Cryptanalysis of Indistinguishability Obfuscations of Circuits over GGH13*

Daniel Apon[†1], Nico Döttling[‡2], Sanjam Garg[§2], and Pratyay Mukherjee[¶2]

[1] *University of Maryland, College Park, USA*
[2] *University of California, Berkeley, USA*

November 14, 2016

Abstract

Annihilation attacks, introduced in the work of Miles, Sahai, and Zhandry (CRYPTO 2016), are a class of polynomial-time attacks against several candidate indistinguishability obfuscation ($i\mathcal{O}$) schemes, built from Garg, Gentry, and Halevi (EUROCRYPT 2013) multilinear maps. In this work, we provide a *general* efficiently-testable property for two single-input branching programs, called *partial inequivalence,* which we show is sufficient for our variant of annihilation attacks on several obfuscation constructions based on GGH13 multilinear maps.

We give examples of pairs of natural NC^1 circuits, which – when processed via Barrington's Theorem – yield pairs of branching programs that are partially inequivalent. As a consequence we are also able to show examples of "bootstrapping circuits," used to obtain obfuscations for all circuits (given an obfuscator for NC^1 circuits), in certain settings also yield partially inequivalent branching programs. Prior to our work, no attacks on any obfuscation constructions for these settings were known.

*Research supported in part from a DARPA/ARL SAFEWARE award, AFOSR Award FA9550-15-1-0274, NSF CRII Award 1464397, and a research grant by the Okawa Foundation. The views expressed are those of the authors and do not reflect the official policy or position of the funding agencies.

[†]Work done while visiting University of California, Berkeley. E-mail: dapon@cs.umd.edu

[‡]E-mail: nico.doettling@gmail.com

[§]E-mail: sanjamg@berkeley.edu

[¶]E-mail: pratyay85@gmail.com

Contents

1 Introduction **3**
- 1.1 Our Contributions . . . 3
- 1.2 Partial Inequivalence and Using it for Annihilation Attacks . . . 4
- 1.3 What Programs are Partially Inequivalent? Attack on NC^1 circuits. . . . 7
- 1.4 Roadmap . . . 8

2 Notations and Preliminaries **8**
- 2.1 Notation . . . 8
- 2.2 Matrix Products . . . 9
- 2.3 Column Space of a Matrix . . . 10
- 2.4 Branching Programs . . . 12
- 2.5 Indistinguishability Obfuscation . . . 12

3 Attack Model for Investigating Annihilation Attacks **13**
- 3.1 Annihilation Attack Model . . . 13
- 3.2 Obfuscation in the Annihilation Attack Model . . . 14
- 3.3 Abstract Indistinguishability Obfuscation Security . . . 15

4 Partially Inequivalent Branching Programs **15**

5 Annihilation Attack for Partially Inequivalent Programs **17**

6 Extending the Abstract Attack to GGH13 Multilinear Maps **21**
- 6.1 The GGH13 Scheme: Background . . . 21
- 6.2 Translating the Abstract Attack to GGH13 . . . 22
- 6.3 Completing the Attack for Large Enough Circuits . . . 23

7 Example of Partially Inequivalent Circuits **23**
- 7.1 Simple Pairs of Circuits that are Partially Inequivalent . . . 23
- 7.2 Larger Pairs of Circuits that are Partially Inequivalent . . . 24
- 7.3 Universal Circuit Leading to Partially Inequivalent Branching Programs . . . 26

A Some details on our implementation **30**

1 Introduction

An obfuscator is a program compiler which hides all partial implementation details of a program. This is formalized via the notion of indistinguishability obfuscation [BGI+01]: we say an obfuscator $\mathcal{O}$ is an indistinguishability obfuscator if it holds for every pair C_0, C_1 of functionally equivalent circuits (i.e. computing the same function) that $\mathcal{O}(C_0)$ and $\mathcal{O}(C_1)$ are indistinguishable. A recent surge of results has highlighted the importance of this notion: virtually "any cryptographic task" can be achieved assuming indistinguishability obfuscation and one-way functions [SW14].

All known candidate constructions of indistinguishability obfuscation, e.g. [GGH+13b, BGK+14, AB15], are based on multilinear-maps [GGH13a, CLT13] and [GGH15][1], which have been the subjects of various attacks ([CHL+15, CGH+15a] [CFL+16, HJ16, CLLT16]). Recently Miles, Sahai, and Zhandry [MSZ16] introduced a new class of polynomial-time[2] attacks against several obfuscation constructions, such as [BR14, BGK+14, AGIS14, MSW14, PST14] and [BMSZ16a], when instantiated with the GGH13 multilinear maps. Prior attacks on GGH13 multilinear maps required explicit access to low-level encodings of zero [GGH13a] [HJ16], or a differently represented low-level encoding of zero, in the form of an encoded matrix with a zero eigenvalue [CGH+15b].

More specifically, Miles et al. [MSZ16] exhibit two simple *branching programs* that are functionally equivalent, yet their [BGK+14]-obfuscations (and similar constructions like [BMSZ16b, MSW14, AGIS14]) are efficiently distinguishable.[3] Additionally they show that their attacks extend to any two branching programs with those two simple programs (respectively) padded into them. However, the branching programs constructed in [MSZ16], in particular the *all-identity* branching program, do not appear in the wild. More specifically, obfuscation constructions for circuits first convert an NC^1 circuit into a branching program via Barrington's transformation that results in non-trivial branching programs, even if one starts with simple circuits. This brings us to the following open question:

Are obfuscations of branching programs resulting from Barrington's Transformation applied to NC^1 circuits susceptible to annihilation attacks?

1.1 Our Contributions

In this work, we are able to answer the above question affirmatively. In particular, our main contributions are:

- We first define a *general* and efficiently-testable property of two branching programs called **partial inequivalence** (discussed below) and demonstrate an annihilation attack against [BGK+14]-like-obfuscations of any two (large enough) branching programs that satisfy this property.
- Next, using implementation in Sage [S+16] (see Appendix A for details on the implementation) we give explicit examples of pairs of (functionally equivalent) natural NC^1 circuits, which

[1] The work of [AJN+16] might be seen as an exception to this: Assuming the (non-explicit) existence of indistinguishability obfuscation, they provide an explicit construction of an indistinguishability obfuscator.

[2] Several subexponential-time or quantum-polynomial-time [CDPR16, ABD16, CJL16] attacks on GGH13 multilinear maps also been considered. We do not consider these in this paper.

[3] To avoid repetitions, from now on we will refer to the obfuscation constructions of [BGK+14, BMSZ16b, MSW14, AGIS14] by [BGK+14]-like obfuscations.

when processed via Barrington's Theorem yield pairs of branching programs that are partially inequivalent – and thus, attackable.

- As a consequence of the above result, we are also able to show that the "bootstrapping circuit(s)" technique used to boost $i\mathcal{O}$ for NC^1 to $i\mathcal{O}$ for $\mathsf{P/poly}$, for a certain choice of the universal circuit, yield partially inequivalent branching programs in a similar manner – and are, thus, also attackable.

Given our work, the new *attack landscape against GGH13-based obfuscators* is depicted in Figure 1. We refer the reader to [AJN+16, Figure 13] for the state of the art on obfuscation constructions based on CLT13 and GGH15 multilinear maps.

Our general partial inequivalence condition is broad and seems to capture a wide range of natural single-input branching programs. However, we do need the program to be large enough.[4] Additionally, we require the program to output 0 on a large number of its inputs.

Finally, our new annihilation attacks are essentially based on linear system solvers and thus quite *systematic*. This is in contrast with the attacks of Miles et al. [MSZ16] which required an exhaustive search operation rendering it hard to extend their analysis for branching programs resulting from Barrington's Theorem. Therefore, at a conceptual level, our work enhances the understanding of the powers and the (potential) limits of annihilation attacks.

One limitation of our technique is that they do *not* extend to dual-input branching programs. We leave it as an interesting open question.

	Branching Programs	NC^1 Circuits (Barrington's)	NC^1-to-$\mathsf{P/poly}$ [GGH+13b, App14] [BGL+15, GIS+10]
[GGH+13b]	◯	◯	◯
[BR14, BGK+14, AGIS14] [PST14, MSW14, BMSZ16a]	✕	⊗	⊗
[GMM+16, DGG+16]	◯	◯	◯

Figure 1: **The Attack Landscape for GGH13-based Obfuscators.** In all cases, the multilinear map is [GGH13a]. ◯ means no attack is known. ✕ means a prior attack is known, and we present more general attacks for this setting. ⊗ means we give the first known attack in this setting.

1.2 Partial Inequivalence and Using it for Annihilation Attacks

Below, after providing some additional backgrounds on multilinear maps and known attacks, we provide a overview of our annihilation attacks.

Multilinear Maps: Abstractly. As a first approximation, one can say that a cryptographic multilinear map system encodes a value $a \in \mathbb{Z}_p$ (where p is a large prime) by using a *homomorphic* encryption scheme equipped with some additional structure. In other words, given encodings of

[4] Note that, for our implementation we consider circuits that are quite small, only depth 3, and the resulting Barrington programs are of length 64. However, using the implementation we then "boost" the attack to a much larger NC^1 circuits that suffice for the real-world attack (discussed in Sec. 6) to go through.

a and b, one can perform homomorphic computations by computing encodings of $a+b$ and $a \cdot b$. Additionally, each multilinear map encoding is associated with some *level* described by a value $i \in \{1 \dots \kappa\}$ for a fixed universe parameter κ. Encodings can be added only if they are at the same level: $\mathsf{Enc}_i(a) \oplus \mathsf{Enc}_i(b) \to \mathsf{Enc}_i(a+b)$. Encodings can be multiplied: $\mathsf{Enc}_i(a) \odot \mathsf{Enc}_j(b) \to \mathsf{Enc}_{i+j}(a \cdot b)$ if $i + j \leq \kappa$ but is meaningless otherwise. We naturally extend the encoding procedure and the homomorphic operations to encode and to compute on matrices, respectively, by encoding each term of the matrix separately. Finally, the multilinear map system comes equipped with a *zero test*: an efficient procedure for testing whether the input is an encoding of 0 at level-κ. However, such zero-test procedure is not perfect as desired when when instantiated with concrete candidate multilinear maps. In particular we are interested in the imperfection in GGH13 map.

An Imperfection of the GGH13 Multilinear Maps. Expanding a little on the abstraction above, a fresh multilinear map encoding of a value $a \in \mathbb{Z}_p$ at level i is obtained by first sampling a random value μ from $\mathbb{Z}_p$ and then encoding $\mathsf{Enc}_i(a + \mu \cdot p)$. Homomorphic operations can be performed just as before, except that the randomnesses from different encodings also get computed on. Specifically, $\mathsf{Enc}_i(a + \mu \cdot p) \oplus \mathsf{Enc}_i(b + \nu \cdot p)$ yields $\mathsf{Enc}_i(a + b + (\mu + \nu) \cdot p)$ and multiplication $\mathsf{Enc}_i(a + \mu \cdot p) \odot \mathsf{Enc}_j(b + \nu \cdot p)$ yields $\mathsf{Enc}_{i+j}(a \cdot b + (b \cdot \mu + a \cdot \nu + \mu \cdot \nu \cdot p) \cdot p)$ if $i + j \leq \kappa$ but is meaningless otherwise. An imperfection of the zero-test procedure is a feature characterized by two phenomena:

1. On input $\mathsf{Enc}_\kappa(0 + r \cdot p)$ the zero-test procedure additionally reveals r in a "scrambled" form.
2. For certain efficiently computable polynomials f and a collection of scrambled values $\{r_i\}$ it is efficient to check if $f(\{r_i\}) = 0 \mod p$ or not for any choice of r_i's.[5]

This imperfection has been exploited to perform attacks in prior works, such as the one by Miles et al. [MSZ16].[6]

Matrix Branching Programs. A matrix branching program of length ℓ for n-bit inputs is a sequence $BP = \left\{A_0, \left\{A_{i,0}, A_{i,1}\right\}_{i=1}^{\ell}, A_{\ell+1}\right\}$, where $A_0 \in \{0,1\}^{1\times 5}$, $A_{i,b}$'s for $i \in [\ell]$ are in $\{0,1\}^{5\times 5}$ and $A_{\ell+1} \in \{0,1\}^{5\times 1}$. Without providing details, we note that the choice of 5×5 matrices comes from Barrington's Theorem [Bar86]. We use the notation $[n]$ to describe the set $\{1, \dots, n\}$. Let inp be a fixed function such that $\mathsf{inp}(i) \in [n]$ is the input bit position examined in the i^{th} step of the branching program. The function computed by this matrix branching program is

$$f_{BP}(x) = \begin{cases} 0 & \text{if } A_0 \cdot \prod_{i=1}^{\ell} A_{i,x[\mathsf{inp}(i)]} \cdot A_{\ell+1} = 0 \\ 1 & \text{if } A_0 \cdot \prod_{i=1}^{\ell} A_{i,x[\mathsf{inp}(i)]} \cdot A_{\ell+1} \neq 0 \end{cases},$$

where $x[\mathsf{inp}(i)] \in \{0,1\}$ denotes the $\mathsf{inp}(i)^{th}$ bit of x.

The branching program described above inspects one bit of the input in each step. More generally, multi-arity branching programs inspect multiple bits in each step. For example, dual-input programs inspect two bits during each step. Our work only works against single-input branching programs, hence we restrict ourselves to that setting.

Exploiting the Imperfection/Weakness. At a high level, obfuscation of a branching program $BP = \{A_0, \{A_{i,0}, A_{i,1}\}_{i=1}^{\ell}, A_{\ell+1}\}$ yields a collection of encodings $\{M_0, \{M_{i,0}, M_{i,1}\}_{i=1}^{\ell}, M_{\ell+1}\}$, say

[5] One can alternatively consider the scrambled values as polynomials over $\{r_i\}$ and then check if $f(\{r_i\})$ is identically zero in $\mathbb{Z}_p$.

[6] Recent works such as [GMM+16, DGG+16], have attempted to realize obfuscation schemes secure against such imperfection.

all of which are obtained at level-1.[7] We let $\{Z_0, \{Z_{i,0}, Z_{i,1}\}_{i=1}^{\ell}, Z_{\ell+1}\}$ denote the randomnesses used in the generation of these encodings, where each Z corresponds to a matrix of random values (analogous to r above) in $\mathbb{Z}_p$. For every input x such that $BP(x) = 0$, we have that $M_0 \odot \bigodot_{i=1}^{\ell} M_{i,x[\mathsf{inp}(i)]} \odot M_{\ell+1}$ is an encoding of 0, say of the form $\mathsf{Enc}(0 + r_x \cdot p)$ from which r_x can be learned in a scrambled form. The crucial observations of Miles et al. [MSZ16] are: (1) for every known obfuscation construction, r_x is a *program dependent* function of $\{Z_0, \{Z_{i,0}, Z_{i,1}\}_{i=1}^{\ell}, Z_{\ell+1}\}$, and (2) for a large enough $m \in \mathbb{Z}$ the values $\{r_{x_k}\}_{k=1}^{m}$ must be correlated, which in turn implies that there exists a (program-dependent) efficiently computable function f^{BP} and input choices $\{x_k^{BP}\}_{k=1}^{m}$ such that for all $k \in [m]$, $BP(x_k^{BP}) = 0$ and $f^{BP}(\{r_{x_k^{BP}}\}_{k=1}^{m}) = 0 \mod p$.[8] Further, just like Miles et al. we are interested in constructing an attacker for the *indistinguishability* notion of obfuscation. In this case, given two arbitrarily distinct programs BP and BP' (such that $\forall x, BP(x) = BP'(x)$) an attacker needs to distinguish between the obfuscations of BP and BP'. Therefore, to complete the attack, it suffices to argue that for the sequence of $\{r'_{x_k^{BP'}}\}$ values obtained from execution of BP' it holds that, $f^{BP}(\{r'_{x_k^{BP'}}\}_{k=1}^{m}) \neq 0 \mod p$. Hence, the task of attacking any obfuscation scheme reduces to the task of finding such distinguishing function f^{BP}.

