that stand in the relation L. That is, if there are group elements $f, g \in G$ such that $x = R(f)$ and $y = R(g)$, and $L(f, g) = 1$, then output 1. Else output 0.

An algorithm $\mathcal{A}$ that works with black-box access to a group G is denoted $\mathcal{A}^{[G]}(\cdots)$. $\mathcal{A}^{[G]}(\cdots)$ has black-box access to the procedures R, P, Samp and Test.

Free and Pseudo-Free Groups. A *free group* is an infinite group generated by a given set of generators which have no non-trivial relationships. More precisely, let $A = \{a_1, a_2, \ldots, a_k\}$ be a non-empty set of distinct symbols, called the *generators* of the free group. Elements of a free group are words over the alphabet A. The empty word, denoted ϵ, is the identity of the free group. We let $F(A)$ denote the free group generated by A as a set of generators. Words can be simplified by grouping together adjacent symbols that are the same. For example, the word $a_1 a_2 a_2 a_3 a_2$ can be simplified to $a_1 a_2^2 a_3 a_2$ (but not to $a_1 a_2^3 a_3$, since the free group is not Abelian). Thus, $F(A)$ is the set of all words over A in canonical form. The multiplication operation for a free group is just concatenating the two words, and reducing it to canonical form. A free group has no surprising identities – the only ones are those that follow from the axioms of group theory.

Informally, a group G is called *pseudo-free* if it is "indistinguishable" from a free group (in a sense that we will make precise below). The notion of pseudo-free groups was first introduced by Hohenberger [Hoh03], and later made precise by Rivest [Riv04].

To define the notion of pseudo-freeness precisely, we introduce the notion of equations over a free group. Let $x_1, x_2, \ldots, x_m$ be variables that can take on values in $F(A)$. An equation in $F(A)$ takes the form $w_1 = w_2$, where w_1 and w_2 are words over the symbols of $F(A)$ and the set of variables $X = \{x_1, x_2, \ldots, x_m\}$. That is, $w_1, w_2 \in (X \cup A)^*$. We will denote such an equation by $\mathsf{E}(x_1, \ldots, x_m; a_1, \ldots, a_k)$. Equations that have solutions in the free group are called *satisfiable*, and ones that are not are called *unsatisfiable*. For example, the equation $x_1 a_2^3 = a_2 x_2^2$ is satisfiable by setting $x_1 = x_2 = a_2^2$, whereas the equation $x_1 a_1 = a_2 x_1$ is unsatisfiable. Letting e denote the identity of the free group, namely the empty word, the equation $x_1 a_2 = e$ is unsatisfiable as well.

Consider now an equation over variables X and a random set of elements $\Gamma = \{g_1, \ldots, g_k\}$ from the group G, obtained by replacing the constants $a_1, \ldots, a_k$ in the equation $E(x_1, \ldots, x_m; a_1, \ldots, a_k)$ by $g_1, \ldots, g_k$. We call this $\mathsf{realE}(x_1, \ldots, x_m; g_1, \ldots, g_k)$. We say that a group G is pseudo-free if no probabilistic polynomial time adversary can, given randomly chosen $g_1, \ldots, g_k \leftarrow G$, produce an equation $E = \mathsf{E}(x_1, \ldots, x_m; a_1, \ldots, a_k)$ that is unsatisfiable in the free group generated by A, together with a solution for the corresponding equation $\mathsf{realE}(x_1, \ldots, x_m; g_1, \ldots, g_k)$ in the group G.

Definition 1 (Pseudo-Free Group [Hoh03, Riv04]) *A family $G = \{G_n\}_{n \in \mathbb{N}}$ of finite computational groups is pseudo-free if for every probabilistic polynomial-time adversary A, for every polynomial p, if $g_1, g_2, \ldots, g_{p(n)}$ are elements chosen uniformly and independently at random from G_n, then the probability*

$$\begin{aligned}\Pr\big[&A(g_1, g_2, \ldots, g_{p(n)}) = (\mathsf{E}, h_1, h_2, \ldots, h_m) : \ h_i \in G_n,\\ &\mathsf{E} = \mathsf{E}(x_1, x_2, \ldots, x_m; a_1, a_2, \ldots, a_{p(n)}) \text{ is unsatisfiable over the free group } F(a_1, \ldots, a_{p(n)}), \text{ and}\\ &\mathsf{realE} = \mathsf{realE}(h_1, \ldots, h_m; g_1, g_2, \ldots, g_{p(n)}) \text{ is satisfied over } G_n\big]\end{aligned}$$

is a negligible function of n.

Remark. This definition differs slightly from that of [Hoh03, Riv04] in that we do not allow the words or the equations to use the inverses of the generators a_i. In particular, in the definition of [Hoh03, Riv04], the free group $F(A)$ is defined to be the set of strings $(A \cup A^{-1})^*$ where $A^{-1} = \{a_1^{-1}, \ldots, a_m^{-1}\}$ denotes the set of all the inverse symbols, whereas in our case the free group $F(A)$ is the set of all strings A^*. The case where inverses are provided explicitly is captured in our next definition, namely that of (free and) pseudo-free groups with relations.

Pseudo-Free Group with Relations. A free group with relations (FGR) is defined similarly, except that in addition, the generators are related in certain ways. In particular, let $A = \{a_1, a_2, \ldots, a_m\}$ be a non-empty set of generators as before. Let $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_\ell\}$ be a set of relations. Each $\mathcal{R}_i$ is an equation of the form $w_{i,0} = w_{i,1}$ where $w_{i,0}$ and $w_{i,1}$ are words over the alphabet A. Words can be simplified by grouping together adjacent symbols that are the same. They can also be simplified using the relations $\mathcal{R}$. For example, in a free group with alphabet

$\{a_1, a_2, a_3\}$ and the relation $a_2 a_3 = a_3 a_2^2$, the word $a_1 a_2 a_2 a_3 a_2$ can be simplified to $a_1 a_3 a_2^5$. This defines an equivalence class of words that can be transformed to each other using the group axioms and the relations. The free group with relations $F_{\mathcal{R}}(A)$ is the set of all equivalence classes of words over A. A free group with relations has no surprising identities – the only ones are those that follow from the axioms of group theory and (combinations of) the given relations.

