Quantum Attacks against Indistinguishablility Obfuscators Proved Secure in the Weak Multilinear Map Model

Alice Pellet-Mary

Univ Lyon, CNRS, ENS de Lyon, Inria, UCBL, LIP, Lyon, France.
alice.pellet___mary@ens-lyon.fr

Abstract. We present a quantum polynomial time attack against the GMMSSZ branching program obfuscator of Garg et al. (TCC'16), when instantiated with the GGH13 multilinear map of Garg et al. (EUROCRYPT'13). This candidate obfuscator was proved secure in the weak multilinear map model introduced by Miles et al. (CRYPTO'16).
Our attack uses the short principal ideal solver of Cramer et al. (EUROCRYPT'16), to recover a secret element of the GGH13 multilinear map in quantum polynomial time. We then use this secret element to mount a (classical) polynomial time mixed-input attack against the GMMSSZ obfuscator. The main result of this article can hence be seen as a classical reduction from the security of the GMMSSZ obfuscator to the short principal ideal problem (the quantum setting is then only used to solve this problem in polynomial time).
As an additional contribution, we explain how the same ideas can be adapted to mount a quantum polynomial time attack against the DGGMM obfuscator of Döttling et al. (ePrint 2016), which was also proved secure in the weak multilinear map model.

1 Introduction

An obfuscator is a cryptographic primitive that should enable a user to compute a function, without revealing anything about it, except its input-output behaviour. Unfortunately, such a security notion for obfuscators, called Virtual Black Box (or VBB) security, has been shown to be impossible to achieve for all circuits [7]. To circumvent this impossibility result, two directions have been explored. The first direction is to build a VBB obfuscator for a restricted class of functions. Recently, the authors of [36] and [25] managed to prove VBB security of their obfuscator, for the restricted class of compute-and-compare functions,[1] under the LWE assumption. The second direction is to consider weaker security notions, and try to build obfuscators for all circuits under these weaker security notions. In addition to their impossibility result, the authors of [7] proposed such a weaker security notion, called indistinguishablility obfuscation (or iO).

[1] A compute-and-compare function CC[f,α] on input x outputs 1 if $f(x) = \alpha$ and 0 otherwise.

Indistinguishability obfuscation requires that it should be hard to distinguish between the obfuscation of two equivalent circuits, i.e., circuits that compute the same function. Even if iO security is weaker than VBB security, achieving iO for all circuits would have a lot of applications (see, e.g., [22, 34]). The first candidate obfuscator for iO security was proposed in 2013 by Garg, Gentry, Halevi, Raykova, Sahai and Waters [22], based on the GGH13 approximate multilinear map [21]. They showed that iO for the class of polynomial-size branching programs[2] could be bootstrapped to iO for all polynomial-size circuits,[3] and they then described a candidate iO obfuscator for polynomial-size branching programs (without a security proof). Since 2013, numerous candidate obfuscators for polynomial-size branching programs have been proposed, all relying on one of the three candidate cryptographic multilinear map constructions [17, 21, 24].[4] However, none of these candidate obfuscators could be proven secure under classical hardness assumptions.

The main security weakness of these candidate obfuscators stems from the underlying candidate multilinear maps. Indeed, all candidate multilinear maps have been shown to suffer from so-called zeroizing attacks [15, 26], and these zeroizing attacks and their generalizations have made it difficult to design potentially secure iO obfuscators. In the following, we will instantiate all the obfuscators with the GGH13 [21] multilinear map,[5] as our attack exploits a weakness of this specific multilinear map.

In order to improve security confidence, recent obfuscator constructions carefully instantiate the underlying multilinear map (to try to avoid zeroizing attacks) and prove VBB security of their obfuscator in some idealised model. First, the authors of [2, 6, 12] proved VBB security of their obfuscators in the so-called ideal graded encoding model, introduced in [11]. But zeroizing attacks against multilinear maps and the resulting annihilation attacks against obfuscators [3, 14, 31] showed that this model was not adapted to capture potential attacks against obfuscators. Another model was then proposed in [31]: the weak multilinear map model. This model captures all the attacks mentioned above, and two candidate obfuscators were proved secure in this model [19, 23].