Miles et al. [MSZ16] accomplishes that by presenting specific examples of branching programs, both of which implement the constant zero function, and a corresponding distinguishing function. They then extend the attack to other related branching programs that are padded with those constant-zero programs. The details of their attack [MSZ16] is quite involved, hence we jump directly to the intuition behind our envisioned more general attacks.

Partial Inequivalence of Branching Programs and Our Attacks. We start with the following observation. For [BGK+14]-like-obfuscations for any branching program $BP = \{A_0, \{A_{i,0}, A_{i,1}\}_{i=1}^{\ell}, A_{\ell+1}\}$ the value $s_x = r_x \mod p$ looks something like: [9]

$$s_x \simeq \prod_{i=1}^{\ell} \alpha_{i,x[\mathsf{inp}(i)]} \underbrace{\left[\sum_{i=0}^{\ell+1} \left(\prod_{j=0}^{i-1} A_{j,x_{\mathsf{inp}(j)}} \cdot Z_{i,x[\mathsf{inp}(i)]} \cdot \prod_{j=i+1}^{\ell+1} A_{j,x_{\mathsf{inp}(j)}}\right)\right]}_{t_x},$$

where $\{Z_0, \{Z_{i,0}, Z_{i,1}\}_{i=1}^{\ell}, Z_{\ell+1}\}$ are the randomnesses contributed by the corresponding encodings. Let $\overline{x}$ denote the value obtained by flipping every bit of x. Now observe that the product value $\Lambda = \prod_{i=1}^{\ell} \alpha_{i,x[\mathsf{inp}(i)]} \cdot \alpha_{i,\overline{x}[\mathsf{inp}(i)]}$ is independent of x. Therefore, $u_x = s_x \cdot s_{\overline{x}} = \Lambda \cdot t_x \cdot t_{\overline{x}}$. Absorbing Λ in the $\{Z_{i,0}, Z_{i,1}\}_{i=1}^{\ell}$, we have that u_x is quadratic in the randomness values $\{Z_0, \{Z_{i,0}, Z_{i,1}\}_{i=1}^{\ell}, Z_{\ell+1}\}$, or linear in the random terms ZZ' obtained by multiplying every choice of $Z, Z' \in \{Z_0, \{Z_{i,0}, Z_{i,1}\}_{i=1}^{\ell}, Z_{\ell+1}\}$. In other words if BP evaluates to 0 both on inputs x and $\overline{x}$, the values revealed by two zero-test operations give one linear equation where the coefficients of the linear equations are program dependent. Now, if BP implements a "sufficiently non-evasive" circuit,(e.g. a PRF) such that there exist sufficiently many such inputs $x, \overline{x}$ for which

[7] Many obfuscation constructions use more sophisticate leveling structure, typically referred to as so-called "straddling sets". However we emphasize that, this structure does not affect our attacks. Therefore we will just ignore this in our setting.

[8] This follows from the existence of an annihilating polynomial for any over-determined non-linear systems of equations. We refer to [Kay09] for more details.

[9] Obtaining this expression requires careful analysis that is deferred to the main body of the paper. Also, by abuse of notation let $A_{0,x_{\mathsf{inp}(0)}} = A_0$, $A_{\ell+1,x_{\mathsf{inp}(\ell+1)}} = A_{\ell+1}$, $Z_{0,x_{\mathsf{inp}(0)}} = Z_0$ and $Z_{\ell+1,x_{\mathsf{inp}(\ell+1)}} = Z_{\ell+1}$.

$BP(x) = BP(\overline{x}) = 0$, then collecting sufficiently many values $\{x_k^{BP}, u_{x_k^{BP}}\}_{k=1}^m$, we get a dependent system of linear relations. Namely, there exist $\{\nu_k^{BP}\}_{k=1}^m$ such that $\sum_{k=1}^m \nu_k^{BP} \cdot u_{x_k^{BP}} = 0$. In other words, $\sum_{k=1}^m \nu_k^{BP} \cdot r_{x_k^{BP}} \cdot r_{\overline{x}_k^{BP}} = 0 \mod p$, where $\{\nu_k^{BP}\}_{k=1}^m$ depends only on the description of the branching program BP.

We remark that, in the process of linearization above we increased (by a quadratic factor) the number of random terms in the system. However, this can be always compensated by using more equations, because the number of random terms is $O(poly(n))$ (n is the input length) whereas the number of choices of input x is $2^{O(n)}$ which implies that there are exponentially many r_x available.

Note that for any branching program BP' that is "different enough" from BP, we could expect that $\sum_{k=1}^m \nu_k^{BP} \cdot r'_{x_k^{BP}} \cdot r'_{\overline{x}_k^{BP}} \neq 0 \mod p$ where $r'_{x_k^{BP}}$ are values revealed in executions of an obfuscation of BP'. This is because the values $\{\nu_k^{BP}\}_{k=1}^m$ depend on the specific implementation of BP through terms of the form $\prod_{j=0}^{i-1} A_{j,x[\mathsf{inp}(i)]}$ and $\prod_{j=i+1}^{\ell+1} A_{j,x[\mathsf{inp}(i)]}$ in s_x above. Two branching programs that differ from each other in this sense are referred to as partially inequivalent.[10]

Generalizing Partial Inequivalence. We only attempted to observe values mod p as they sufficed for our desired attacks. However, for (pairs of) branching programs that can not be distinguished using partial inequivalence, we could attempt to consider looking at terms $r_x \cdot r_{\overline{x}}$ mod p^2. In more detail, we just argued that for a given BP (with sufficiently many $x, \overline{x}$ such that $BP(x) = BP(\overline{x}) = 0$) there always exists a vector $\{\nu_k^{BP}\}_{k=1}^m$ such that $\sum_{k=1}^m \nu_k^{BP} \cdot r_{x_k^{BP}} \cdot r_{\overline{x}_k^{BP}} = 0$ mod p. However, it may well be possible that for a some BP and BP' all such $\{\nu_k^{BP}\}_{k=1}^m$ are equal. That's the exact case when they can not be distinguished via our attack. However, now for a tuple of inputs $\mathbf{x} = \{x_k\}_{k=1}^m$ we have already obtained one value for each program, say $\gamma_{\mathbf{x}}^{BP}, \gamma_{\mathbf{x}}^{BP'}$, that are multiples of p; therefore we can repeat the exact same steps, but now with respect to mod p^2 with several such values $\{\gamma_{\mathbf{x}_k}^{BP}\}_{k=1}^{m'}$ for many tuples $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{m'}\}$. In fact, more generally we can keep repeating this procedure until we get a distinguishing vector. However, the number of random variables may potentially grow doubly-exponentially with the number of such iterations, but since we have exponentially many inputs we can continue this procedure until mod p^c for a constant c for sufficiently non-evasive functions. However, in practice this growth in system-size can be problematic as it is not clear how the performance will be for some large value of c (even a large constant) in real-world implementations. We do not analyze this case formally.[11]

1.3 What Programs are Partially Inequivalent? Attack on NC^1 circuits.

The condition we put on seems to be a quite general one based on our understanding. However finding evidence analytically seemed like a hard task for a general setting.[12] Nonetheless, we manage to show via implementation in Sage [S+16] (c.f. Appendix A) that the attack works on a pair of branching programs obtained from a pair of simple NC^1 circuits, (say C_0, C_1) (see Sec. 7 for the

[10] Note that the only other constraint we need is that both BP and BP' evaluates to 0 for sufficiently many inputs, which we include in the definition(c.f. Def. 4.2) of partial inequivalence.

[11] The formal analysis would require much involved tensor-product computations than the current one as detailed in Sec. 4, as the structures of those higher order partial products would become more complicated for those cases. However, applying (extensions of) the properties of matrix-tensor products (see Sec. 2.2 for details) a similar form of equations (albeit of larger dimensions) can be obtained that can be analyzed in analogous manner.

[12] Note that, the analysis of Miles et al. uses 2×2 matrices in addition to simple branching programs. These simplifications allow them to base their analysis on many facts related to the structures of these programs. Our aim here is to see if the attack works for programs obtained via implementing Barrington's Theorem. So, unfortunately it is not clear if their approach can be applicable here as the structure of the programs yielded via Barrington's Theorem become too complicated (and much larger in size) to analyze.

exact circuits) by applying Barrington's Theorem. The circuits take 4 bits of inputs and on any input they evaluate to 0. In our attack we use all possible 16 inputs. Furthermore, we can escalate the attack to any pair of NC^1 circuits (E_0, E_1) where $E_b = \neg C_b \wedge D_b$ $(b \in \{0,1\})$ for practically any two NC^1 circuits D_0, D_1 (we need only one input x for which $D(x) = D(\overline{x}) = 0$). We now take again a sequence of 16-inputs such that we vary the parts of all the inputs going into C_b and keep the part of inputs read by D_b fixed to x. Intuitively, since the input to D_b is always the same, each evaluation chooses the exactly same randomnesses (that is Z_i's as mentioned above) always. Hence in the resulting system all the random variables can be replaced by a single random variable. So we can "collapse" the circuit $\neg C_b \wedge D_b$ effectively to a much smaller circuit namely, $\neg C_b \wedge 0$. Finally, again via our Sage-implementation we show that for circuits $\neg C_0 \wedge 0$ and $\neg C_1 \wedge 0$ the corresponding branching programs are partially inequivelent.

As a consequence of the above we are also able to show examples of universal circuits U_b for which the same attack works. Since the circuit D can be almost any arbitrary NC^1 circuit, we can, in particular use any universal circuit U' and carefully combine that with C to obtain our attackable universal circuit U that results in partially inequivalent Barrigton programs when compiled with any two arbitrary NC^1 circuits. The details are provided in Sec. 7.3.

1.4 Roadmap

The rest of the paper is organized as follows. We provide basic definitions in Sec. 2. In Sec. 3 we formalize our abstract-attack model that is mostly similar to the attack model considered by Miles et al. [MSZ16]. In Sec. 4 we formalize partial inequivalence of two branching programs. In Sec. 5 we describe our annihilation attack in the abstract model for two partially inequivalent branching programs. In Sec. 6 we then extend the abstract attack to real-world attack in GGH13 setting. Finally in Sec. 7 we provide details on our example NC^1 circuits for which the corresponding branching programs generated via Barrington's Theorem are partially inequivalent.

Additionally, in Appendix A we provide some details on our implementations in Sage.

2 Notations and Preliminaries

2.1 Notation

We denote the set of natural numbers $\{1, 2, \ldots\}$ by $\mathbb{N}$, the set of all integers $\{\ldots, -1, 0, 1 \ldots\}$ by $\mathbb{Z}$ and the set of real numbers by $\mathbb{R}$. We use the notation $[n]$ to denote the set of first n natural numbers, namely $[n] \stackrel{\text{def}}{=} \{1, \ldots, n\}$.

For any bit-string $x \in \{0,1\}^n$ we let $x[i]$ denotes the i-th bit. For a matrix A we denote its i-th row by $A[i, \star]$, its j-th column by $A[\star, j]$ and the element in the i-th row and j-th column by $A[i, j]$. The i-th element of a vector $\boldsymbol{v}$ is denoted by $\boldsymbol{v}[i]$.

Bit-Strings. The compliment of $x \in \{0,1\}^n$ is denoted by $\overline{x}$ and defined as: $\overline{x} \stackrel{\text{def}}{=} 1^n \oplus x$, where $\oplus$ denotes the bitwise XOR operation. The hamming weight of $x \in \{0,1\}^n$ denoted by $\mathsf{Ham}(x)$ is equal to $\sum_i x[i]$.

Matrices. The transpose of A is denoted by A^T. We denote matrix multiplications between two matrices A and B by $A \cdot B$ whereas scalar multiplications between one scalar a with a matrix (or scalar) A by aA. A boolean matrix is a matrix for which each of its entries is from $\{0,1\}$. A permutation matrix is a boolean matrix such that each of its rows and columns has exactly one 1.

Concatenation of two matrices A, B of dimensions $d_1 \times d_2$ and $d_1 \times d_2'$ is a $d_1 \times (d_2 + d_2')$ matrix denoted by $[A \mid B]$. For multiple matrices $A_1, A_2, \ldots, A_m$ the concatenation is denoted as $[\mathbf{II}_{i \in [n]} A_i]$.

Vectors. Matrices of dimension $1 \times d$ and $d \times 1$ are referred to as row-vectors and column-vectors, respectively. Unless otherwise mentioned, by default we assume that a vector is a row-vector. Any matrix operation is also applicable for vectors. For example, the inner product $\boldsymbol{a} \cdot \boldsymbol{b}$ is a scalar defined as $\boldsymbol{a} \cdot \boldsymbol{b} \stackrel{\text{def}}{=} \sum_{i=1}^{d} \boldsymbol{a}[i]\boldsymbol{b}[i]$, where $\boldsymbol{a}$ and $\boldsymbol{b}$ are row and column vectors of dimension d respectively. We define the *vectorization* of any matrix M of dimension $d_1 \times d_2$ to be a column vector of dimension $d_1 d_2 \times 1$ that is obtained by concatenating the rows of the matrix M and then taking the transpose. We denote:

$$\mathbf{vec}\,(M) = \big[M[1,\star] \mid M[2,\star] \mid \cdots \mid M[d_1,\star]\big]^T.$$

Note that if M is a column-vector then $\mathbf{vec}\,(M) = M$ and if M is a row-vector then $\mathbf{vec}\,(M) = M^T$.

2.2 Matrix Products

Below, we provide additional notation and background on matrix products that will be needed in our technical sections.

Definition 2.1 (Matrix Tensor Product (Kronecker Product))**.** *The Tensor Product of a $d_1 \times d_2$ matrix A and a $d_1' \times d_2'$ matrix B is a $d_1 d_1' \times d_2 d_2'$ matrix defined as:*

$$A \otimes B = \begin{bmatrix} A[1,1]B & \cdots & A[1,d_2]B \\ \vdots & \ddots & \vdots \\ A[d_1,1]B & \cdots & A[d_1,d_2]B \end{bmatrix}$$

where $A[i,j]B$ is a matrix of dimension $d_1' \times d_2'$ that is a scalar product of the scalar $A[i,j]$ and matrix B.

Property 2.2 (Rule of Mixed Product)**.** *Let A, B, C and D be matrices for which the matrix multiplications $A \cdot B$ and $C \cdot D$ is defined. Then we have:*

$$(A \cdot B) \otimes (C \cdot D) = (A \otimes C) \cdot (B \otimes D).$$

Property 2.3 (Matrix Equation via Tensor Product)**.** *Let A, X and B be matrices such that the multiplication $A \cdot X \cdot B$ is defined, then we have that:*

$$\mathbf{vec}\,(A \cdot X \cdot B) = (A \otimes B^T) \cdot \mathbf{vec}\,(X)$$

We define a new matrix product.

Definition 2.4 (Row-wise Tensor Product of Matrices)**.** *Let A and B be two matrices of dimensions $d_1 \times d_2$ and $d_1 \times d_2'$ respectively. Then the row-wise tensor product of A and B is a matrix C of dimension $d_1 \times d_2 d_2'$ such that each row of C is a tensor product of rows of A and B. Formally,*

$$C = A \boxtimes B \text{ where } C[i,\star] \stackrel{\text{def}}{=} A[i,\star] \otimes B[i,\star].$$

The following fact is straightforward to see.