We highlight two important special cases of free groups with relations:

1. The first example is a free Abelian group which is a free group generated by $A = \{a_1, \ldots, a_m\}$ with the $\binom{m}{2}$ relations $a_i a_j = a_j a_i$ for all $i, j \in [m]$.

2. The second example corresponds to the definition of a free group used in [Hoh03, Riv04] where the words and equations are allowed to work with inverses of the generators. This is captured as a free group in our sense generated by $A = \{a_1, \ldots, a_m, a'_1, \ldots, a'_m\}$ with the m relations $a_i a'_i = e$ for all $i \in [m]$ (where e, the empty word, is the identity of the free group.)

A pseudofree group with relations is defined similarly:

Definition 2 (Pseudo-Free Group with Relations) *A family $G = \{G_n\}_{n\in\mathbb{N}}$ of finite computational groups is pseudo-free with relations $\mathcal{R}$ if for every probabilistic polynomial-time adversary A, for every polynomial p, if $g_1, g_2, \ldots, g_{p(n)}$ are elements chosen uniformly and independently at random from G_n* subject to satisfying relations $\mathcal{R}$*, then the probability*

$$\begin{aligned}\Pr\big[&A(g_1, g_2, \ldots, g_{p(n)}) = (\mathsf{E}, h_1, h_2, \ldots, h_m) : \ h_i \in G_n,\\ &\mathsf{E} = \mathsf{E}(x_1, x_2, \ldots, x_m; a_1, a_2, \ldots, a_{p(n)}) \text{ is unsatisfiable over } F_{\mathcal{R}}(a_1, \ldots, a_{p(n)}), \text{ and}\\ &\mathsf{realE} = \mathsf{realE}(h_1, \ldots, h_m; g_1, g_2, \ldots, g_{p(n)}) \text{ is satisfied over } G_n\big]\end{aligned}$$

is a negligible function of n.

We stress that the real group elements $g_1, \ldots, g_{p(n)}$ that the adversary is given satisfy all the relations of the group. That is, for every $\mathcal{R}_i$, we have that the equation $\mathcal{R}_i(g_1, \ldots, g_{p(n)})$ is satisfied. In our first example of Abelian pseudo-free groups, this simply means that $g_1, \ldots, g_{p(n)}$ are uniformly random elements of the group, since we are guaranteed that every two elements in the real group commute. In our second example, the real group elements $g_1, \ldots, g_m, g'_1, \ldots, g'_m$ are chosen to be random elements in the group together with their inverses.

Recall that RSA group $\mathbb{Z}_N^*$ is a pseudo-free Abelian group:

Theorem 1 ([Mic05]) *Let $\mathcal{G} = \{\mathcal{G}_n\}$ denote an ensemble of groups where each*

$$\mathcal{G}_n = \{\mathbb{Z}_N^* : N = pq \text{ where } p \text{ and } q \text{ are } n\text{-bit primes}\}$$

Then the group ensemble $\mathcal{G}$ is a pseudo-free Abelian group family under the strong RSA assumption.

We also need the following easy fact:

Proposition 2 *Let $\mathcal{G} = \{\mathcal{G}_n\}$ and $\mathcal{H} = \{\mathcal{H}_n\}$ denote two families of pseudo-free groups with relations $\mathcal{R}_G$ and $\mathcal{R}_H$, respectively. Then their product $\mathcal{G}_n \times \mathcal{H}_n := \{G \times H : G \in \mathcal{G}_n, H \in \mathcal{H}_n\}$ is also pseudo-free with relations $\mathcal{R}_{G\times H}$ defined as follows: for every equation $R_G := (w_{1,G} = w_{2,G}) \in \mathcal{R}_G$ and $R_H := (w_{1,H} = w_{2,H}) \in \mathcal{R}_H$, we have that the equation $(w_{1,G}, w_{1,H}) = (w_{2,G}, w_{2,H}) \in \mathcal{R}_{G\times H}$.*

2.1 Branching Programs

Branching Programs are a non-uniform model of computation introduced by Lee [Lee59] and studied by [BDFP86, Bar86, BT87, BT88]. This notion was generalized to branching programs over groups, monoids and other algebraic structures starting from the work of [BT87, BT88]. We will use the definition of branching programs over groups, given below.

For a group G, a G-branching program is a family of programs $\{P_n\}_{n\in\mathbb{N}}$, where P_n operates on inputs of length n. Each P_n is a sequence of m instructions of the form $(\mathsf{var}(j), g_{j,0}, g_{j,1})$ where $g_{j,0}, g_{j,1} \in G$ and $\mathsf{var} : [m] \to [n]$ is a function where $\mathsf{var}(j)$ specifies which of the n variables is associated to the j^{th} instruction. For a group element $\alpha \in G$, we say that a program P_n α-computes a Boolean function f_n if for every $x \in \{0,1\}^n$:

$$\prod_{j=1}^{m} g_{j,x_{\mathsf{var}(j)}} = \begin{cases} \alpha & \text{if } f_n(x) = 1 \\ ID & \text{if } f_n(x) = 0 \end{cases}$$

where ID denotes the identity element of G.