Previous work. The annihilation attack of Miles, Sahai and Zhandry [31] already impacted many obfuscators: [2,5,6,12,30,32]. One limitation of this attack is that it is captured by the weak multilinear map model and so cannot apply against the recent obfuscators of [19, 23]. A formalisation and generalisation of this attack was then proposed by [3]. This attack enables to distinguish a larger class of circuits than the one of [31], but applies to the same candidate obfuscators.

[2] See Section 2.3 for the definition of a matrix branching program.

[3] The proof relies on Barrington's theorem [8], and on a bootstrapping procedure enabled by fully homomorphic encryption.

[4] The GGH15 multilinear map is a restricted multilinear map that cannot be used for all obfuscator constructions.

[5] Some obfuscators, like [19] are specifically designed to work with the GGH13 multilinear map. Some others can be instantiated with either GGH13 or CLT13 multilinear map. For those, we only consider the GGH13 instantiation.

Moreover, it only works for single-input branching programs. In a parallel work, Chen, Gentry and Halevi [14], proposed an attack against the original obfuscator of [22], and a quantum attack against the GGH15 construction [24], that were both unbroken so far. These attacks rely on specific branching programs, namely input partitionable branching programs. Since then, Fernando, Rasmussen and Sahai [20] proposed a technique to transform any branching program into an equivalent branching program which is not input partitionable. This transformation can be used either with the GGH13 map or with the CLT map. Hence, using the [22] obfuscator combined with the technique of [20] prevents the attack of [14].

Our contribution. In this work, we propose quantum polynomial time attacks against the branching program obfuscators of [19,23], when instantiated with the GGH13 multilinear map. These candidate obfuscators were not broken yet, and were proven secure in the weak multilinear map model (the current strongest ideal model for obfuscators). As a secondary contribution, our attack also applies to the obfuscators of [2,5,6,30,32], which were already broken in classical polynomial time by [31]. Our attack is still interesting for these obfuscators, as it uses different techniques than those of [31], and in particular, techniques that are not captured by the weak multilinear map model. Note that our attack does not work against the obfuscator of [12], while [31] does. Finally, as a last contribution, our attack also applies to the circuit obfuscators of [4,37], when instantiated with the GGH13 multilinear map.[6] Overall, we prove the following theorem (informally stated for the moment).

Theorem 1 (Informal, heuristic). *Let $\mathcal{O}$ be any of the branching program obfuscators in [2,5,6,23,30,32], on single or dual input branching programs (respectively, let $\mathcal{O}$ be any of the circuit obfuscators in [4,19,37]), instantiated with the GGH13 multilinear map [21]. There exist two explicit equivalent branching programs (respectively, two equivalent circuits) $\boldsymbol{A}$ and $\boldsymbol{A}'$ such that $\mathcal{O}(\boldsymbol{A})$ and $\mathcal{O}(\boldsymbol{A}')$ can be distinguished in quantum polynomial time, under some conjecture and heuristic (see Theorem 3 for a formal statement).*

We note that the only part of our attack which is quantum is the principal ideal solver of Biasse and Song [10]. All the other steps of our attack are classical. Hence, our attack can also be viewed as a (classical) reduction from the iO security of the candidate obfuscators mentioned in Theorem 1 to the principal ideal problem. One might then want to use the classical sub-exponential principal ideal solver of Biasse, Espitau, Fouque, Gélin and Kirchner [9] to obtain a classical sub-exponential attack against the above obfuscators. However, the dimension of the cyclotomic ring used in current instantiations on the GGH multilinear map is chosen to be at least λ^2 where λ is the security parameter. This is done to thwart the attacks of [1,16,27] over the GGH13 multilinear map,

[6] These obfuscators need composite-order multilinear maps, and hence were originally instantiated with the CLT multilinear map. However, as observed in [19], the GGH13 multilinear map can also be used with composite-order.

but it also means that the classical variant of the attack described in this article is exponential in the security parameter, even when using the sub-exponential principal ideal solver of [9]. It is still interesting to note that any future improvement for solving the principal ideal problem will directly imply an improvement for the attack described in this article.