Fact 2.5 (Concatenation of Row-wise Tensors)**.** *Let* $A \stackrel{\text{def}}{=} [A_1 \mid A_2 \mid \cdots \mid A_m]$ *and* $B \stackrel{\text{def}}{=} [B_1 \mid B_2 \mid \cdots \mid B_n]$ *be two matrices, then we have:*

$$A \boxtimes B = [\mathbf{\Pi}_{i\in[m],j\in[n]} A_i \boxtimes B_j].$$

Definition 2.6 (Permutation Equivalence)**.** *Let* A, B *be matrices with dimensions* $d_1 \times d_2$*, then* A *and* B *are called permutation equivalent if there exists a permutation matrix* P *such that* $A = B \cdot P$*. We denote by* $A \stackrel{\text{per}}{=} B$

Property 2.7. *For any two matrices* A *and* B *of dimensions* $d_1 \times d_2$ *and* $d_1 \times d_2'$ *respectively we have that:*

$$A \boxtimes B \stackrel{\text{per}}{=} B \boxtimes A$$

Proof. Let $C \stackrel{\text{def}}{=} A \boxtimes B$ then for any $k \in [d_2 d_2']$ the k-th column of C can be written as:

$$C[\star, k] = \begin{bmatrix} A[1,j]B[1,i] \\ \vdots \\ A[d_1,j]B[d_1,i], \end{bmatrix}$$

where $i = k \mod d_2'$ and $j = \frac{k-i}{d_2'} + 1$. For $\ell \in [d_2']$, define the matrix

$$D_\ell = [C[\star,\ell] \mid C[\star,\ell+d_2'] \mid \ldots \mid C[\star,\ell+d_2'(d_2-1)]].$$

Observe that we can express $B \boxtimes A$ as follows:

$$B \boxtimes A = [D_1 \mid \ldots \mid D_{d_2'}] = (A \boxtimes B) \cdot P$$

where P is a permutation matrix that maps the k-th column of $A \boxtimes B$ to the $d_2(i-1)+j$-th column where $i = k \mod d_2'$ and $j = \frac{k-i}{d_2'} + 1$. □

2.3 Column Space of a Matrix

Our attacks will require certain properties on the column space of certain matrices which we elaborate on below.

Definition 2.8 (Column Space of a matrix)**.** *Let* A *be a matrix of dimension* $d_1 \times d_2$*. Then the column space of* A *is defined as the vector space generated by linear combinations of its columns, formally the column space contains all vectors generated as* $\sum_{i=1}^{d_2} c_i A[\star, i]$ *for all choices of* $c_i \in \mathbb{R}$*. We denote the column-space of* A *by* $\mathsf{colsp}(A)$*.*

Definition 2.9 (Null-space of a matrix)**.** [13] *Let* A *be a matrix of dimension* $d_1 \times d_2$*. Then the null-space of* A *consists of all vectors* $\boldsymbol{v}$ *of dimension* $1 \times d_1$ *for which* $\boldsymbol{v} \cdot A = 0$*. We denote the null-space of* A *by* $\mathsf{nullsp}(A)$*.*

We state some basic property of the above vector spaces.

Property 2.10 ([Ogu16])**.** *Let* A *and* B *be two matrices of dimensions* $d_1 \times d_2$*. Then the following statements are equivalent:*

[13] Traditionally such space is called left-null space or co-kernel.

- $\mathsf{colsp}(A) = \mathsf{colsp}(B)$.
- $\mathsf{nullsp}(A) = \mathsf{nullsp}(B)$.
- *There exists an invertible square matrix C such that $A \cdot C = B$.*

Corollary 2.11. *Since $A \overset{\text{per}}{=} B$ is a special case of item-3 in the above property, we have that $A \overset{\text{per}}{=} B \implies \mathsf{colsp}(A) = \mathsf{colsp}(B)$.*

Combining above corollary along with Property 2.7 we can get the following corollary.

Corollary 2.12. *For any two matrices A and B having equal number of rows we have that*

$$\mathsf{colsp}(A \boxtimes B) = \mathsf{colsp}(B \boxtimes A)$$

Next we prove the following lemma that will be useful later in Sec. 7.

Lemma 2.13. *Let A and B be two boolean matrices of dimensions $d_1 \times d_2$ and $d_1 \times d_2'$ such that both A and B have equal number of 1's in each of its rows. Then we have:*

$$\mathsf{colsp}(A) \subseteq \mathsf{colsp}(A \boxtimes B) \quad \textit{and} \quad \mathsf{colsp}(B) \subseteq \mathsf{colsp}(A \boxtimes B)$$

Proof. For each column $A[\star, j]$ of A, we define the matrix $W_j \in \{0,1\}^{d_1 \times d_2'}$ as a row-wise tensor product between $A[\star, j]$ and B:

$$W_j = A[\star, j] \boxtimes B.$$

Summing up the columns of W_j we get:

$$\sum_{j'} W_j[\star, j'] = \begin{bmatrix} A[1,j] \sum_{j'} B[1,j'] \\ \vdots \\ A[d_1,j] \sum_{j'} B[d_1,j'] \end{bmatrix} = c(A[\star, j]).$$

for some integer c. Moreover we can write $A \boxtimes B$ as:

$$A \boxtimes B = [W_1 \mid W_2 \mid \ldots \mid W_{d_2}].$$

Hence there is a linear combination of columns of $A \boxtimes B$ that generates the j-th column of A for any $j \in [d_2]$. This allows us to conclude that $\mathsf{colsp}(A) \subseteq \mathsf{colsp}(A \boxtimes B)$. Now similar to the proof of the statement $\mathsf{colsp}(A) \subseteq \mathsf{colsp}(A \boxtimes B)$ we can prove that:

$$\mathsf{colsp}(B) \subseteq \mathsf{colsp}(B \boxtimes A).$$

From Corollary 2.12 we get that $\mathsf{colsp}(A \boxtimes B) = \mathsf{colsp}(B \boxtimes A)$. This allows us to conclude that $\mathsf{colsp}(B) \subseteq \mathsf{colsp}(A \boxtimes B)$. □

2.4 Branching Programs

In this subsection, we recall definitions of branching programs.

Definition 2.14 (w-ary Input-Oblivious Matrix Branching Program [BGK+14])**.** *A w-ary input oblivious matrix branching program of dimension d, length ℓ over n-bit inputs is given by a tuple,*

$$\mathbf{A} = (\mathsf{inp}, A_0, \{A_{i,b}\}_{i\in[\ell], b\in\{0,1\}^w}, A_{\ell+1})$$

where $\mathsf{inp}(\cdot) : [\ell] \to [n]^w$ is a function such that $\mathsf{inp}(i)$ is the set of w bit locations of the input examined in step i; $A_{i,b}$ are permutation matrices over $\{0,1\}^{d\times d}$ and $A_0 \in \{0,1\}^{1\times d} \setminus (0^d)^T$, $A_{\ell+1} \in \{0,1\}^{d\times 1} \setminus 0^d$ are fixed bookend vectors such that:

$$A_0 \cdot A \cdot A_{\ell+1} = \begin{cases} 0 & \text{if and only if } A = I_{d\times d} \\ 1 & \text{otherwise.} \end{cases} \tag{1}$$

The output of the matrix branching program on an input $x \in \{0,1\}^n$ is given by:

$$\mathbf{A}(x) = \begin{cases} 1 & \text{if } A_0 \left(\prod_{i\in[\ell]} A_{i,x[\mathsf{inp}(i)]}\right) A_{\ell+1} = 1 \\ 0 & \text{if } A_0 \left(\prod_{i\in[\ell]} A_{i,x[\mathsf{inp}(i)]}\right) A_{\ell+1} = 0 \end{cases},$$

where $\mathsf{inp}(i)$ denotes the set of locations that are inspected at step i of $\mathbf{A}$ and $x[\mathsf{inp}(i)]$ denotes the bits of x at locations $\mathsf{inp}(i)$. A w-ary branching program is said to be input-oblivious *if the function inp is fixed and independent of the program $\mathbf{A}$.*

Remark 2.15. *A 1-ary branching program is also called a* single-input *branching program. Unless otherwise stated we will always assume that any branching program is single-input and input-oblivious.*

Barrington [Bar86] showed that all circuits in NC^1 can be equivalently represented by a branching program of polynomial length. We provide the theorem statement below adapted to our definition of branching programs.

Theorem 2.16 (Barrington's Theorem[Bar86])**.** *For any depth-D, fan-in-2 boolean circuit C, there exists an input oblivious branching program of matrix-dimension 5 and length at most 4^D that computes the same function as the circuit C.*

Given a circuit C of depth D, Barrington's Theorem provides yield a single-input branching program of matrix dimension 5 implementing circuit C. We stress that the specific implementation obtained by use of Barrigton's depends on the specific choices made in its implementation and therefore the obtained implementation is *not* unique. Sometimes the branching program obtained via applying Barrington's Theorem to a circuit is called the Barrington-implementation of that circuit. We choose a specific one for our Sage-implementation. The details are provided in Appendix. A.

2.5 Indistinguishability Obfuscation

Below, we recall the notion of indistinguishability obfuscation ($i\mathcal{O}$).

Definition 2.17 (Indistinguishability Obfuscator ($i\mathcal{O}$)[GGH+13b])**.** *A uniform PPT machine $i\mathcal{O}$ is called an* indistinguishability obfuscator *for a circuit class $\{\mathcal{C}_\lambda\}$ if the following conditions are satisfied:*

- *For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have that*

$$\Pr[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(\lambda, C)] = 1$$

- *For any (not necessarily uniform) PPT distinguisher D, there exists a negligible function α such that the following holds: For all security parameters $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$, we have that if $C_0(x) = C_1(x)$ for all inputs x, then*

$$\left| \Pr\left[D(i\mathcal{O}(\lambda, C_0)) = 1\right] - \Pr\left[D(i\mathcal{O}(\lambda, C_1)) = 1\right] \right| \leq \alpha(\lambda)$$

3 Attack Model for Investigating Annihilation Attacks

Miles, Sahai, and Zhandry [MSZ16] describe an abstract obfuscation scheme, designed to encompass the main ideas of [BGK+14, BR14, AGIS14, PST14, MSW14, BMSZ16a] for the purposes of investigating annihilation attacks. We use the same abstract attack model as the starting point for our new attacks. Below, we first describe the model, obfuscation in this model and what violating indistinguishability obfuscation security means.

3.1 Annihilation Attack Model

We describe the abstract annihilation attack model. An abstract model is parameterized with n *arbitrary secret* variables $X_1, ..., X_n$, m *random secret* variables $Z_1, ..., Z_m$, a *special secret* variable g. Then the *public* variables $Y_1, \ldots, Y_m$ are such that $Y_i := q_i(\{X_j\}_{j\in[n]}) + gZ_i$ for some polynomials q_i. The polynomials are defined over a field $\mathbb{F}$.[14] An abstract model attacker $\mathcal{A}$ may adaptively make two types of queries:

- **Type 1: Pre-Zeroizing Computation.** In a **Type 1** query, the adversary $\mathcal{A}$ submits a "valid" polynomial p_k on the public Y_i. Here, valid polynomials are those polynomials as enforced by the graded encodings.[15]

  Then, we expand the representation of the (public) polynomial on Y_i in order to express p_k as a polynomial of the (private) formal variables X_j, Z_i, g stratified in the powers of g as follows:

$$p_k = p_k^{(0)} + g \cdot p_k^{(1)} + g^2 \cdot p_k^{(2)} + \ldots.$$

  If p_k is identically 0 or if $p_k^{(0)}$ is not identically 0, then the adversary $\mathcal{A}$ receives $\perp$ in return. Otherwise, the adversary $\mathcal{A}$ receives a new handle to a new variable W_k, which is set to be

$$W_k := p_k/g = p_k^{(1)} + g \cdot p_k^{(2)} + g^2 \cdot p_k^{(3)} + \ldots.$$

[14] Looking ahead, the arbitrary variables represent the plain-texts (the branching program or circuit to be obfuscated) of encoding, the random variables represent the randomness of encodings generated by the obfuscator, the variable g represents the "short" generator $\boldsymbol{g}$ of the ideal lattice and the public variables represent the encodings available to the attacker.

[15] For example, for a branching program obfuscation it must be a correct (and complete) evaluation of a branching program on some specific input as directed by the inp function of the program.

- **Type 2: Post-Zeroizing Computation.** In a **Type 2** query, the adversary $\mathcal{A}$ is allowed to submit *arbitrary* polynomials r of polynomial degree, on the W_k that it has seen so far. We again view $r(\{W_k\})$ as a polynomial of the (secret) formal variables X_j, Z_i, g, and write it as:
$$r = r^{(0)} + g \cdot r^{(1)} + g^2 \cdot r^{(2)} + \dots.$$
If $r^{(0)}$ is identically 0, then the adversary $\mathcal{A}$ receives 0 in return. Otherwise, the adversary $\mathcal{A}$ receives 1 in return.

Comparing the Abstract Model to other Idealized Models. We briefly compare the Abstract Model described above to the ideal graded encoding model that has traditionally been used to argue about obfuscation security in prior works, e.g. as in the [BR14, BGK+14]. All adversarial behavior allowed within the Ideal Graded Encoding model is captured by **Type 1** queries in the Abstract Model and the **Type 2** queries are not considered. The works of [GMM+16, DGG+16] argue security in this new model also referred to as the *Weak Multilinear Map Model*.

3.2 Obfuscation in the Annihilation Attack Model

The abstract obfuscator $\mathcal{O}$ takes as input a single-input branching program A of length ℓ, input length n. We describe our obfuscation using notation slightly different from Miles et al. [MSZ16] as it suits our setting better and is closer to notation of branching programs (Def. 2.4). The branching program has an associated input-indexing function $\mathsf{inp} : [\ell] \to [n]$. The branching program has $2\ell+2$ matrices $A_0, \{A_{i,b}\}_{i\in[\ell],b\in\{0,1\}}, A_{\ell+1}$. In most generality, in order to evaluate a branching program on input x, we compute the matrix product
$$A(T) = A_0 \cdot \prod_{i=1}^{\ell} A_{i,x[\mathsf{inp}(i)]} \cdot A_{\ell+1},$$
where $x[\mathsf{inp}(i)]$ denotes the bit of x at locations described by the set $\mathsf{inp}(i)$. Finally the program outputs 0 if and only if $A(T) = 0$.

The abstract obfuscator randomizes its input branching program by sampling random matrices $\{R_i\}_{i\in[\ell+1]}$ (Killian randomizers) and random scalars $\{\alpha_{i,b}\}_{i\in[\ell],b\in\{0,1\}}$, then setting
$$\widetilde{A_0} := A_0 \cdot R_1^{adj}, \qquad \widetilde{A_{i,b}} := \alpha_{i,b}(R_i \cdot A_{i,b} \cdot R_{i+1}^{adj}), \qquad \widetilde{A_{\ell+1}} := R_{\ell+1} \cdot A_{\ell+1}.$$
that are the abstract model's arbitrary secret variables. Here R^{adj} denotes the adjugate matrix of R that satisfies $R^{adj} \cdot R = det(R) \cdot I$. Then the obfuscator defines the public variables to be
$$Y_0 := \widetilde{A_0} + gZ_0; \quad Y_{i,b} := \widetilde{A_{i,b}} + gZ_{i,b}; \quad Y_{\ell+1} := \widetilde{A_{\ell+1}} + gZ_{\ell+1},$$
where g is the special secret variable and Z_is are the random variables. This defines the abstract obfuscated program $\mathcal{O}(A) = \{Y_i\}_i$. The set of valid **Type 1** polynomials consists of all the honest evaluations of the branching program. This is, the allowed polynomials are
$$p_x = Y_0 \cdot \prod_{i=1}^{\ell} Y_{i,x[\mathsf{inp}(i)]} \cdot Y_{\ell+1},$$
for all $x \in \{0,1\}^n$.[16]

[16] Looking ahead, the Z_is are random noise component sampled in the encoding procedure of GGH13 maps and g is a "short" generator of the ideal lattice. The abstract model is agnostic to the exact choice of those variables, but only depends on the structure of the variables.