In our presentation, it will be convenient to think of universal branching programs $\{BP_n\}_{n\in\mathbb{N}}$ that operate on $2n$ bits – n function bits that specify the function to be computed, in addition to n input bits. The universal program BP_n defines a family of functions $\{F_{\mathbf{b}}(\mathbf{c}) := BP_n(\mathbf{b}, \mathbf{c})\}_{\mathbf{b}\in\{0,1\}^n}$.

Barrington's Theorem for Group Branching Programs. The following theorem is essentially Barrington's Theorem, stated in the language of pseudo-free groups with relations:

Theorem 3 *Let fG be the free group generated by the set of symbols $(a_1, \ldots, a_k; \alpha, \beta, \gamma, p, q, r)$ with the inverses of $\alpha, \beta, \gamma, p, q$ and r, together with the following four relations:*

$$p\alpha^{-1}p^{-1} = \alpha, \qquad q\alpha q^{-1} = \beta, \qquad \alpha\beta\alpha^{-1}\beta^{-1} = \gamma, \qquad r\gamma r^{-1} = \alpha$$

Then, there is a fG-branching program of size $O(4^d)$ for any function that can be computed by binary AND-OR circuits of depth d.

For completeness, we sketch the proof below.

Let C be a Boolean circuit of depth d composed of NOT and two-input AND gates. We will construct a fG-branching program of size 4^d that computes the same function as C. The construction proceeds inductively.

1. For a variable x_i, the single instruction $(x_i, g_0 = 1, g_1 = \alpha)$ is a branching program that α-computes the function $f(x_1, \ldots, x_n) = x_i$. The single instruction $(x_1, g_0 = 1, g_1 = 1)$ computes the constant function that outputs 0, and $(x_1, g_0 = \alpha, g_1 = \alpha)$ computes the constant function that outputs 1.

2. Let P be a branching program that computes a circuit C. Then, the program that computes the function $NOT(C)$ is constructed as follows. Let (x_{I_0}, g_0^0, g_0^1) be the first instruction in P and let (x_{I_m}, g_m^0, g_m^1) be the last instruction. Then, to create the program P' for $NOT(C)$:
   - Change the first instruction to $(x_{I_0}, pg_0^0, pg_0^1)$; and
   - Change the last instruction to $(x_{I_m}, g_m^0\alpha^{-1}p^{-1}, g_m^1\alpha^{-1}p^{-1})$.

Whenever P evaluates to the identity element ID, P' will evaluate to $p\alpha^{-1}p^{-1} = \alpha$, and whenever P evaluates to α, P' will evaluate to $p\alpha\alpha^{-1}p^{-1} = ID$, as required.

3. Let P and Q be branching programs that compute circuits C and D. Then, the program that computes the circuit $AND(C, D)$ is constructed as follows. Let (x_{I_0}, g_0^0, g_0^1) – resp. (y_{I_0}, h_0^0, h_0^1) – be the first instruction in P – resp. Q – and let (x_{I_m}, g_m^0, g_m^1) – resp. (y_{I_m}, h_m^0, h_m^1) – be the last instruction in P – resp. Q. Then, to create the program for $AND(C, D)$, concatenate four programs C_1, D_1, C_2 and D_2, defined as follows:

   - To create C_1, change the first instruction of C to $(x_{I_0}, rg_0^0, rg_0^1)$;
   - To create D_1, change the first instruction of D to $(y_{I_0}, qh_0^0, qh_0^1)$; and the last instruction to $(y_{I_m}, h_m^0 q^{-1}, h_m^1 q^{-1})$;
   - To create C_2, change the first instruction of C to $(x_{I_0}, p^{-1}g_0^0, p^{-1}g_0^1)$ and the last instruction to $(x_{I_m}, g_m^0 p^{-1}, g_m^1 p^{-1})$; and
   - To create D_2, change the first instruction of D to $(y_{I_0}, q^{-1}h_0^0, q^{-1}h_0^1)$; and the last instruction to $(y_{I_m}, h_m^0 q r^{-1}, h_m^1 r^{-1})$;

   When either P or Q evaluate to the identity element ID, the new program will evaluate to ID, and and when both P and Q evaluate to α, P' will evaluate to $r\alpha\beta\alpha^{-1}\beta^{-1}r^{-1} = r\gamma r^{-1} = \alpha$, as required.

2.2 Obfuscation

Barak et al. [BGI+01] gave the first formal definition of a notion of obfuscation that captured the property that the obfuscator strip programs of non-black-box information. They formalized this by requiring that any predicate computable from the obfuscated program is also computable from black-box access to it.

Our variant of the definition will be in the programmable black-box group oracle model, where the adversary gets access to an oracle $[G] \leftarrow \mathsf{BB.Init}_G(1^n)$ for some group, but the simulator doesn't. Informally, this gives the simulator the ability to fake an oracle of his choice on-the-fly during the simulation. A formal definition, called *predicate obfuscation* follows.

For a family C of polynomial-size circuits, for a length parameter n let C_n be the circuits in C with input length n (i.e. $\mathsf{C} = \{\mathsf{C}_n\}$).

Definition 3 (Predicate Obfuscation [BGI+01]) *An efficient algorithm $\mathcal{O}$ is a* predicate obfuscator for the family $\mathsf{C} = \{\mathsf{C}_n\}$ *in the black-box group oracle model for a family of groups $\{G_n\}_{n\in\mathbb{N}}$ if it has the following properties:*

- *Preserving Functionality: There exists a negligible function $\mathsf{negl}(n)$, s.t. for all input lengths n, for any $C \in \mathsf{C}_n$:*

$$\Pr[\exists x \in \{0,1\}^n : (\mathcal{O}^{[G]}(C))^{[G]}(x) \neq C(x)] \leq \mathsf{negl}(n)$$

  The probability is taken over $\mathcal{O}$'s random coins and the choice of the oracle $[G] \leftarrow \mathsf{BB.Init}_G(1^n)$.