3.3 Abstract Indistinguishability Obfuscation Security

We define security of $i\mathcal{O}$ in the abstract model. Formally consider the following indistinguishability game consisting of three phases.

Set Up. The adversary $\mathcal{A}$ comes up with a pair of matrix branching programs $(\mathbf{A}_0, \mathbf{A}_1)$ that are (i) functionally equivalent, (ii) of same length and (iii) input oblivious and some auxiliary information aux. $\mathcal{A}$ outputs the pair $(\mathbf{A}_0, \mathbf{A}_1)$ to the challenger.

Challenge. The challenger applies the abstract obfuscator $\mathcal{O}$ to a branching program, uniformly chosen as $\mathbf{A}_b \leftarrow \{\mathbf{A}_0, \mathbf{A}_1\}$ and returns the public variables $\{Y_0, \{Y_{i,b}\}, Y_{\ell+1}\}$, generated by applying $\mathcal{O}$ to $\mathbf{A}_b$, to the adversary.

Pre-zeroing (Type-1) Queries. In this phase the adversary makes several type-1 valid queries p_k and gets back handles $\{W_1, W_2, \ldots\}$.

Post-zeroing (Type-2) Query. In this phase the adversary makes one type-2 query r with some degree $poly(\lambda)$ polynomial Q over the formal variables corresponding to handles $\{W_1, W_2, \ldots\}$ and receives a bit as a response from the challenger. Finally $\mathcal{A}$ outputs its guess $b' \in \{0, 1\}$.

Definition 3.1 (Abstract $i\mathcal{O}$ Security)**.** *An abstract obfuscation candidate $\mathcal{O}$ is called an indistinguishability obfuscator if for any probabilistic polynomial time adversary $\mathcal{A}$ the probability that $\mathcal{A}$ guesses the choice of $\mathbf{A}_b$ correctly is negligibly close to $1/2$. Formally, in the above game*

$$|\Pr[b = b'] - 1/2| \leq negl(\lambda)$$

for any security parameter $\lambda \in \mathbb{N}$, where the probability is over the randomness of $\mathcal{A}$ and the challenger.

4 Partially Inequivalent Branching Programs

In this section, we provide a formal condition on two branching programs, namely partial inequivalence, that is sufficient for launching a distinguishing attack in the abstract model. In Section 5 we prove that this condition is sufficient for the attack.[17]

All the below definitions consider single-input branching programs, but they naturally extends to multi-input setting.

Definition 4.1 (Partial Products)**.** *Let $\mathbf{A} = (\mathsf{inp}, A_0, \{A_{i,b}\}_{i\in[\ell],b\in\{0,1\}}, A_{\ell+1})$ be a single-input branching program of matrix-dimension d and length ℓ over n-bit input.*

1. *For any input $x \in \{0,1\}^n$ and any index $i \in [\ell+1] \cup \{0\}$ we define the vectors $\phi^{(i)}_{\mathbf{A},x}$ as follows:*

$$\phi^{(i)}_{\mathbf{A},x} \stackrel{\text{def}}{=} \begin{cases} \left(A_0 \cdot \prod_{j=1}^{i-1} A_{j,x[\mathsf{inp}(j)]}\right) \otimes \left(\prod_{j=i+1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot A_{\ell+1}\right)^T \in \{0,1\}^{1\times d^2} & \text{if } i \in [\ell] \\ \left(\prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot A_{\ell+1}\right)^T \in \{0,1\}^{1\times d} & \text{if } i = 0 \\ A_0 \cdot \prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \in \{0,1\}^{1\times d} & \text{if } i = \ell+1 \end{cases},$$

[17] We note that this condition is not necessary. Looking ahead, we only consider first order partially inequivalent programs in paper and remark that higher order partially inequivalent programs could also be distinguished using our techniques.

Additionally, define $\widetilde{\phi}^{(i)}_{\mathbf{A},x}$ *for any such branching program as:*

$$\widetilde{\phi}^{(i)}_{\mathbf{A},x} \stackrel{\text{def}}{=} \begin{cases} [\phi^{(i)}_{\mathbf{A},x} \mid 0^{d^2}] & \text{if } i \in [\ell] \text{ and } x[\mathsf{inp}(i)] = 0 \\ [0^{d^2} \mid \phi^{(i)}_{\mathbf{A},x}] & \text{if } i \in [\ell] \text{ and } x[\mathsf{inp}(i)] = 1 \\ \phi^{(i)}_{\mathbf{A},x} & \text{if } i = 0 \text{ or } \ell+1 \end{cases},$$

where inp *is a function from* $[\ell] \to [n]$ *and that* $x[\mathsf{inp}(x)]$ *denotes the bit of* x *corresponding to location described by* $\mathsf{inp}(x)$*.*

2. *Then the* ***linear partial product vector*** $\boldsymbol{\phi}_{\mathbf{A},x}$ *and the* ***quadratic partial product vector*** $\boldsymbol{\psi}_{\mathbf{A},x}$ *of* $\mathbf{A}$ *with respect to* x *are defined as:*

$$\boldsymbol{\phi}_{\mathbf{A},x} \stackrel{\text{def}}{=} [\widetilde{\phi}^{(0)}_{\mathbf{A},x} \mid \cdots \mid \widetilde{\phi}^{(\ell+1)}_{\mathbf{A},x}] \in \{0,1\}^{1\times(2d+2\ell d^2)}.$$

$$\boldsymbol{\psi}_{\mathbf{A},x} \stackrel{\text{def}}{=} \boldsymbol{\phi}_{\mathbf{A},x} \otimes \boldsymbol{\phi}_{\mathbf{A},\overline{x}} \in \{0,1\}^{(2d+2\ell d^2)^2},$$

where $\overline{x} = x \oplus 1^n$ *is the compliment of* x*.*

3. *For a set of inputs* $X = \{x_1, x_2, \ldots, x_m\}$ *the the* ***linear partial product matrix*** $\Phi_{\mathbf{A},X}$ *and the* ***quadratic partial product matrix*** $\Psi_{\mathbf{A},X}$ *of* $\mathbf{A}$ *with respect to* X*:*

$$\Phi_{\mathbf{A},X} \stackrel{\text{def}}{=} \begin{bmatrix} \boldsymbol{\phi}_{\mathbf{A},x_1} \\ \boldsymbol{\phi}_{\mathbf{A},x_2} \\ \vdots \\ \boldsymbol{\phi}_{\mathbf{A},x_m} \end{bmatrix} \in \{0,1\}^{m\times(2d+2\ell d^2)}$$

$$\Psi_{\mathbf{A},X} \stackrel{\text{def}}{=} \Phi_{\mathbf{A},X} \boxtimes \Phi_{\mathbf{A},\overline{X}} + \Phi_{\mathbf{A},\overline{X}} \boxtimes \Phi_{\mathbf{A},X} = \begin{bmatrix} \boldsymbol{\psi}_{\mathbf{A},x_1} + \boldsymbol{\psi}_{\mathbf{A},\overline{x}_1} \\ \boldsymbol{\psi}_{\mathbf{A},x_2} + \boldsymbol{\psi}_{\mathbf{A},\overline{x}_2} \\ \vdots \\ \boldsymbol{\psi}_{\mathbf{A},x_m} + \boldsymbol{\psi}_{\mathbf{A},\overline{x}_m} \end{bmatrix} \in \{0,1\}^{m\times(2d+2\ell d^2)^2},$$

where $\overline{X} \stackrel{\text{def}}{=} \{\overline{x}_1, \overline{x}_2, \ldots\}$*.*

18

Definition 4.2 (Partial Inequivalence)**.** *Let* $\mathbf{A}_0$ *and* $\mathbf{A}_1$ *be two single-input matrix branching programs of matrix-dimension* d *and length* ℓ *over* n*-bit input. Then they are called* ***partially inequivalent*** *if there exists a polynomial in security parameter sized set* X *of inputs such that:*

- *For every* $x \in X$*, we have that* $\mathbf{A}_0(x) = \mathbf{A}_1(x) = 0$ *and* $\mathbf{A}_0(\overline{x}) = \mathbf{A}_1(\overline{x}) = 0$*.*
- $\mathsf{colsp}\left(\Psi_{\mathbf{A}_0,X}\right) \neq \mathsf{colsp}\left(\Psi_{\mathbf{A}_1,X}\right)$*.*

[18] Note that in the above definition we add the row-wise tensors. Looking ahead, this is done to capture the commutativity in the polynomial multiplications. Namely since for any two ring elements z_1, z_2 we have $z_1 z_2 = z_2 z_1$, their coefficients add up. Also note that the sum in the above expression equivalently double the coefficients of the quadratic terms z_1^2, z_2^2. But, due to our choices of inputs $x, \overline{x}$ we would only have such terms for the bookends which are nonetheless always stays the same (in fact they are independent of the actual program) and does not affect the column-space.

Lemma 4.3. *For any matrix branching program* $\mathbf{A}$ *we have that for any two inputs* x, x' *the linear partial product vectors* $\boldsymbol{\phi}_{\mathbf{A},x}$ *and* $\boldsymbol{\phi}_{\mathbf{A},x'}$ *contain the same number of* 1*'s.*

Proof. Note that for any input x and index i, via definition of $\phi^{(i)}_{\mathbf{A},x}$, we have:

$$\phi^{(i)}_{\mathbf{A},x} = \left(A_0 \cdot P_{x,1} \otimes A^T_{\ell+1} \cdot P_{x,2}\right)$$

for some x dependent permutations $P_{x,1}$ and $P_{x,2}$. Note that A_0 is a row vector and therefore $A_0 \cdot P_{x,1}$ is also a row vector. Since $P_{x,1}$ therefore we conclude that $\mathsf{Ham}(A_0 \cdot P_{x,1}) = \mathsf{Ham}(A_0)$ where $\mathsf{Ham}(A_0)$ is the hamming weight of the vector A_0 (specifically, the number of locations at which it is 1). Similarly, $\mathsf{Ham}(P_{x,2} \cdot A_{\ell+1}) = \mathsf{Ham}(A_{\ell+1})$. Hence, the $\mathsf{Ham}(\boldsymbol{\phi}_{\mathbf{A},x}) = \mathsf{Ham}(A_0)\mathsf{Ham}(A_{\ell+1})$ which is independent of x. Consequently, $\mathsf{Ham}(\boldsymbol{\phi}_{\mathbf{A},x}) = (\ell+2)\mathsf{Ham}(A_0)\mathsf{Ham}(A_{\ell+1})$ which is also independent of x. This concludes the proof. □

5 Annihilation Attack for Partially Inequivalent Programs

In this section, we describe an abstract annihilation attack against any two branching programs that are partially inequivalent. In this section, we show an attack only in the abstract model and provide details on how it can be extended to the real GGH13 setting in Section 6. Formally we prove the following theorem.

Theorem 5.1. *Let* $\mathcal{O}$ *be the generic obfuscator described in Sec. 3.2. Then for any two functionally equivalent same length single-input branching programs* $\mathbf{A}_0, \mathbf{A}_1$ *that are partially inequivalent there exists a probabilistic polynomial time attacker that distinguishes between between* $\mathcal{O}(\mathbf{A}_0)$ *and* $\mathcal{O}(\mathbf{A}_1)$ *with noticeable probability in the abstract attack model (violating Definition 3.1).*

Proof of Theorem 5.1. Below we provide the proof.

Setup for the attack. The given branching programs $\mathbf{A}_0$ and $\mathbf{A}_1$ are provided to be functionally equivalent and partially inequivalent. Therefore there exists a set X such that: (1) for all $x \in X, \mathbf{A}_0(x) = \mathbf{A}_0(\overline{x}) = \mathbf{A}_1(x) = \mathbf{A}_1(\overline{x}) = 0$, and (2) $\mathsf{colsp}\left(\Psi_{\mathbf{A}_0,X}\right) \neq \mathsf{colsp}\left(\Psi_{\mathbf{A}_1,X}\right)$. We will assume that the adversary has access to X as auxiliary information.

Challenge. $\mathcal{A}$ receives as a challenge the obfuscation of the branching program: $\mathbf{A} \in \{\mathbf{A}_0, \mathbf{A}_1\}$ by the challenger. Recall from the description of the abstract obfuscator that, the obfuscation of program $\mathbf{A} = (\mathsf{inp}, A_0, \{A_{i,b}\}_{i\in[\ell],b\in\{0,1\}}, A_{\ell+1})$, denoted by $\mathcal{O}(\mathbf{A})$ consists of the following public variables:

$$Y_0 := A_0 \cdot R^{adj}_1 + gZ_0, \quad Y_{i,b} := \alpha_{i,b} R_i \cdot A_{i,b} \cdot R^{adj}_{i+1} + gZ_{i,b}, \quad Y_0 := R_{\ell+1} \cdot A_{\ell+1} + gZ_0,$$

where the arbitrary secret variables are:

$$\widetilde{A}_0 \stackrel{\text{def}}{=} A_0 \cdot R^{adj}_1, \quad \widetilde{A}_{i,b} \stackrel{\text{def}}{=} \alpha_{i,b}(R_{i,b} \cdot A_{i,b} \cdot R^{adj}_{i,b}), \quad \widetilde{A}_{\ell+1} \stackrel{\text{def}}{=} R_{\ell+1} \cdot A_{\ell+1};$$

for random variables (i.e. Killian randomizers) $R_1, \{R_i\}, R_{\ell+1}$ and the random secret variables are denoted by $Z_0, \{Z_{i,b}\}_{i\in[\ell],b\in\{0,1\}}, Z_{\ell+1}$ and the special secret variable is g. Via change of variables we can equivalently write:

$$Y_0 := (A_0 + gZ_0) \cdot R^{adj}_1; \quad Y_{i,b} := \alpha_{i,b} R_i \cdot (A_{i,b} + gZ_{i,b}) \cdot R^{adj}_{i+1}; \quad Y_{\ell+1} := R_{\ell+1} \cdot (A_{\ell+1} + gZ_{\ell+1}).$$

Pre-Zeroizing Computation (Type-1 queries). On receiving the obfuscation of $\mathbf{A} \in \{\mathbf{A}_0, \mathbf{A}_1\}$, $\mathcal{O}(\mathbf{A}) = \{Y_0, \{Y_{i,b}\}, Y_{\ell+1}\}$ the attacker, in the pre-zeroizing step, performs a "valid" Type-1 queries on all the inputs $X, \overline{X}$ where $X = \{x_1, \ldots, x_m\}, \overline{X} = \{\overline{x}_1, \ldots, \overline{x}_m\}$. That is, for any $x \in \{0,1\}^n$, and the abstract obfuscation $\mathcal{O}(\mathbf{A})$, the attacker queries the polynomial:

$$P_{\mathbf{A},x} = Y_0 \cdot \prod_{i=1}^{\ell} Y_{i,x[\mathsf{inp}(i)]} \cdot Y_{\ell+1}.$$

Then, expressing $P_{\mathbf{A},x}$ stratified as powers of g we obtain:

$$P_{\mathbf{A},x} = P^{(0)}_{\mathbf{A},x}(\{Y_i\}_i) + g \cdot P^{(1)}_{\mathbf{A},x}(\{Y_i\}_i) + ... + g^{\ell+2} \cdot P^{(\ell+2)}_{\mathbf{A},x}(\{Y_i\}_i)$$

for some polynomials $P^{(j)}_{\mathbf{A},x}(\{Y_i\}_i)$ ($j \in \{0, ..., \ell+1\}$). However, by Lemma 5.2 we have that:

$$P^{(0)}_{\mathbf{A},x} = \rho \widehat{\alpha}_x \mathbf{A}(x)$$

for $\rho \stackrel{\text{def}}{=} \prod_i \det(R_i)$ (or $\rho I = \prod_i R_i^{adj} R_i$) and $\widehat{\alpha}_x \stackrel{\text{def}}{=} \prod_{i=1}^{\ell} \alpha_{i,x_{\mathsf{inp}(i)}}$. Since for $x \in X$ we have that $\mathbf{A}(x) = 0$, the polynomial $P^{(0)}_{\mathbf{A},x}$ is identically 0. Consequently, for each such Type 1 query the attacker receives a new handle to a variable $W_{\mathbf{A},x}$ that can be expressed as follows:

$$W_{\mathbf{A},x} = P_{\mathbf{A},x}/g = P^{(1)}_{\mathbf{A},x} + g \cdot P^{(2)}_{\mathbf{A},x} + ... + g^{\ell+1} \cdot P^{(\ell+2)}_{\mathbf{A},x}.$$

Analogously, the attacker obtains handles $W_{\mathbf{A},\overline{x}}$. After obtaining handles

$$\{(W_{\mathbf{A},x_1}, W_{\mathbf{A},\overline{x}_1}), ...(W_{\mathbf{A},x_m}, W_{\mathbf{A},\overline{x}_m})\}$$

the attacker starts the post-zeroizing phase.

Post-Zeroizing Computation. The goal of post-zeroizing computation is to *find* a polynomial Q^{ann} of degree $poly(\lambda)$ such that following holds for some $b \in \{0,1\}$:

(i) $Q^{\mathsf{ann}}(P^{(1)}_{\mathbf{A}_b,x_1}, P^{(1)}_{\mathbf{A}_b,\overline{x}_1} ..., P^{(1)}_{\mathbf{A}_b,x_m}, P^{(1)}_{\mathbf{A}_b,\overline{x}_m}) \equiv 0.$

(ii) $Q^{\mathsf{ann}}(P^{(1)}_{\mathbf{A}_{1-b},x_1}, P^{(1)}_{\mathbf{A}_{1-b},\overline{x}_1} ..., P^{(1)}_{\mathbf{A}_{1-b},x_m}, P^{(1)}_{\mathbf{A}_{1-b},\overline{x}_m}) \not\equiv 0.$

Clearly, this leads to an attack on the obfuscation security(c.f. Definition 3.1) as $\mathcal{A}$ would receive 0 from the challenger if and only if $Q^{\mathsf{ann}}(P^{(1)}_{\mathbf{A},x_1}, P^{(1)}_{\mathbf{A},\overline{x}_1} ..., P^{(1)}_{\mathbf{A},x_m}, P^{(1)}_{\mathbf{A},\overline{x}_m})$ is identically zero, hence it would receive 0 if and only if $\mathbf{A}_b$ is chosen by the challenger in the challenge phase. To find such Q^{ann} the attacker continues as follows. Observe that by Lemma 5.2, for every $x \in X$ we have that:

$$P^{(1)}_{\mathbf{A},x} = \rho \widehat{\alpha}_x (\boldsymbol{\phi}_{\mathbf{A},x} \cdot \boldsymbol{z}^T) \tag{2}$$

$$P^{(1)}_{\mathbf{A},\overline{x}} = \rho \widehat{\alpha}_{\overline{x}} (\boldsymbol{\phi}_{\mathbf{A},\overline{x}} \cdot \boldsymbol{z}^T) \tag{3}$$

Next, multiplying the polynomials $P^{(1)}_{\mathbf{A},x}$ and $P^{(1)}_{\mathbf{A},\overline{x}}$ (Eq. 2 and Eq. 3) we get:

$$\widetilde{P}^{(1)}_{\mathbf{A},x} \stackrel{\text{def}}{=} P^{(1)}_{\mathbf{A},x} P^{(1)}_{\mathbf{A},\overline{x}} = \rho^2 \widehat{\alpha} \left((\boldsymbol{\phi}_{\mathbf{A},x} \cdot \boldsymbol{z}^T) \otimes (\boldsymbol{\phi}_{\mathbf{A},\overline{x}} \cdot \boldsymbol{z}^T) \right) \tag{4}$$

$$= \rho^2 \widehat{\alpha} \left((\boldsymbol{\phi}_{\mathbf{A},x} \otimes \boldsymbol{\phi}_{\mathbf{A},\overline{x}}) \cdot (\boldsymbol{z}^T \otimes \boldsymbol{z}^T) \right) \tag{5}$$

$$= \rho^2 \widehat{\alpha} (\boldsymbol{\psi}_{\mathbf{A},x} \cdot \boldsymbol{z}^T \otimes \boldsymbol{z}^T)$$

where $\widehat{\alpha} \stackrel{\text{def}}{=} \widehat{\alpha}_x \widehat{\alpha}_{\overline{x}}$ is now independent of input x.[19] Similarly we can also have:

$$\begin{aligned}\widetilde{P}^{(1)}_{\mathbf{A},\overline{x}} \stackrel{\text{def}}{=} P^{(1)}_{\mathbf{A},\overline{x}} P^{(1)}_{\mathbf{A},x} &= \rho^2 \widehat{\alpha} \left((\boldsymbol{\phi}_{\mathbf{A},\overline{x}} \cdot \boldsymbol{z}^T) \otimes (\boldsymbol{\phi}_{\mathbf{A},x} \cdot \boldsymbol{z}^T) \right) \\ &= \rho^2 \widehat{\alpha} \left((\boldsymbol{\phi}_{\mathbf{A},\overline{x}} \otimes \boldsymbol{\phi}_{\mathbf{A},x}) \cdot (\boldsymbol{z}^T \otimes \boldsymbol{z}^T) \right) \\ &= \rho^2 \widehat{\alpha} (\boldsymbol{\psi}_{\mathbf{A},\overline{x}} \cdot \boldsymbol{z}^T \otimes \boldsymbol{z}^T)\end{aligned}$$

However, since field multiplication is commutative, adding we get:

$$\begin{aligned}\widetilde{P}^{(1)}_{\mathbf{A},x} + \widetilde{P}^{(1)}_{\mathbf{A},\overline{x}} = 2 P^{(1)}_{\mathbf{A},x} P^{(1)}_{\mathbf{A},\overline{x}} &= \rho^2 \widehat{\alpha} (\boldsymbol{\psi}_{\mathbf{A},x} \cdot \boldsymbol{z}^T \otimes \boldsymbol{z}^T) + \rho^2 \widehat{\alpha} (\boldsymbol{\psi}_{\mathbf{A},\overline{x}} \cdot \boldsymbol{z}^T \otimes \boldsymbol{z}^T) \\ &= \rho^2 \widehat{\alpha} (\boldsymbol{\psi}_{\mathbf{A},x} + \boldsymbol{\psi}_{\mathbf{A},\overline{x}}) \cdot (\boldsymbol{z}^T \otimes \boldsymbol{z}^T)\end{aligned}$$

Using the given conditions that $\Psi_{\mathbf{A}_0,X}$ and $\Psi_{\mathbf{A}_1,X}$ have distinct column spaces (and hence distinct left-kernel) the attacker can efficiently compute (e.g. via Gaussian Elimination) a vector $\boldsymbol{v}_{\mathsf{ann}} \in \{0,1\}^{1 \times m}$ that belongs to it left-kernel, call it the *annihilating vector*, such that for some $b \in \{0,1\}$ we have:

$$\boldsymbol{v}_{\mathsf{ann}} \cdot \Psi_{\mathbf{A}_b,X} = 0 \qquad \text{but} \qquad \boldsymbol{v}_{\mathsf{ann}} \cdot \Psi_{\mathbf{A}_{1-b},X} \neq 0.$$

The corresponding annihilation polynomial Q^{ann} can be written as:

$$Q^{\mathsf{ann}}_{\boldsymbol{v}_{\mathsf{ann}}}(W_{\mathbf{A},x_1}, W_{\mathbf{A},\overline{x}_1}, \ldots, W_{\mathbf{A},x_m}, W_{\mathbf{A},\overline{x}_m}) = \boldsymbol{v}_{\mathsf{ann}} \cdot \begin{bmatrix} W_{\mathbf{A},x_1} W_{\mathbf{A},\overline{x}_1} \\ \vdots \\ W_{\mathbf{A},x_m} W_{\mathbf{A},\overline{x}_m} \end{bmatrix}$$

Observe that the coefficient of g^0 in the expression $Q^{\mathsf{ann}}_{\boldsymbol{v}_{\mathsf{ann}}}(W_{\mathbf{A},x_1}, W_{\mathbf{A},\overline{x}_1}, \ldots, W_{\mathbf{A},x_m}, W_{\mathbf{A},\overline{x}_m})$ from above is equal to $Q^{\mathsf{ann}}_{\boldsymbol{v}_{\mathsf{ann}}}(P^{(1)}_{\mathbf{A}_b,x_1}, P^{(1)}_{\mathbf{A}_b,\overline{x}_1} ..., P^{(1)}_{\mathbf{A}_b,x_m}, P^{(1)}_{\mathbf{A}_b,\overline{x}_m})$. Moreover this value for $\mathbf{A} = \mathbf{A}_b$ is:

$$Q^{\mathsf{ann}}_{\boldsymbol{v}_{\mathsf{ann}}}(P^{(1)}_{\mathbf{A}_b,x_1}, P^{(1)}_{\mathbf{A}_b,\overline{x}_1} ..., P^{(1)}_{\mathbf{A}_b,x_m}, P^{(1)}_{\mathbf{A}_b,\overline{x}_m}) = \boldsymbol{v}_{\mathsf{ann}} \cdot \frac{\Psi_{\mathbf{A}_b,X}}{2} \cdot (\boldsymbol{z} \otimes \boldsymbol{z})^T \equiv 0$$

but for $\mathbf{A}_{1-b}$:

$$Q^{\mathsf{ann}}_{\boldsymbol{v}_{\mathsf{ann}}}(P^{(1)}_{\mathbf{A}_{1-b},x_1}, P^{(1)}_{\mathbf{A}_{1-b},\overline{x}_1} ..., P^{(1)}_{\mathbf{A}_{1-b},x_m}, P^{(1)}_{\mathbf{A}_{1-b},\overline{x}_m}) = \boldsymbol{v}_{\mathsf{ann}} \cdot \frac{\Psi_{\mathbf{A}_{1-b},X}}{2} \cdot (\boldsymbol{z} \otimes \boldsymbol{z})^T \not\equiv 0.$$

Hence, the response to Type 2 query is sufficient to distinguish between obfuscation of $\mathbf{A}_b$ and $\mathbf{A}_{1-b}$ in the abstract model. This concludes the proof.

Evaluations of $P^{(0)}_{\mathbf{A},x}$ and $P^{(1)}_{\mathbf{A},x}$. Below we provide a lemma that described what the terms $P^{(0)}_{\mathbf{A},x}$ and $P^{(1)}_{\mathbf{A},x}$ look like.

Lemma 5.2. *For every $x \in \{0,1\}^n$, we have that:*

$$\begin{aligned}P^{(0)}_{\mathbf{A},x} &= \rho \widehat{\alpha}_x \mathbf{A}(x) \\ P^{(1)}_{\mathbf{A},x} &= \rho \widehat{\alpha}_x (\boldsymbol{\phi}_{\mathbf{A},x} \cdot \boldsymbol{z}^T),\end{aligned}$$

where $\rho \stackrel{\text{def}}{=} \prod_i \det(R_i)$ and $\widehat{\alpha}_x \stackrel{\text{def}}{=} \prod_{i=1}^{\ell} \alpha_{i,x_{\mathsf{inp}(i)}}$ and $\boldsymbol{z}$ is a vector consisting of the random terms $Z_0, Z_{i,b}$, and $Z_{\ell+1}$ used to generate the obfuscation terms $Y_0, Y_{i,b}$, and $Y_{\ell+1}$ in an appropriate sequence.

[19] Here, we use the fact that the branching programs are single-input. For multi-input programs we do *not* know how to make $\widehat{\alpha}$ independent of x. The rest of the analysis does not require the programs to be single-input.

Proof of Lemma 5.2. For each $x \in \{0,1\}^n$ note that:

$$\begin{aligned} P^{(0)}_{\mathbf{A},x} &= \widetilde{A}_0 \cdot \prod_{i=1}^{\ell} \widetilde{A}_{i,x[\mathsf{inp}(i)]} \cdot \widetilde{A}_{\ell+1} \\ &= A_0 \cdot R_1^{adj} \times \prod_{i=1}^{\ell} \left(\alpha_{i,x_{\mathsf{inp}(i)}} R_i \cdot A_{i,x[\mathsf{inp}(i)]} \cdot R_{i+1}^{adj} \right) \times R_{\ell+1} \cdot A_{\ell+1} \\ &= \rho \widehat{\alpha}_x \mathbf{A}(x) \end{aligned}$$

for $\rho \stackrel{\text{def}}{=} \prod_i \det(R_i)$ (or $\rho I = \prod_i R_i^{adj} R_i$) and $\widehat{\alpha}_x \stackrel{\text{def}}{=} \prod_{i=1}^{\ell} \alpha_{i,x_{\mathsf{inp}(i)}}$.