- *Polynomial Slowdown: There exists a polynomial $p(n)$ such that for sufficiently large input lengths n, for any $C \in \mathsf{C}_n$, the obfuscator $\mathcal{O}$ only enlarges C by a factor of p: $|\mathcal{O}^{[G]}(C)| \leq p(|C|)$.*

- *Virtual Black-box: For every polynomial sized adversary circuit A, there exists a polynomial size simulator circuit S and a negligible function $\mathsf{negl}(n)$, such that for every input length n, for every $C \in \mathsf{C}_n$, for every predicate π:*

$$\left|\Pr[A^{[G]}(\mathcal{O}^{[G]}(C)) = 1] - \Pr[S^C(1^n) = 1]\right| \leq \mathsf{negl}(n)$$

The probability is over the coins of the adversary, the simulator and the obfuscator as well as the choice of the black-box oracle $[G] \leftarrow \mathsf{BB.Init}_G(1^n)$.

3 Obfuscating Branching Programs using Pseudo-free Groups

In this section, we show a simple obfuscator for all functions computable in NC^1, in the idealized black-box group model for the product group $G := G_A \times G_K \times G_P \times G_B$ where:

- G_A is a pseudo-free Abelian group. For instance, one can use the family of groups $\{\mathbb{Z}_N^* : N = pq$ is a product of two n-bit primes p and $q\}$ which is pseudo-free Abelian by a result of Micciancio [Mic05];
- G_K and G_P are two copies of a pseudo-free non-Abelian group[1]; and
- G_B is a Barrington group which could, for instance, be the symmetric group S_5.

The testing relation L for Test is described later on. The sampling distributions D for Samp allow sampling elements that have 1 in three out of the four coordinates, and in the fourth coordinate a random element in the corresponding group. In addition, it is possible to obtain the description of the six special elements $(1, 1, 1, g_1), ..., (1, 1, 1, g_6)$ where $g_1, ..., g_6 \in G_B$ are used in the Barrington construction. From now on, we allow ourselves to say "choose a random element in G_B" (resp., G_A, G_P, G_K) and mean "choose an element of G that has a random element of G_B (resp., G_A, G_P, G_K) in the appropriate coordinate, and identity elements in the other three coordinates."

Outline of the Obfuscator Construction. We start with a universal NC^1 circuit U that takes as input $2n$ bits: n *function bits* $\mathbf{f} = f_1 \dots f_n$ that specify the circuit to be computed, and n *input bits* $\mathbf{x} = x_1 \dots x_n$. The circuit computes a family of functions on n bits given by $\mathcal{C} = \left\{C_{\mathbf{f}}(\mathbf{x}) = U(\mathbf{f}, \mathbf{x})\right\}_{\mathbf{f} \in \{0,1\}^n}$.

Our first step is to apply Barrington's theorem (Theorem 3) to the circuit $U(\mathbf{f}, \cdot)$ to produce a branching program over the group G_B in canonical form. That is, the branching program has $m = \mathrm{poly}(n)$ layers, numbered 1 through m, where in the i^{th} layer, the program looks at variable $i \pmod{2n}$. We assume that the first n variables are the function bits and the last n are the input bits. That is, the first variable is read in layers $1, 2n + 1, 4n + 1, \dots$, and so forth. We stress that this is a simplifying convention only: clearly, any branching program can be converted into one in this canonical form.

The obfuscator will henceforth work with this branching program which is composed of tuples

$$(\mathsf{var}(j), g_{j,0}, g_{j,1}) \qquad \text{for every } j \in [m].$$

[1] A possible candidate for such a group is $GL_2(\mathbb{Z}_N)$, the group of two-by-two matrices with elements in the ring $\mathbb{Z}_N$ where the group operation is matrix multiuplication.

where $\mathsf{var} : \{1, 2, \ldots, m\} \to \{1, 2, \ldots, n\}$ is a function that assigns a variable to every tuple in the branching program, and $g_{j,0}, g_{j,1} \in G_B$ are group elements. Below we view this as a branching program over the product group G where the constituent group elements are $\mathbf{g}_{j,0} := (1, 1, 1, g_{j,0})$ and $\mathbf{g}_{j,1} := (1, 1, 1, g_{j,1})$, respectively.

Our second step is to use Kilian's technique [K89] to randomize this branching program, with the goal of forcing the adversary to perform its evaluation "in the right order". Namely, we sample uniformly random elements $[\mathbf{u}_1], \ldots, [\mathbf{u}_m] \leftarrow G_K$ using the sampling interface $\mathsf{Samp}_{R,D}$ where the distribution D picks a uniformly random element in G_K. (More precisely, each $[\mathbf{u}_m]$ is in fact the element $(1, 1, u_m, 1)$.) We then randomize each tuple $(\mathsf{var}(j), [\mathbf{g}_{j,0}], [\mathbf{g}_{j,1}])$ into:

$$\big(\mathsf{var}(j), [\mathbf{u}_j] \cdot [\mathbf{g}_{j,0}], [\mathbf{u}_j] \cdot [\mathbf{g}_{j,1}]\big)$$

Note that if the original branching program $([\boldsymbol{\alpha}], ID)$-computes a function f, then the new randomized program $([\mathbf{u}^* \cdot \boldsymbol{\alpha}'], [\mathbf{u}^*])$-computes the same function, where $\mathbf{u}^* := \prod_{i=1}^{m} \mathbf{u}_i$.