Also, note that for any $x \in \{0,1\}^n$ we can express $P^{(1)}_{\mathbf{A},x}$ as:

$$\begin{aligned} P^{(1)}_{\mathbf{A},x} &= Z_0 \cdot R_1^{adj} \cdot \prod_{j=1}^{\ell} \widetilde{A}_{j,x[\mathsf{inp}(j)]} \cdot \widetilde{A}_{\ell+1} \\ &\quad + \sum_{i=1}^{\ell} \left(\widetilde{A}_0 \cdot \prod_{j=1}^{i-1} \widetilde{A}_{j,x[\mathsf{inp}(j)]} \cdot \left(\alpha_{i,x[\mathsf{inp}(i)]} R_i \cdot Z_{i,x[\mathsf{inp}(i)]} \cdot R_{i+1}^{adj} \right) \cdot \prod_{j=i+1}^{\ell} \widetilde{A}_{j,x[\mathsf{inp}(j)]} \cdot \widetilde{A}_{\ell+1} \right) \\ &\quad + \widetilde{A}_0 \cdot \prod_{j=1}^{\ell} \widetilde{A}_{j,x[\mathsf{inp}(j)]} \cdot R_{\ell+1} \cdot Z_{\ell+1} \\ &= \rho \widehat{\alpha}_x \left(Z_0 \cdot \prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot A_{\ell+1} \right) \\ &\quad + \rho \widehat{\alpha}_x \sum_{i=1}^{\ell} \left(A_0 \cdot \prod_{j=1}^{i-1} A_{j,x[\mathsf{inp}(j)]} \cdot Z_{i,x[\mathsf{inp}(i)]} \cdot \prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot A_{\ell+1} \right) \\ &\quad + \rho \widehat{\alpha}_x \left(A_0 \cdot \prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot Z_{\ell+1} \right) \end{aligned} \tag{6}$$

Now, define:

$$\mathbf{z}_0 \stackrel{\text{def}}{=} \mathbf{vec}(Z_0) \in \{0,1\}^{d \times 1} \quad , \quad \mathbf{z}_{\ell+1} \stackrel{\text{def}}{=} \mathbf{vec}(Z_{\ell+1}) \in \{0,1\}^{d \times 1},$$

and

$$\mathbf{z}_{i,b} \stackrel{\text{def}}{=} \mathbf{vec}(Z_{i,b}) \in \{0,1\}^{d^2 \times 1}$$

Now, we set

$$\boldsymbol{z}_i = \left[\mathbf{z}_{i,0}^T \mid \mathbf{z}_{i,1}^T \right].$$

And finally set, as

$$\boldsymbol{z} \stackrel{\text{def}}{=} [\boldsymbol{z}_0 \mid \boldsymbol{z}_1 | \ldots | \boldsymbol{z}_\ell \mid \boldsymbol{z}_{\ell+1}] \in \{0,1\}^{1 \times (2\ell+2)d^2}$$

where $\boldsymbol{z}$ consists of all random secret variables involved in $\mathcal{O}(\mathbf{A})$. Next using the property of tensor products (Property 2.3) we can rewrite Eq. 6 as:

$$
\begin{aligned}
P^{(1)}_{\mathbf{A},x} = \mathbf{vec}\left(P^{(1)}_{\mathbf{A},x}\right) &= \rho\widehat{\alpha}_x \mathbf{vec}\left(Z_0 \cdot \prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot A_{\ell+1}\right) \\
&\quad + \rho\widehat{\alpha}_x \sum_{i=1}^{\ell} \mathbf{vec}\left(A_0 \cdot \prod_{j=1}^{i-1} A_{j,x[\mathsf{inp}(j)]} \cdot Z_{i,x[\mathsf{inp}(i)]} \cdot \prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot A_{\ell+1}\right) \\
&\quad + \rho\widehat{\alpha}_x \mathbf{vec}\left(A_0 \cdot \prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot Z_{\ell+1}\right) \\
&= \rho\widehat{\alpha}_x \left(\prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot A_{\ell+1}\right)^T \cdot \mathbf{z}_0 \\
&\quad + \rho\widehat{\alpha}_x \sum_{i=1}^{\ell} \left(A_0 \cdot \prod_{j=1}^{i-1} A_{j,x[\mathsf{inp}(j)]}\right) \otimes \left(\prod_{j=i+1}^{\ell} A_{j,x[\mathsf{inp}(j)]} \cdot A_{\ell+1}\right)^T \cdot \mathbf{z}_{i,x[\mathsf{inp}(i)]} \\
&\quad + \rho\widehat{\alpha}_x \left(A_0 \cdot \prod_{j=1}^{\ell} A_{j,x[\mathsf{inp}(j)]}\right) \cdot \mathbf{z}_{\ell+1} \qquad (7) \\
&= \rho\widehat{\alpha}_x \left(\phi^{(0)}_{\mathbf{A},x} \cdot \mathbf{z}_0 + \sum_{i=1}^{\ell} \phi^{(i)}_{\mathbf{A},x} \cdot \mathbf{z}_{i,x[\mathsf{inp}(i)]} + \phi^{(\ell+1)}_{\mathbf{A},x} \cdot \mathbf{z}_{\ell+1}\right) \\
&= \rho\widehat{\alpha}_x \left(\widetilde{\phi}^{(0)}_{\mathbf{A},x} \cdot \mathbf{z}_0 + \sum_{i=1}^{\ell} \widetilde{\phi}^{(i)}_{\mathbf{A},x} \cdot \boldsymbol{z}_i + \widetilde{\phi}^{(\ell+1)}_{\mathbf{A},x} \cdot \mathbf{z}_{\ell+1}\right) \\
&= \rho\widehat{\alpha}_x (\boldsymbol{\phi}_{\mathbf{A},x} \cdot \boldsymbol{z}^T). \qquad (8)
\end{aligned}
$$

6 Extending the Abstract Attack to GGH13 Multilinear Maps

In this section, we show that an attack in abstract model described in Section 3.1 can be translated to an attack in the GGH13 setting. This part of the attack is heuristic and analogous to some of the previous attacks on GGH13 such as in [GGH13a, MSZ16, CHL+15].

6.1 The GGH13 Scheme: Background

In the GGH13 scheme [GGH13a], the plaintext space is a quotient ring $R/\boldsymbol{g}R$, where R is the ring of integers in a cyclotomic number field and $\boldsymbol{g} \in R$ is a "small prime element." The space of encodings is $R_q = R/qR$ for a large (exponential in the security parameter λ) modulus q. We write $[\cdot]_q$ to denote operations are done in $\mathbb{Z}_q$.

A uniformly random secret $\boldsymbol{z}_1 \ldots \boldsymbol{z}_k \in R_q$ is chosen, and used to encode plaintext values as follows: A plaintext element $\boldsymbol{a} \in R/\boldsymbol{g}R$ is encoded at the level-1 as $\boldsymbol{u} = [\boldsymbol{c}/\boldsymbol{z}]_q$, where the numerator $\boldsymbol{c}$ is a "small" element in the coset of $\boldsymbol{a}$; i.e. $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{g}\boldsymbol{r}$ for a small random term $r \in R$, chosen from an appropriate distribution. We describe the GGH13 and our attack assuming use of "symmetric"

multilinear maps just for simplicity of notation. Note that in our attacks we compute on provided multilinear maps encodings in a prescribed manner. Furthermore, the $\boldsymbol{z}$ always vanish in our attacks. Therefore, the attack immediately generalize to the "asymmetric GGH" setting, with many distinct choices of $\boldsymbol{z}$'s and we continue to use the "symmetric" notation for simplicity.

Addition and subtraction of encodings at the same level is performed by addition in R_q, and outputs an encoding of the sum of the encoded plaintext values at the same level. Multiplication of encodings at levels t_1 and t_2 yields a new level-$t_1 + t_2$ encoding of the product of the corresponding plaintexts.

The level-k encodings of the *zero plaintext,* $0 \in R/\boldsymbol{g}R$, have the form $\boldsymbol{u} = [\boldsymbol{gr}/\boldsymbol{z}^k]_q$. Public parameter of the GGH13 multilinear maps include a public zero-testing parameter $\mathsf{p_{zt}} = [\boldsymbol{hz}^k/\boldsymbol{g}]_q$, for a "somewhat small' element $\boldsymbol{h} \in R$, which is roughly of size $\sqrt{q}$. The zero-test operation involves multiplying $\mathsf{p_{zt}}$ by a level-k encoding $\boldsymbol{u}$, and checking if the result $[\mathsf{p_{zt}} \cdot \boldsymbol{u}]_q$ is much smaller than the modulus q. Note that if $\boldsymbol{u}$ is indeed an encoding of zero then we have that $[\mathsf{p_{zt}} \cdot \boldsymbol{u}]_q = [\boldsymbol{hr}]_q$. If $\boldsymbol{h}, \boldsymbol{r}$, are much smaller than q then we have that this computed value will also be much smaller than q. On the other hand if $\boldsymbol{u} = [\boldsymbol{c}/\boldsymbol{z}^k]_q$ is not an encoding of zero, then we have that $[\mathsf{p_{zt}} \cdot \boldsymbol{u}]_q = [\boldsymbol{c}/\boldsymbol{g}]_q$ will be large.

6.2 Translating the Abstract Attack to GGH13

In this section, we assume that we are provided programs $\mathbf{A}_0$ and $\mathbf{A}_1$, set of inputs X and a vector $\boldsymbol{v}_{\mathsf{ann}}$ such that $\boldsymbol{v}_{\mathsf{ann}} \cdot \Psi_{\mathbf{A}_0,X} \cdot (\boldsymbol{z} \otimes \boldsymbol{z})^T \equiv 0$ and $\boldsymbol{v}_{\mathsf{ann}} \cdot \Psi_{\mathbf{A}_1,X} \cdot (\boldsymbol{z} \otimes \boldsymbol{z})^T \not\equiv 0$. Recall that $\boldsymbol{v}_{\mathsf{ann}}$ is sufficient to complete an attack in the abstract model. Given the above we describe an attack strategy of distinguishing between obfuscations of $\mathbf{A}_0$ and $\mathbf{A}_1$ generated using GGH13 multilinear maps. We do this in two steps. In the first step, we will use the abstract attack to compute an element $\boldsymbol{u}$ whose distribution depends on whether $\mathbf{A}_0$ was used or $\mathbf{A}_1$ was used. We explain this step in this subsection below. The second step that involves efficiently testing the distribution from which $\boldsymbol{u}$ is sampled is described in the next subsection.

Our attack is provided an obfuscation of either $\mathbf{A} \in \{\mathbf{A}_0, \mathbf{A}_1\}$ and it proceeds as follows. It mimics the abstract attack for the pre-zeroing computation queries by computing the values using the provided encodings. Since only "valid" queries were made in the abstract model, therefore the corresponding computation can be done locally. Specifically, for each $x \in X$, we obtain

$$\left[\frac{P_{\mathbf{A},x}}{\boldsymbol{z}^k}\right]_q = \left[\frac{P^{(0)}_{\mathbf{A},x} + \boldsymbol{g}P^{(1)}_{\mathbf{A},x} + \boldsymbol{g}^2(\ldots)}{\boldsymbol{z}^k}\right].$$

Since, $P^{(0)}_{\mathbf{A},x} = 0$, zero-testing on this value yields a value that is unreduced $\mod q$. In particular, zero-test reveals the value:

$$W_{\mathbf{A},x} = \boldsymbol{h}\left(P^{(1)}_{\mathbf{A},x} + \boldsymbol{g}(\ldots)\right).$$

Using these values, we set $\boldsymbol{u} = \boldsymbol{v}_{\mathsf{ann}} \cdot \widetilde{W}_{\mathbf{A},X}$ where

$$\widetilde{W}_{\mathbf{A},X} = \begin{bmatrix} W_{\mathbf{A},x_1} W_{\mathbf{A},\overline{x}_1} \\ \vdots \\ W_{\mathbf{A},x_m} W_{\mathbf{A},\overline{x}_m} \end{bmatrix}$$

Note that if $\mathbf{A} = \mathbf{A}_0$ then we have that $\boldsymbol{u} \in \langle \rho^2 \hat{\alpha} \boldsymbol{h}^2 \boldsymbol{g} \rangle$, where $\rho \stackrel{\text{def}}{=} \prod_i \det(R_i)$ and $\widehat{\alpha} \stackrel{\text{def}}{=} \widehat{\alpha}_x \widehat{\alpha}_{\overline{x}}$ both of which are fixed terms. On the other hand, if $\mathbf{A} = \mathbf{A}_1$ then we have that $\boldsymbol{u} \notin \langle \rho^2 \hat{\alpha} \boldsymbol{h}^2 \boldsymbol{g} \rangle$ with overwhelming probability by Schwartz-Zippel Lemma.

6.3 Completing the Attack for Large Enough Circuits

In order to complete the attack we need to check if $\boldsymbol{u}$ obtained in the previous step is in the ideal $\langle \rho^2 \hat{\alpha} \boldsymbol{h}^2 \boldsymbol{g} \rangle$ or not. Below we describe a method to compute several (heuristically) linearly independent elements is in the ideal $\mathcal{J} = \langle \rho^4 \hat{\alpha}^2 \boldsymbol{h}^4 \boldsymbol{g} \rangle$. Note that if $\boldsymbol{u} \in \langle \rho^2 \hat{\alpha} \boldsymbol{h}^2 \boldsymbol{g} \rangle$ then $\boldsymbol{u}^2 \in \mathcal{J}$ as well. However, since $\boldsymbol{g}$ is prime, if $\boldsymbol{u} \notin \langle \rho^2 \hat{\alpha} \boldsymbol{h}^2 \boldsymbol{g} \rangle$ then $\boldsymbol{u}^2$ will not be in $\mathcal{J}$.

Let $X_1, X_2, \ldots$ be disjoint sets of inputs such that for each i we have that $X_i \cap (X \cup \overline{X}) = \emptyset$, $|X_i| = (2d + 2^\ell d^2)^2$ and that $\forall x \in X_i$ we have that $\mathbf{A}(x) = \mathbf{A}(\overline{x}) = 0$.[20] Since, the number of inputs is 2^n for a large enough circuit we can define any polynomial number of such sets X_i.

Note that for each i, since the number of equations is larger than the number of variables, therefore $\exists \boldsymbol{a}_i, \boldsymbol{b}_i$ such that $\boldsymbol{a}_i \cdot \Psi_{\mathbf{A}_0, X_i} = 0$ and $\boldsymbol{b}_i \cdot \Psi_{\mathbf{A}_1, X_i} = 0$. Therefore, for $\mathbf{A} \in \{\mathbf{A}_0, \mathbf{A}_1\}$ we can conclude that $(\boldsymbol{a}_i \cdot \widetilde{W}_{\mathbf{A}, X_i})(\boldsymbol{b}_i \cdot \widetilde{W}_{\mathbf{A}, X_i}) \in \mathcal{J} = \langle \rho^4 \hat{\alpha}^2 \boldsymbol{h}^4 \boldsymbol{g} \rangle$ where

$$\widetilde{W}_{\mathbf{A}, X_i} = \begin{bmatrix} W_{\mathbf{A}, x_{i,1}} W_{\mathbf{A}, \overline{x}_{i,1}} \\ \vdots \\ W_{\mathbf{A}, x_{i,m}} W_{\mathbf{A}, \overline{x}_{i,m}} \end{bmatrix}$$

and $X_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,m}\}$. Repeating this process for each choice of i we obtain several elements in $\mathcal{J}$. Note that these values are linearly independent except that some of these values (possibly all of them) might actually be in $\mathcal{J}' = \langle \rho^4 \hat{\alpha}^2 \boldsymbol{h}^4 \boldsymbol{g}^2 \rangle$. However, this doesn't affect our attack because $\boldsymbol{u}^2$ is in $\mathcal{J}'$ as well.

7 Example of Partially Inequivalent Circuits

In this section, we show examples of pairs of NC^1 circuits such that the corresponding Barrington-implemented[21] branching programs are partially inequivalent and therefore are subject to the abstract annihilation attacks shown in Section 5. Note that here we extend the notion of partial inequivalence from branching programs to circuits in a natural way. Unless otherwise mentioned, partial inequivaelnce of circuits specifically imply that the corresponding branching programs generated via applying Barrngton's Theorem are partially inequivalent.

7.1 Simple Pairs of Circuits that are Partially Inequivalent

Consider the following pair of circuits (C_0, C_1) each of which implements a boolean function $\{0,1\}^4 \to \{0,1\}$:

$$C_0(x)] \stackrel{\text{def}}{=} (x[1] \wedge 1) \bigwedge (x[2] \wedge 0) \bigwedge (x[3] \wedge 1) \bigwedge (x[4] \wedge 0),$$

$$C_1(x) \stackrel{\text{def}}{=} (x[1] \wedge 0) \bigwedge (x[2] \wedge 0) \bigwedge (x[3] \wedge 0) \bigwedge (x[4] \wedge 0).$$

[20] Since the programs are functionally equivalent we have this condition.

[21] Recall that by Barrington-implementation of a circuit we mean the single-input branching program produced as a result of Barrington Theorem on the circuit. Also we implicitly assume that the branching programs are input-oblivious.

Define the set $X \stackrel{\text{def}}{=} \{0,1\}^4$. Now, we provide an implementation (see Appendix A for more details on the implementation) in Sage[S+16] that evaluates the column spaces of matrices produced via applying a Barrington-implementation to the above circuits. The outcome from the implementation led us to conclude the following claim:

Claim 7.1. *Let $\mathbf{A}_{C_0}, \mathbf{A}_{C_1}$ be the Barrington-Implementation of the circuits C_0, C_1 respectively, then we have that:*

$$\mathsf{colsp}\left(\Psi_{\mathbf{A}_{C_0},X}\right) \neq \mathsf{colsp}\left(\Psi_{\mathbf{A}_{C_1},X}\right).$$

Remark 7.2. *We emphasize that we use branching programs generated with a particular Barrington-implementation that makes a set of specific choices. We remark that Barrington's construction can be implemented in many different ways. However, since in this section we aim to find one concrete example for which the condition of our abstract attack satisfies, we restrict ourselves to this specific program. We refer the reader to Appendix A for the details of our implementation. Throughout this section we refer to this particular Barrington-implementation.*

The circuits presented above are of constant size. Looking ahead, though, they are partially inequivalent and hence (by Theorem 5.1) are susceptible to the abstract attack that does not translate to a real-world attack in GGH13 setting immediately. For that we need to consider larger (albeit NC^1) circuits which we construct next based on the above circuits.

7.2 Larger Pairs of Circuits that are Partially Inequivalent

We construct pairs of NC^1 circuits (in fact, exponentially many of them) that build on the constant-size circuits described in Sec. 7.1.

Consider *any* pair of functionally equivalent NC^1 circuits (D_0, D_1) and an input $x^\star \in \{0,1\}^n$ such that $D_0(x^\star) = D_1(x^\star) = D_0(\overline{x^\star}) = D_1(\overline{x^\star}) = 0$. Now define the circuits E_0, E_1 each of which computes a boolean function $\{0,1\}^{n+4} \to \{0,1\}$ as follows:

$$E_0(y) \stackrel{\text{def}}{=} \neg C_0(x) \wedge D_0(x'),$$

$$E_1(y) \stackrel{\text{def}}{=} \neg C_1(x) \wedge D_1(x')$$

($\neg C$ is the circuit C with output negated) such that for each $y \in \{0,1\}^{n+4}$ we have $y = x \circ x'$ ($\circ$ denotes concatenation) where $x \in \{0,1\}^4$ and $x' \in \{0,1\}^n$. Define the input-sequence $Y \stackrel{\text{def}}{=} \{x \circ x^\star \mid x \in \{0,1\}^4\}$ (consisting of 16 inputs). Then we show the following statement.

Lemma 7.3. *Let $\mathbf{A}_{E_0}, \mathbf{A}_{E_1}$ be the Barrington-implementations of E_0, E_1 respectively, then we have that:*

$$\mathsf{colsp}\left(\Psi_{\mathbf{A}_{E_0},Y}\right) \neq \mathsf{colsp}\left(\Psi_{\mathbf{A}_{E_1},Y}\right),$$

Proof. As a first step, similar to Claim 7.1 we also verify the following claim via our Sage-implementations (c.f. Appendix A for more details on the implementation).

Claim 7.4. *Let $\mathbf{A}_{\neg C_0 \wedge 0}, \mathbf{A}_{\neg C_1 \wedge 0}$ be the Barrington-implementations of the circuits $\neg C_0 \wedge 0, \neg C_1 \wedge 0$ respectively, then we have that:*

$$\mathsf{colsp}\left(\Psi_{\mathbf{A}_{\neg C_0 \wedge 0},X}\right) \neq \mathsf{colsp}\left(\Psi_{\mathbf{A}_{\neg C_1 \wedge 0},X}\right)$$

Now, recall (Def. 2.4) that any branching program $\mathbf{A}$ has the following representation:

$$\mathbf{A} = (\mathsf{inp}, A_0, \{A_{i,b}\}_{i\in[\ell],b\in\{0,1\}}, A_{\ell+1}).$$

Let us call the "core" of $\mathbf{A}$ as: $\mathbf{A}' \stackrel{\text{def}}{=} \{A_{1,b}, \ldots, A_{\ell,b}\}_{b\in\{0,1\}}$.[22] For any such $\mathbf{A}'$ we define the inverse as $\mathbf{A}'^{-1} \stackrel{\text{def}}{=} \{A^{-1}_{\ell,b}, A^{-1}_{\ell-1,b}, \ldots, A^{-1}_{1,b}\}_{b\in\{0,1\}}$. Furthermore, for any permutation matrix $\boldsymbol{\rho} \in S_5$ (recall that Barrington-implementation works with matrices in the symmetric group S_5). we define an operation on $\mathbf{A}'$:

$$\boldsymbol{\rho}(\mathbf{A}')\boldsymbol{\rho}^{-1} = \left\{(\boldsymbol{\rho}\cdot A_{1,b}), \{A_{i,b}\}_{i\in[\ell]}, (A_{\ell,b}\cdot\boldsymbol{\rho}^{-1})\right\}_{b\in\{0,1\}}$$

Now recall that using the construction from Barrington's Theorem with the above notations we can write for any choice of $E \in (E_0, E_1)$ (where $E = C \wedge D$)

$$\underbrace{\mathbf{A}'_{\neg C\wedge D}}_{\Phi_{\mathbf{A}_{\neg C\wedge D},Y}} = \underbrace{(\boldsymbol{\rho}(\mathbf{A}'_{\neg C})\boldsymbol{\rho}^{-1})}_{M_0} \circ \underbrace{(\boldsymbol{\varrho}(\mathbf{A}'_D)\boldsymbol{\varrho}^{-1})}_{M_1} \circ \underbrace{(\boldsymbol{\rho}(\mathbf{A}'_{\neg C})^{-1}\boldsymbol{\rho}^{-1})}_{M_2} \circ \underbrace{(\boldsymbol{\varrho}(\mathbf{A}'_D)^{-1}\boldsymbol{\varrho}^{-1})}_{M_3} \tag{9}$$

where $\boldsymbol{\rho}, \boldsymbol{\varrho} \in S_5$ are specific to the Barrington-implementation (see Appendix A for the exact values we used) and fixed for a particular implementation. Now we can split the linear partial matrix of $\mathbf{A}_{\neg C\wedge D}$ into four parts:

$$\Phi_{\mathbf{A}_{\neg C\wedge D},Y} \stackrel{\text{def}}{=} [M_0 \mid M_1 \mid M_2 \mid M_3]$$

where each M_i is corresponding to a part of the core $\mathbf{A}'_{C\wedge D}$ as shown in Eq. 9. However, since we have $D_{x^\star} = 0$ for all $y = x \circ x^\star$ in Y and for any two inputs $y_1, y_2 \in Y$ the sub-input to the circuit D (corresponding to parts M_1 and M_3) is same (equal to $x^\star$) clearly when i is in the range of M_1 or M_3 we get that:

$$\phi^{(i)}_{\mathbf{A}_{\neg C\wedge D},y_1} = \phi^{(i)}_{\mathbf{A}_{\neg C\wedge D},y_2}.$$

which implies that $M_1, M_3 \in \mathcal{T}$ where $\mathcal{T}$ is a family of all "trivial matrices" with columns which are either all 0 or all 1 as follows:

$$\begin{bmatrix} \cdots & 1 \cdots & 0 & \cdots \\ \cdots & 1 \cdots & 0 & \cdots \\ \vdots & \ddots & \vdots & \vdots \\ \cdots & 1 \cdots & 0 & \cdots \end{bmatrix}$$

Again, using Barrington's Theorem for the circuit $\neg C \wedge 0$ we have that:

$$\underbrace{\mathbf{A}'_{\neg C\wedge 0}}_{\Phi_{\mathbf{A}_{\neg C\wedge 0},X}} = \underbrace{(\boldsymbol{\rho}(\mathbf{A}'_{\neg C})\boldsymbol{\rho}^{-1})}_{N_0} \circ \underbrace{(\boldsymbol{\varrho}(ID)\boldsymbol{\varrho}^{-1})}_{N_1} \circ \underbrace{(\boldsymbol{\rho}(\mathbf{A}'_{\neg C})^{-1}\boldsymbol{\rho}^{-1})}_{N_2} \circ \underbrace{(\boldsymbol{\varrho}(ID)^{-1}\boldsymbol{\varrho}^{-1})}_{N_3} \tag{10}$$

for $X = \{0,1\}^4$ where we again have $N_1, N_3 \in \mathcal{T}$ that follows using similar arguments.

Moreover, using again the fact that for any $y = x \circ x^\star$ in Y we have that $D(x^\star) = 0$, the core of D would always evaluates to ID on any choice of $y \in Y$. Hence when i lies in the range of M_0, M_2 the partial vectors $\phi^{(i)}_{\mathbf{A}_{\neg C\wedge D},Y}$ are independent of the part of the program corresponding to

[22] The order of the matrices are taken into account here and the evaluation of branching program depends on that. So, essentially we abuse notations of sets to denote an ordered tuple here. Unless otherwise mentions we assume that the set $\{A_{i,b}\}_{i\in[\ell],b\in\{0,1\}}$ is ordered as $\{A_{1,b}, \ldots, A_{\ell,b}\}_{b\in\{0,1\}}$

the ranges of M_1, M_3. Therefore, we can conclude that the i-th partial vectors corresponding to ranges M_0, M_2 would be equal to the i-th partial vectors corresponding to ranges N_0, N_2. Hence,

$$M_0 = N_0 \quad \text{and} \quad M_2 = N_2$$

On the other hand, via exactly the same analysis for the inputs $\overline{Y} = \{\overline{y} \mid \overline{x} \circ \overline{x^\star}\}_{x\in\{0,1\}^4}$ we have that:

$$\Phi_{\mathbf{A}_{\neg C\wedge D},\overline{Y}} = [\overline{M}_0 \mid \overline{M}_1 \mid \overline{M}_2 \mid \overline{M}_3]$$
$$\Phi_{\mathbf{A}_{\neg C\wedge 0},\overline{X}} = [\overline{N}_0 \mid \overline{N}_1 \mid \overline{N}_2 \mid \overline{N}_3]$$

where $\overline{M_1}, \overline{M}_3, \overline{N}_1, \overline{N}_3 \in \mathcal{T}$ and

$$\overline{M}_0 = \overline{N}_0 \quad \text{and} \quad \overline{M}_2 = \overline{N}_2.$$

Hence we conclude:

$$\begin{aligned}\mathsf{colsp}\left(\Psi_{\mathbf{A}_{\neg C\wedge D},Y}\right) &= \mathsf{colsp}\left(\left[\mathbf{\Pi}_{i,j}[M_i \boxtimes \overline{M}_j]\right] + \left[\mathbf{\Pi}_{i,j}[\overline{M}_i \boxtimes M_j]\right]\right) &(11)\\ &= \mathsf{colsp}\left(\left[\tau \mid \mathbf{\Pi}_{i,j\in\{0,2\}}[M_i \boxtimes \overline{M}_j]\right] + \left[\tau' \mid \mathbf{\Pi}_{i,j\in\{0,2\}}[\overline{M}_i \boxtimes M_j]\right]\right) &(12)\end{aligned}$$

for some $\tau, \tau' \in \mathcal{T}$. Note that, in the above equations the first step follows from Fact 2.5. In the second step we first observe that for $\tau, \tau' \in \mathcal{T}$ and any matrix $M_i, \overline{M}_i$ we have that $\mathsf{colsp}\left(\tau \boxtimes M_i\right) = \mathsf{colsp}\left(M_i\right)$ (resp. $\mathsf{colsp}\left(\tau \boxtimes \overline{M}_i\right) = \mathsf{colsp}\left(\overline{M}_i\right)$). Also applying Lemma 4.3 it is straightforward to verify that each of the matrices $\{M_i, \overline{M}_i\}_{i\in[3]\cup\{0\}}$ has the same number of 1's in each row. Hence, then we use Lemma 2.13 to obtain the final expression. Similarly we get:

$$\begin{aligned}\mathsf{colsp}\left(\Psi_{\mathbf{A}_{\neg C\wedge 0},X}\right) &= \mathsf{colsp}\left(\left[\mathbf{\Pi}_{i,j}[N_i \boxtimes \overline{N}_j]\right] + \left[\mathbf{\Pi}_{i,j}[\overline{N}_i \boxtimes N_j]\right]\right) &(13)\\ &= \mathsf{colsp}\left(\left[\sigma \mid \mathbf{\Pi}_{i,j\in\{0,2\}}[N_i \boxtimes \overline{N}_j]\right] + \left[\sigma' \mid \mathbf{\Pi}_{i,j\in\{0,2\}}[\overline{N}_i \boxtimes N_j]\right]\right) &(14)\end{aligned}$$

for some $\sigma, \sigma' \in \mathcal{T}$.

Using the facts, $M_k = N_k$ and $\overline{M}_k = \overline{N}_k$ for $k \in \{0, 2\}$ from Eq. 12 and Eq. 14 we get:

$$\mathsf{colsp}\left(\Psi_{\mathbf{A}_{\neg C\wedge D},Y}\right) = \mathsf{colsp}\left(\Psi_{\mathbf{A}_{\neg C\wedge 0},X}\right).$$

Now combining this equation with Claim 7.4 the lemma follows.

□

7.3 Universal Circuit Leading to Partially Inequivalent Branching Programs

In this section we present constructions of (NC^1) universal circuits that, when compiled with two arbitrary distinct (NC^1) but functionally equivalent circuits as inputs, then the obfuscations of the Barrington-implementation of the compiled circuits are distinguishable by the abstract attack.

For any circuit C, its description is denoted by a bit-string, abusing notation slightly we use the same symbol C to represent the description of C.

Definition 7.5 (Universal Circuits)**.** *An universal circuit U is a boolean circuit that computes a function $\{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}$ which takes two inputs, a λ-bit circuit-description of some boolean circuit $C : \{0,1\}^n \to \{0,1\}$ and a n-bit input x to output $C(x)$. We denote $U(C,x) \stackrel{\text{def}}{=} C(x)$. We also denote the compiled universal circuit with the description of C hard-coded into it by $U[C]$.*

Theorem 7.6. *There exists a family of NC^1 universal circuits $\mathcal{U} = \{U_1, U_2, \ldots, U_v\}$ of size $v = O(poly(\lambda))$ such that: given two arbitrary functionally equivalent NC^1 circuits G_0, G_1 that computes arbitrary boolean function $\{0,1\}^n \to \{0,1\}$ satisfying (i) $|G_0| = |G_1| = v$ and (ii) there exists an input $x^\star$ such that $G_0(x^\star) = G_1(x^\star) = G_0(\overline{x^\star}) = G_1(\overline{x^\star}) = 0$; then for at least one $i \in [v]$ the Barrington-implementations of the circuits $U_i[G_0]$ and $U_i[G_1]$ are partially inequivalent.*

Proof. Our construction of the family $\mathcal{U}$ is similar to the construction of circuits E_0, E_1 constructed in Section 7.1

Construction of the family $\mathcal{U}$. Given a universal circuit U' we construct a family of NC^1 universal circuits $\mathcal{U} = \{U_1, \ldots, U_v\}$ where each U_i is described as follows for any circuit $G : \{0,1\}^n \to \{0,1\}$ we define $U_i[G]$

$$U_i[G](y,x) = C(y) \wedge U_i'(G,x) \quad \text{where} \quad C = (y[1] \wedge G[i]) \bigwedge (y[2] \wedge 0) \bigwedge (y[3] \wedge G[i]) \bigwedge (y[4] \wedge 0),$$

as the circuit from $\{0,1\}^{n+4} \to \{0,1\}$. Since the given circuits must have different descriptions, they differ by at least one bit location, say i^{th} location. Clearly, assuming that $G_0[i] = 1$ and $G_1[i] = 0$ the circuit $U_i[G_b]$ is the same as the circuit E_b as described in Sec. 7.2. Hence applying Lemma 7.3 we conclude that, if $G_0[i] = 1$ and $G_1[i] = 0$ then,

$$\mathsf{colsp}\left(\Psi_{\mathbf{A}_{U_i[G_0]},X}\right) \neq \mathsf{colsp}\left(\Psi_{\mathbf{A}_{U_i[G_1]},X}\right),$$

where $X = \{x \circ x^\star \mid x \in \{0,1\}^4\}$.