The obfuscated program contains these tuples, and access to the black-box group oracle with a Test function parameterized by the relation

$$L\Big((f_a, f_k, f_p, f_b), (g_a, g_k, g_p, g_b)\Big) = 1 \text{ if and only if } g_a = f_a = 1, g_k = f_k = \mathbf{u}^* \text{ and } g_b = f_b = 1$$

Informally, the Kilian randomization makes sure that the only way for the adversary to deduce useful information from the obfuscated program is to multiply elements from consecutive layers, and do so in the prescribed order. However, this still does not provide sufficient protection, since an adversarial evaluator could still evaluate only parts of the program and compare partial results when run on different inputs (what we call the "partial evaluation attack"); furthermore, she could multiply elements from different layers in the branching program in a way thats inconsistent with any input string to the branching program (what we call the "inconsistent evaluation attack"). We thus use two additional randomization steps:

Our third step is to ensure security against the partial evaluation attack. We do this by completely randomizing the group elements in each tuple using the non-Abelian pseudo-free group G_P. That is, we randomize each tuple $(\mathsf{var}(j), [\mathbf{g}_{j,0}], [\mathbf{g}_{j,1}])$ into:

$$\big(\mathsf{var}(j), [\mathbf{v}_{j,0}] \cdot [\mathbf{g}_{j,0}], [\mathbf{v}_{j,1}] \cdot [\mathbf{g}_{j,1}]\big)$$

where the uniformly random elements $\mathbf{v}_{j,b} \leftarrow G_P$ are sampled using the Samp oracle. (More precisely, the sampled element is of the form $(1, 1, v_{j,b}, 1)$.)

All these still leave one more attack in the picture, namely the inconsistent evaluation attack. That is, for two tuples j and j' such that $\mathsf{var}(j) = \mathsf{var}(j')$, the adversary could choose $\mathbf{g}_{j,0}$ from the first tuple and $\mathbf{g}_{j',1}$ from the second tuple, when computing a group product. This will leak information to the adversary beyond what he could learn from black-box access to the function, and in some case, could even reveal the entire function to him.

To force consistency, we use an *Abelian randomization technique.* Namely,

- For each $j \in \{1, \ldots, m\}$ and $b \in \{0, 1\}$, choose uniformly random numbers $a_{j,b}$ subject to the condition that for each i, and each $b \in \{0, 1\}$,

$$\prod_{j:\mathsf{var}(j)=i} a_{j,b} = 1$$

- Randomize each tuple $(\mathsf{var}(j), [\mathbf{g}_{j,0}], [\mathbf{g}_{j,1}])$ into:

$$\big(\mathsf{var}(j), [\mathbf{a}_{j,0}] \cdot [\mathbf{g}_{j,0}], [\mathbf{a}_{j,1}] \cdot [\mathbf{g}_{j,1}]\big)$$

  where $\mathbf{a}_{j,b} := (a_{j,b}, 1, 1, 1)$.

The idea is that if the adversary evaluates the program consistently, then all the elements $a_{j,b}$ cancel out. On the other hand, the adversary cannot force these random Abelian components to cancel out on any invalid execution, since doing so would mean that he found surprising relations between the Abelian group elements, thus violating pseudofreeness of the Abelian group G_A.[2]

Both the pseudo-freeness of the groups as well as the black-box idealization are necessary for security. For example, even though $\mathbb{Z}_N^*$ is a pseudo-free Abelian group, this alone does not prevent an adversary from *adding* two numbers $x, y \in \mathbb{Z}_N^*$, an operation that is not legal in the multiplicative group interface.

Our final obfuscated branching program consists of the tuples $(j, [\mathbf{g}_{j,0}], [\mathbf{g}_{j,1}])$ together with the black-box group interface. For a formal description of the obfuscator, see Figure 1.

Theorem 4 *Assume that $H = \{H_n\}_{n\in\mathbb{N}}$ is a (non-Abelian) pseudo-free family of groups, and assume that the Strong RSA assumption holds. Then, there exists an obfuscator $\mathcal{O}$ for the class of NC^1 circuits, secure under the virtual black-box definition, in the black-box group model for the product group $G := G_A \times G_P \times G_K \times G_B$ where G_A is an Abelian pseudo-free group, G_K and G_P are isomorphic copies of the non-Abelian pseudofree group H, and G_B is the symmetric group S_5.*

Proof: Define a layer $j \in \{1, \ldots, m\}$ of the branching program to be a *function layer* if $\mathsf{var}(j) \in \{1, 2, \ldots, n\}$ is a variable that describes the function being obfuscated. Let S denote the set of (secret) function layers. Define a layer $j \in \{1, \ldots, m\}$ of the branching program to be an *input layer* if $\mathsf{var}(j) \in \{n + 1, n + 2, \ldots, 2n\}$ is a variable that describes the input to the function. Let P denote the set of (public) input layers.

For a formal description of the obfuscator, see Figure 1 and for the computation of the obfuscated program, see Figure 2. Correctness is straightforward. To show that this is an obfuscation of the class of NC^1 circuits $\mathcal{C} := \{U(\mathbf{f}, \cdot)\}_{\mathbf{f}\in\{0,1\}^n}$,it remains to show that it satisfies the virtual black-box property.

Virtual Black-box. Let $\mathsf{A}^{[G]}$ be a PPT adversary. We now construct a simulator $\mathsf{Sim}^{U(\mathbf{f},\cdot)}$ which has oracle access to the functionality of $U(\mathbf{f}, \cdot)$, and simulates the view of the adversary given the obfuscated program $\mathcal{O}(U(\mathbf{f}, \cdot))$.

The simulator runs the adversary on input a fake obfuscated branching program $\widetilde{BP}$ that consists of uniformly random strings in $\{0, 1\}^{\ell(n)}$. Namely, a uniformly random string $[\sigma_j]$ for every function layer j, and a pair of uniformly random strings $([\sigma_{j,0}], [\sigma_{j,1}])$ for every input layer j.

During the course of the simulated execution, the simulator maintains a list of tuples where each tuple T is of the form $T := (\Lambda, y)$ where $y \in \{0, 1\}^{\ell(n)}$ and $\Lambda = (\lambda_1, \ldots, \lambda_\ell)$ is an ordered sequence of pointers where each λ_i is either: σ_j, $\sigma_{j,b}$, or a group element that the simulator generated as a result of a query to the Samp oracle (to be described below.) The semantics is that y is the product of the group elements represented in the list Λ.

[2] An alternative way to obtain such elements in G_A is to choose $k - 1$ random elements $a_1, ..., a_{k-1}$ in G_A along with their inverses $a_1^{-1}, ..., a_{k-1}^{-1}$. Now, set $a_k = \Pi_{i=1..k} a_i^{-1}$.

Obfuscator $\mathcal{O}^{[G]}$

Input: A string $\mathbf{f} = (f_1, \ldots, f_n)$, specifying the function $U(\mathbf{f}, \cdot)$ to be computed.

Oracle Access: The black-box group oracle for the product group $G := G_A \times G_P \times G_K \times G_B$, consisting of the oracles P, Samp and Test with parameters described below.

Operation of the Obfuscator $\mathcal{O}$:

1. Run Barrington's procedure on the universal NC^1 circuit to obtain a universal branching program $BP := \left\{(\mathsf{var}(j), [\mathbf{g}_{j,0}], [\mathbf{g}_{j,1}])\right\}_{j=1}^{m}$.
2. Call the oracle Samp_{D_K} to get group elements $[\mathbf{u}_1], \ldots, [\mathbf{u}_m] \in G$ where the distribution D_K samples uniformly random elements from the set $\{1\} \times \{1\} \times G_K \times \{1\}$.
3. Call the oracle Samp_{D_P} to get a group element $[\mathbf{v}_j] \in G$ for every function layer j, and a pair of group elements $[\mathbf{v}_{j,0}], [\mathbf{v}_{j,1}] \in G$ for every input layer j. Here, the distribution D_P samples uniformly random elements from the set $\{1\} \times \{1\} \times G_K \times \{1\}$.
4. For every input variable $i \in \{n+1, \ldots, 2n\}$, and for every j such that $\mathsf{var}(j) = i$, and $c \in \{0, 1\}$, use the oracle Samp_{D_A} to get group elements $[\mathbf{a}_{j,c}]$ such that

$$\prod_{j:\mathsf{var}(j)=i} [\mathbf{a}_{j,0}] = 1 \text{ and } \prod_{j:\mathsf{var}(j)=i} [\mathbf{a}_{j,1}] = 1$$

   Here, the distribution D_A samples uniformly random elements from the set $G_A \times \{1\} \times \{1\} \times \{1\}$. (This can be done in many ways, see the description in the outline for two alternatives.)

Now, randomize the branching program. That is, for every $1 \le j \le m$:

1. If $\mathsf{var}(j) \in S$, then set
$$[\mathbf{g}_j] := [\mathbf{g}_j] \cdot [\mathbf{u}_j] \cdot [\mathbf{v}_j]$$
2. If $\mathsf{var}(j) \in P$, output both $[\mathbf{g}_{i,0}]$ and $[\mathbf{g}_{i,1}]$ where
$$[\mathbf{g}_{j,c}] := [\mathbf{g}_{j,c}] \cdot [\mathbf{u}_j] \cdot [\mathbf{v}_{j,c}] \cdot [\mathbf{a}_{j,c}]$$

As the description of the branching program, output these group elements together with a Test oracle for the relation L defined by

$$L\Big((1, \mathbf{u}^*, -, 1), (g_a, g_k, g_p, g_b)\Big) = 1 \text{ if and only if } g_a = 1, g_k = \mathbf{u}^* \text{ and } g_b = 1$$

where $\mathbf{u}^* := \mathbf{u}_1\mathbf{u}_2 \ldots \mathbf{u}_m$.

Figure 1: The Obfuscator for NC^1 Circuits

Evaluating the Obfuscated Program $\mathcal{O}^{[G]}(U(\mathbf{f},\cdot))$

Input: An obfuscated branching program given by a group element $[\mathbf{g}_j]$ for every function layer j, and a pair of group elements $([\sigma_{j,0}],[\sigma_{j,1}])$ for every input layer j. An input $\mathbf{x}=(x_1,\ldots,x_n)$.

The Obfuscated Program is evaluated as follows.

- For every function layer $j \in \{1,\ldots,m\}$, let $\mathbf{g}_{j,0} := \mathbf{g}_{j,1} := \mathbf{g}_j$.
- Compute

$$\mathbf{g}_{\mathbf{x}} := \prod_{j=1}^{m} \mathbf{g}_{j,x_{\mathsf{var}(j)}}$$

  using the product oracle P.
- Call the oracle Test on input $\mathbf{g}_{\mathbf{x}}$. If the Test oracle returns 1, output 0; otherwise output 1.

Figure 2: Evaluating the Obfuscated Branching Program

The data structure $\mathcal{T}$ is initialized with the tuples $\big(([\sigma_j]),[\sigma_j]\big)$ for every function layer j, and tuples

$$\big(([\sigma_{j,0}]),[\sigma_{j,0}]\big) \text{ and } \big(([\sigma_{j,1}]),[\sigma_{j,1}]\big)$$

for every input layer j.