□

References

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

[ABD16] Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 153–178. Springer, Heidelberg, August 2016.

[AGIS14] Prabhanjan Vijendra Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai. Optimizing obfuscation: Avoiding Barrington's theorem. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 646–658. ACM Press, November 2014.

[AJN+16] Prabhanjan Ananth, Aayush Jain, Moni Naor, Amit Sahai, and Eylon Yogev. Universal constructions and robust combiners for indistinguishability obfuscation and witness encryption. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 491–520. Springer, Heidelberg, August 2016.

[App14] Benny Applebaum. Bootstrapping obfuscators via fast pseudorandom functions. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 162–172. Springer, Heidelberg, December 2014.

[Bar86] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc^1. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 1–5, 1986.

[Bar89] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc^1. *J. Comput. Syst. Sci.*, 38(1):150–164, 1989.

[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 Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

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

[BGL+15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 439–448. ACM Press, June 2015.

[BMSZ16a] Saikrishna Badrinarayanan, Eric Miles, Amit Sahai, and Mark Zhandry. Post-zeroizing obfuscation: New mathematical tools, and the case of evasive circuits. In *EUROCRYPT*, 2016.

[BMSZ16b] Saikrishna Badrinarayanan, Eric Miles, Amit Sahai, and Mark Zhandry. Post-zeroizing obfuscation: New mathematical tools, and the case of evasive circuits. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 764–791. Springer, Heidelberg, May 2016.

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

[CDPR16] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 559–585. Springer, Heidelberg, May 2016.

[CFL+16] Jung Hee Cheon, Pierre-Alain Fouque, Changmin Lee, Brice Minaud, and Hansol Ryu. Cryptanalysis of the new CLT multilinear map over the integers. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 509–536. Springer, Heidelberg, May 2016.

[CGH+15a] Jean-Sébastien Coron, Craig Gentry, Shai Halevi, Tancrède Lepoint, Hemanta K. Maji, Eric Miles, Mariana Raykova, Amit Sahai, and Mehdi Tibouchi. Zeroizing without low-level zeroes: New MMAP attacks and their limitations. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 247–266, 2015.

[CGH+15b] Jean-Sébastien Coron, Craig Gentry, Shai Halevi, Tancrède Lepoint, Hemanta K. Maji, Eric Miles, Mariana Raykova, Amit Sahai, and Mehdi Tibouchi. Zeroizing without low-level zeroes: New MMAP attacks and their limitations. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 247–266. Springer, Heidelberg, August 2015.

[CHL+15] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 3–12. Springer, Heidelberg, April 2015.

[CJL16] Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An algorithm for NTRU problems and cryptanalysis of the GGH multilinear map without an encoding of zero. *IACR Cryptology ePrint Archive*, 2016:139, 2016.

[CLLT16] Jean-Sébastien Coron, Moon Sung Lee, Tancrède Lepoint, and Mehdi Tibouchi. Cryptanalysis of GGH15 multilinear maps. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 607–628. Springer, Heidelberg, August 2016.

[CLT13] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 476–493. Springer, Heidelberg, August 2013.

[DGG+16] Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Pratyay Mukherjee. Obfuscation from low noise multilinear maps. Cryptology ePrint Archive, Report 2016/599, 2016. http://eprint.iacr.org/2016/599.

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

[GGH+13b] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

[GGH15] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 498–527. Springer, Heidelberg, March 2015.

[GIS+10] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 308–326. Springer, Heidelberg, February 2010.

[GMM+16] Sanjam Garg, Eric Miles, Pratyay Mukherjee, Amit Sahai, Akshayaram Srinivasan, and Mark Zhandry. Secure obfuscation in a weak multilinear map model. In *TCC 2016-B*, 2016.

[HJ16] Yupu Hu and Huiwen Jia. Cryptanalysis of GGH map. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 537–565. Springer, Heidelberg, May 2016.

[Kay09] N. Kayal. The complexity of the annihilating polynomial. In *Computational Complexity, 2009. CCC '09. 24th Annual IEEE Conference on*, pages 184–193, July 2009.

[MSW14] Eric Miles, Amit Sahai, and Mor Weiss. Protecting obfuscation against arithmetic attacks. Cryptology ePrint Archive, Report 2014/878, 2014. http://eprint.iacr.org/2014/878.

[MSZ16] Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 629–658. Springer, Heidelberg, August 2016.

[Ogu16] Arthur Ogus. Row equivalence of matrices (lecture notes). https://math.berkeley.edu/~ogus/old/Math_110-07/Supplements/week6.pdf, 2016. Online; accessed 30 September 2016.

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

[S+16] W. A. Stein et al. *Sage Mathematics Software (Version 7.3)*. The Sage Development Team, 2016. http://www.sagemath.org.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *46th ACM STOC*, pages 475–484. ACM Press, May / June 2014.

A Some details on our implementation

In this section we provide details on our Barrington-implementation and discuss some optimizations in the Sage-code.

Overview of Barrington's Programs [Bar89]. Barrington's construction works over permutations in the symmetric group S_5. We assume that permutations are represented as matrices for all practical purpose. A Barrington-implementation specifies permutations $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\rho}, \boldsymbol{\varrho} \in S_5$ such that the following holds:

- $\boldsymbol{\alpha}, \boldsymbol{\beta}$ are 5-cycles.
- $\boldsymbol{\gamma} = \boldsymbol{\alpha}\boldsymbol{\beta}\boldsymbol{\alpha}^{-1}\boldsymbol{\beta}^{-1}$ and one can verify that $\boldsymbol{\gamma}$ is also a cycle.

- $\boldsymbol{\rho} \cdot \boldsymbol{\gamma} \cdot \boldsymbol{\rho}^{-1} = \boldsymbol{\alpha}$.
- $\boldsymbol{\varrho} \cdot \boldsymbol{\gamma} \cdot \boldsymbol{\varrho}^{-1} = \boldsymbol{\beta}$.

We define some syntaxes for branching programs. Some of them are redefinitions from Sec. 7.2 (provided in the proof of Lemma 7.3 as it uses some details on Barrington-implementations).

Core of a Branching Program. Recall (Def. 2.4) that any branching program $\mathbf{A}$ has the following representation:

$$\mathbf{A} = (\mathsf{inp}, A_0, \{A_{i,b}\}_{i\in[\ell],b\in\{0,1\}}, A_{\ell+1}).$$

Let us call the "core" of $\mathbf{A}$ as: $\mathbf{A}' \stackrel{\text{def}}{=} \{A_{1,b}, \ldots, A_{\ell,b}\}_{b\in\{0,1\}}$.[23] For any such $\mathbf{A}'$ we define the inverse as $\mathbf{A}'^{-1} \stackrel{\text{def}}{=} \{A^{-1}_{\ell,b}, A^{-1}_{\ell-1,b}, \ldots, A^{-1}_{1,b}\}_{b\in\{0,1\}}$. Furthermore, for any permutation matrix $\boldsymbol{\rho} \in S_5$ we define an operation on $\mathbf{A}'$:

$$\boldsymbol{\rho}(\mathbf{A}')\boldsymbol{\rho}^{-1} \stackrel{\text{def}}{=} \left\{(\boldsymbol{\rho} \cdot A_{1,b}), \{A_{i,b}\}_{i\in[\ell]}, (A_{\ell,b} \cdot \boldsymbol{\rho}^{-1})\right\}_{b\in\{0,1\}}$$

$\boldsymbol{\gamma}$-computation. Any branching program $\mathbf{A}_C = (\mathsf{inp}, A_0, \{A_{i,b}\}_{i\in[\ell],b\in\{0,1\}}, A_{\ell+1})$ is said to be $\boldsymbol{\gamma}$-computes a boolean circuit C if the following holds:

$$\prod_{i=1}^{\ell} A_{i,x[\mathsf{inp}(i)]} = \begin{cases} \boldsymbol{\gamma} & when \;\; C(x) = 1 \\ ID^{5\times 5} & when \;\; C(x) = 0 \end{cases}$$

If $\mathbf{A}_{C_0}, \mathbf{A}_{C_1}$ $\boldsymbol{\gamma}$-computes C_0, C_1 then one can construct $\mathbf{A}_{C_0\wedge C_1}$ that $\boldsymbol{\gamma}$-computes $C = C_0 \wedge C_1$ as follows:

$$\mathbf{A}'_{C_0\wedge C_1} = (\boldsymbol{\rho}(\mathbf{A}'_{C_0})\boldsymbol{\rho}^{-1}) \circ (\boldsymbol{\varrho}(\mathbf{A}'_{C_1})\boldsymbol{\varrho}^{-1}) \circ (\boldsymbol{\rho}(\mathbf{A}'_{C_0})^{-1}\boldsymbol{\rho}^{-1}) \circ (\boldsymbol{\varrho}(\mathbf{A}'_{C_1})^{-1}\boldsymbol{\varrho}^{-1})$$

and with the same bookends.[24]

Let us also define the operation $(\mathbf{A}') \cdot \boldsymbol{\gamma}$ as $(\mathbf{A}') \cdot \boldsymbol{\gamma} \stackrel{\text{def}}{=} \{A_{1,b}, \ldots, A_{\ell_b} \cdot \boldsymbol{\gamma}\}_{b\in\{0,1\}}$ that has the final pairs right-multiplied with $\boldsymbol{\gamma}$. Then one can construct another branching program $\mathbf{A}_{\neg C}$ that $\boldsymbol{\gamma}$-computes the circuit $\neg C$ as follows:

$$\mathbf{A}'_{\neg C} = (\mathbf{A}'^{-1}_{C}) \cdot \boldsymbol{\gamma}$$

Since any boolean circuit can be converted to a circuit containing only NOT ($\neg$) and AND ($\wedge$) gates Barrington's theorem [Bar86] follows.

Our Barrington-implementation. In our implementations the branching programs are single-input and input-oblivious. We stress that the input-obliviousness comes automatically from our choice of circuits.

We choose the following permutations for our implementation:

[23] The order of the matrices are taken into account here and the evaluation of branching program depends on that. So, essentially we abuse notations of sets to denote an ordered tuple here. Unless otherwise mentioned we assume that the set $\{A_{i,b}\}_{i\in[\ell],b\in\{0,1\}}$ is ordered as $\{A_{1,b}, \ldots, A_{\ell,b}\}_{b\in\{0,1\}}$

[24] Our input function is a fixed one and designed as suggested by Barrington's Theorem. Namely to compute a program of size 4 on 2-bit input, $\boldsymbol{\alpha}\boldsymbol{\beta}\boldsymbol{\alpha}^{-1}\boldsymbol{\beta}^{-1}$ we use input function $\mathsf{inp} = (1 \to 1, 1 \to 2, 3 \to 1, 4 \to 2)$, that is the first position of the program reads the first bit, the fourth position the second and so on. Similarly for AND operation the input-functions can be extended with adjusted indexes. For more details we refer to Barrington's result [Bar86].

$$\boldsymbol{\alpha} \stackrel{\text{def}}{=} (1 \to 2 \to 3 \to 4 \to 5) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\boldsymbol{\beta} \stackrel{\text{def}}{=} (1 \to 3 \to 5 \to 4 \to 2) = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\boldsymbol{\gamma} \stackrel{\text{def}}{=} (1 \to 3 \to 2 \to 5 \to 4) = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\boldsymbol{\rho} \stackrel{\text{def}}{=} (\alpha \to \gamma) = (1 \to 1, 2 \to 3, 3 \to 2, 4 \to 5, 5 \to 4) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\boldsymbol{\varrho} \stackrel{\text{def}}{=} (\beta \to \gamma) = (1 \to 1, 3 \to 3, 5 \to 2, 4 \to 5, 2 \to 4) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

We fix the bookends to:

$$A_0 \stackrel{\text{def}}{=} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad A_{\ell+1} \stackrel{\text{def}}{=} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Source Code and Experimental Set-Up. We provide an implementation in Sage [S+16]. The sage-executable file named implementations.sagews and a corresponding pdf file (implementations.pdf) of our source-code can be found at `https://people.eecs.berkeley.edu/~pratyay85/Implementations.zip`. The code can be run on the SageMath cloud server (`https://sagemath.cloud/`) The approximate performance for the 3 circuits on the SageMath cloud are given below:

Circuit	Approx time	Approx memory
C	3100 sec ($\sim$ 55 minutes)	4 GB
$\neg C$	3100 sec ($\sim$ 55 minutes)	4 GB
$\neg C \wedge 0$	33400 sec ($\sim$ 10 hours)	9 GB

Optimizations. Our source-code is not low-level optimized. However, to run the quadratic attack in practical time we required some algorithmic optimization in order to get the program terminated in reasonable time. In particular, since the number of columns for the quadratic partial matrix, $\Psi_{\mathbf{A},X}$ becomes squared compared to number of columns in the linear matrices $\Phi_{\mathbf{A},X}, \Phi_{\mathbf{A},\overline{X}}$, even for the case of the simplest circuits ($\mathbf{A}_C$ or $\mathbf{A}_{\neg C}$) the estimated time to compute directly $\Psi_{\mathbf{A},X}$ as $(\Phi_{\mathbf{A},X} \boxtimes \Phi_{\mathbf{A},\overline{X}} + \Phi_{\mathbf{A},\overline{X}} \boxtimes \Phi_{\mathbf{A},X})$ becomes huge. Instead, we first remove the columns that are all-zero in both $\Phi_{\mathbf{A},X}, \Phi_{\mathbf{A},\overline{X}}$ since the corresponding random variables $\mathbf{z}_{i,b}$ appear in neither of the linear partial matrices. Then we observe that, even after performing that removal, there are many columns that are all-zero in exactly one of $\Phi_{\mathbf{A},X}, \Phi_{\mathbf{A},\overline{X}}$. Hence we first collect those that appear in both and then those appear in one of them. Let us call these three parts $M_X, M_{\overline{X}}$ and $M_{X,\overline{X}}$. Then we have:

$$\Phi^{\star}_{\mathbf{A},X} = [M_X \mid M_{X,\overline{X}}] \quad \Phi^{\star}_{\mathbf{A},\overline{X}} = [M_{\overline{X}} \mid M_{X,\overline{X}}]$$

where $\Phi^{\star}_{\mathbf{A},X}$ (resp. $\Phi^{\star}_{\mathbf{A},\overline{X}}$) is the same as $\Phi_{\mathbf{A},X}$ (resp. $\Phi_{\mathbf{A},\overline{X}}$) but without some all 0 columns (those appear in none).

Then we compute

$$N = M_X \boxtimes \Phi_{\mathbf{A},\overline{X}} + M_{\overline{X}} \boxtimes \Phi_{\mathbf{A},X} + \Phi_{\mathbf{A},X} \boxtimes \Phi_{\mathbf{A},\overline{X}} + \Phi_{\mathbf{A},X} \boxtimes \Phi_{\mathbf{A},\overline{X}} \overset{\text{per}}{=} \Psi_{\mathbf{A},X}$$

by combining Fact 2.5 with the above observation. This reduced the number of row-wise tensor product by at least 2 (even after removing the all-zero columns) as we are not computing tensor products from both directions for the matrices containing columns that appear only once.