Before proceeding further, we need the notions of *equivalent tuples*, *invalid tuples* and *inconsistent tuples*. Two tuples Λ_1 and Λ_2 are equivalent if they are either: (a) identical, or (b) identical except that they differ by a single group element which is an output of Samp_{G_A}. Equivalence of tuples is the transitive closure of this condition.

Definition 4 (Valid and Invalid Tuples) *A valid tuple $T=(\Lambda,y)$ relative to the branching program $\widetilde{BP}$ is one where $\Lambda=(\lambda_1,\ldots,\lambda_m)$ where each $\lambda_i \in \{0,1\}^{\ell(n)}$ and*

1. *for every function layer j, $\lambda_j=[\sigma_j]$; and*
2. *for every input layer j, $\lambda_j \in \big\{[\sigma_{j,0}],[\sigma_{j,1}]\big\}$.*

A tuple that is not valid is called invalid.

That is, a valid tuple represents a product of exactly m elements, where the j^{th} element is chosen from the j^{th} tuple of the branching program $\widetilde{BP}$. Note, however, that a valid tuple need not correspond to the evaluation of $\widetilde{BP}$ on any single input $\mathbf{x}$.

Definition 5 (Consistent and Inconsistent Tuples) *A consistent tuple $T=(\Lambda,y)$ relative to the branching program $\widetilde{BP}$ and an input $\mathbf{x}\in\{0,1\}^n$ is one where $L=(\lambda_1,\ldots,\lambda_m)$ where each $\lambda_i \in \{0,1\}^{\ell(n)}$ and*

1. *for every function layer j, $\lambda_j=[\sigma_j]$; and*

2. for every input layer j, $\lambda_j = [\sigma_{j,x_{\mathsf{var}(j)}}]$.

A tuple that is valid but not consistent is called inconsistent.

Note that a consistent tuple is always valid, but not necessarily vice versa.

We now proceed with the description of the simulator. The simulator runs the adversary on the fake branching program $\widetilde{BP}$ and answers its queries to the black-box group oracle $[G]$, as follows:

1. *Answering the* A *queries to* P: When A issues a query of the form $(y_1, y_2) \in (\{0,1\}^{\ell(n)})^2$ to the black-box group multiplication oracle P, Sim does the following:
   - Check if there are tuples of the form (Λ_1, y_1) and (Λ_2, y_2) in his list. If there are no such tuples, then return $\perp$.
   - Check if there is a tuple (Λ, y) such that Λ is equivalent to $\Lambda_1||\Lambda_2$. If yes, return y.
   - Otherwise, return a uniformly random string y and add the tuple $\big((\Lambda_1||\Lambda_2), y\big)$ to the list.
2. *Answering the* A *queries to* Samp: When A issues a query to Samp, return a random string $y \in \{0,1\}^*$, and add the tuple $((y), y)$ to the list.
3. *Answering the* A *queries to* Test: When A issues a query $y \in \{0,1\}^*$ to Test, do the following:
   (a) Check if there is a tuple of the form (Λ, y) in the list. If not, output 0;
   (b) Check if Λ is a consistent list w.r.t. input $\mathbf{x}$. If not, output 0;
   (c) If both checks pass, query the oracle $U(\mathbf{f}, \cdot)$ on input $\mathbf{x}$, and output whatever the oracle returns.
4. Finally, output whatever A outputs.

We now show that the simulated execution of the adversary described above is computationally indistinguishable from the real execution, assuming that the groups G_K and G_P are pseudo-free non-Abelian and that that group G_A is pseudo-free Abelian. That is,

$$\left|\Pr[\mathsf{A}^{[G]}(\mathcal{O}^{[G]}(U(\mathbf{f}, \mathbf{x}))) = 1] - \Pr[\mathsf{Sim}^{[G], U(\mathbf{f}, \cdot)}(1^n) = 1]\right| \leq \mathsf{negl}(n)$$

We will show this by the following sequence of claims:

Claim 5 *Assume that the adversary makes two queries to the product oracle* P *with lists* $L_1 \neq L_2$ *such that the products evaluate to the same real group element. Then, there is a PPT algorithm* E *such that*

$$\mathsf{Adv}_{\mathsf{E},\mathsf{pf}(G_P)}(1^n) \geq \mathsf{Adv}_{\mathsf{A}}^{\mathcal{O},U(\mathbf{f},\cdot)}(1^n) - \mathsf{negl}(n)$$

Proof: Let the two multiplication queries along with their corresponding histories be $(L_1, y_1), (L_2, y_2)$ and $(L_1', y_1'), (L_2', y_2')$. Let S_1, S_2, S_1', S_2' denote the ordered sequence of branching program group elements encoded in L_1, L_2, L_1', L_2' respectively.

Assume that $L_1||L_2 \neq L_1'||L_2'$, and yet

$$R^{-1}(y_1) \cdot R^{-1}(y_2) = R^{-1}(y_1') \cdot R^{-1}(y_2')$$

Let σ_i (resp. σ'_i) be the projections of $R^{-1}(y_i)$ (resp. $R^{-1}(y'_i)$) on the G_P component. Thus, we have $\sigma_1 \cdot \sigma_2 = \sigma'_1 \cdot \sigma'_2$ and consequently,

$$\prod_{i \in S_1 || S_2} \sigma_i = \prod_{i \in S'_1 || S'_2} \sigma_i \qquad (1)$$

Yet, $S_1 || S_2 \neq S'_1 || S'_2$, violating pseudofreeness of G_P. $\square$

Claim 6 *Assume that the adversary makes an inconsistent query to the black-box group oracle. Then, there is an algorithm* B *such that*

$$\mathsf{Adv}_{\mathsf{B}}^{\mathsf{pf.ab}}(1^n) \geq \mathsf{Adv}_{\mathsf{A}}^{\mathcal{O},BP_{\mathbf{b}}}(1^n) - \mathsf{negl}(n)$$

Proof: Let $\ell := \lfloor m/2n \rfloor$. The adversary B gets as input a sequence of elements $b_0, b_2, \ldots, b_{\ell-1} \in G_A$ and $b_0, \ldots, b'_{\ell-1} \in G_A$ together with their inverses. It generates the Abelian randomization in the following way:

- Choose a random variable $i \in \{1, 2, \ldots, 2n\}$.
- Set the Abelian randomization elements for variable i as follows: choose $v_{2nj+i,0} = b_j b^{-1}_{j+1 \pmod{\ell}}$, and $v_{2nj+i,1} = b'_j (b'_{j+1 \pmod{\ell}})^{-1}$. Note that the numbers $v_{2nj+i,0}$ and $v_{2nj+i,1}$ are random subject to the condition that their product is 1. That is, $\prod_{j=0}^{\ell} v_{2nj+i,0} = 1$ and $\prod_{j=0}^{\ell} v_{2nj+i,1} = 1$.

For the remaining variables $i' \neq i$, set the Abelian randomization components to be 1. Any inconsistent evaluation on variable i that results in a test query evaluating to 1 gives us an equation that is unsatisfiable in the free Abelian group but is clearly satisfied in G_A. $\square$

Claim 7 *Assume that the adversary makes an invalid query to the black-box group oracle. Then, there is an algorithm* B *such that*

$$\mathsf{Adv}_{\mathsf{B},\mathsf{pf}(G_K)}(1^n) \geq \mathsf{Adv}_{\mathsf{A},\mathcal{O},U(\mathbf{b},\cdot)}(1^n) - \mathsf{negl}(n)$$

Proof: Let the invalid test query together with its corresponding history be (L, y). Let S denote the ordered sequence of branching program group elements encoded in L. Since this is an invalid query, we have

$$\prod_{i \in S} \sigma_i = \mathbf{g}_1 \mathbf{g}_2 \ldots \mathbf{g}_m$$

and $S \neq (1, 2, \ldots, m)$. This immediately gives us an attack against the pseudo-freeness of G_K. $\square$
$\square$

Acknowledgments. We thank Guy Rothblum for insightful remarks on our result.

References

[BR13] Zvika Brakerski and Guy N. Rothblum. Obfuscating conjunctions. In *Crypto*, 2013.

[BS84] László Babai and Endre Szemerédi. On the complexity of matrix group problems i. In *FOCS*, pages 229–240, 1984.

[Bar86] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc. In *STOC*, pages 1–5, 1986.

[BDFP86] Allan Borodin, Danny Dolev, Faith E. Fich, and Wolfgang J. Paul. Bounds for width two branching programs. *SIAM J. Comput.*, 15(2):549–560, 1986.

[BGI$^+$01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18, 2001.

[BT87] David A. Mix Barrington and Denis Thérien. Non-uniform automata over groups. In *ICALP*, pages 163–173, 1987.

[BT88] David A. Mix Barrington and Denis Thérien. Finite monoids and the fine structure of nc^1. *J. ACM*, 35(4):941–952, 1988.

[Can97] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *CRYPTO*, pages 455–469, 1997.

[CD08] Ran Canetti and Ronny Ramzi Dakdouk. Obfuscating point functions with multibit output. In *EUROCRYPT*, pages 489–508, 2008.

[CRV10] Ran Canetti, Guy N. Rothblum, and Mayank Varia. Obfuscation of hyperplane membership. In *TCC*, pages 72–89, 2010.

[CV07] Ran Canetti and Vinod Vaikuntanathan. Obfuscating Branching Programs. Manuscript, 2007. Available at http//cs.tau.ac.il/˜canetti/cv07.pdf

[DS05] Yevgeniy Dodis and Adam Smith. Correcting errors without leaking partial information. In *STOC*, pages 654–663, 2005.

[GGH$^+$13] Sanjam Garg and Craig Gentry and Shai Halevi and Amit Sahai and Mariana Raikova and Brent Waters. Candidate Indistinguishability Obfuscation and Functional Encryption for all circuits. In *FOCS*, 2013.

[GR07] , Shafi Goldwasser and Guy Rothblum. On Best-Possible Obfuscation. TCC, pages 194-213, 2007.

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

[HMLS07] Dennis Hofheinz, John Malone-Lee, and Martijn Stam. Obfuscation for cryptographic purposes. In *TCC*, 2007.

[HMRV04] Susan Hohenberger, David Molnar, Ronald Rivest, and Vinod Vaikuntanathan. One-way groups: Definitions and applications, 2004. manuscript.

[Hoh03] Susan Hohenberger. The cryptographic impact of groups with infeasible inversion, 2003. Master's Thesis, EECS Department, MIT.

[HRSV07] Susan Hohenberger, Guy Rothblum, Abhi Shelat, and Vinod Vaikuntanathan. Securely obfuscating re-encryption. In *TCC*, 2007.

[K89] J. Kilian Uses of Randomness in Algorithms and Protocols, Chapter 3, The ACM Distinghished Dissertation 1989, MIT press.

[Lee59] C. Y. Lee. Representation of switching functions by binary decision diagrams, 1959. Bell System Technical Journal 38.

[LPS04] Ben Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In *EUROCRYPT*, pages 20–39, 2004.

[Mic05] Daniele Micciancio. The RSA group is Pseudo-free. In *EUROCRYPT*, pages 387–403, 2005.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, pages 84–93, 2005.

[Riv04] Ronald L. Rivest. On the notion of pseudo-free groups. In *TCC*, pages 505–521, 2004.

[Wee05] Hoeteck Wee. On obfuscating point functions. In *STOC*, pages 523–532, 2005.