Technical overview. Recent branching program obfuscators, starting with the one of [6], use the underlying multilinear map to prevent mixed-input attacks, using so-called straddling set systems. A mixed-input attack is an attack in which the attacker does not evaluate honestly the obfuscated circuit, but changes the value of one bit along the computation: for example, if the same bit of the entry is used twice during the computation, the attacker puts it to 1 the first time and to 0 the second time. By choosing good levels for the encodings of the multilinear map, the authors of [6] proved that one could prevent such dishonest computations: an attacker that tries to mix the bits of the input will obtain a final encoding which does not have the good level to be zero-tested and provide a useful output. Following this idea, the obfuscators of [2, 5, 23, 30, 32] also used straddling set systems to prevent mixed-input attacks.

However, straddling set systems only ensures that an attacker cannot mixed the inputs of the obfuscated program to obtain a dishonest top level encoding of zero. But it does not prevent an attacker to create a dishonest encoding of zero at a level higher than the top level. In the case where the multilinear map is ideal, this is not a security threat, because the attacker should not be able to test at a level higher than the top level whether it has created an encoding of zero or not. However, this is not the case of the GGH13 multilinear map. Indeed, using recent improvements on the short Principal Ideal Problem [10, 13, 18] (abbreviated as sPIP), it has been shown that it is possible to recover in quantum polynomial time some secret zero-testing element h of the GGH13 map (see Section 2.2 for more details on the GGH13 map). Recovering this secret element will then allow us to zero-test at a higher level than the one initially authorised.[7] This is the starting point of our mixed-input attack against the iO security of [2, 5, 6, 23, 30, 32].

As said above, all these candidate obfuscators use straddling set systems, meaning that performing a dishonest evaluation of the branching program outputs an encoding at a forbidden level. However, if we perform two well-chosen dishonest evaluations and take the product of the resulting encodings, we can obtain an encoding whose level is twice the maximal level of the multilinear map. The idea to construct well-chosen dishonest evaluations is to take complementary ones. For instance, assume the first bit of the input is used three times during the evaluation of the branching program. A first illegal computation could be to take this first bit to be equal to 0 the first time it is used, and then to 1 for the other two times. The complementary illegal computation will then be to take

[7] To be correct, we cannot really test whether we have an encoding of 0, but rather whether we have an encoding which is a product of two encodings of 0. More details can be found in Section 4.

the first bit to be equal to 1 the first time, and to 0 the other two times. These two illegal computation will result in encodings that are not at the top level, but there levels will be complementary in the sense that taking the product of them gives an encoding whose level is twice the top-level. We can then use the new zero-test parameter obtained above to determine whether this product of illegal encodings is an encoding of zero or not. It then remains to find a pair of equivalent branching programs such that the illegal encoding obtained above is an encoding of zero for one of the two branching programs only. We exhibit such a pair of branching programs in Section 4.3. While we just exhibit one pair, it should be possible to find many other pairs that can also be distinguished. We do not pursue this, as finding one such pair suffices to violate the iO property.

All the branching program obfuscators described above have a similar structure. In order to simplify the description of the attack, and to highlight which characteristics of these obfuscators are needed for the attack, we describe in Section 3 an abstract obfuscator, that captures the obfuscators of [2, 5, 23, 30, 32]. This abstract obfuscator is elementary, and it suffices to describe our attack against it, in order to attack all the obfuscators of [2,5,23,30,32]. The obfuscator of [6] does not completely fit in this abstract obfuscator and is discussed later.

We finally handle the case of the [19] obfuscator. This obfuscator is different from the ones presented above, as it encodes a circuit rather than a branching program. However, it also uses straddling set system to prevent mixed-input attacks. The same ideas as above can then be adapted to mount a mixed-input attack against the obfuscator of [19], in quantum polynomial time. Here, a new difficulty arises, as a dishonest evaluation of the circuit may not always be possible (for example it can lead to impossible additions, between encodings which are not at the same level). We handle this difficulty by choosing a specific universal circuit, for which we know that some dishonest evaluations are possible. As in the case of the branching program obfuscators, we then give an explicit example of two circuits whose obfuscated versions can be efficiently distinguished by a quantum attacker. Also, as for the the branching program obfuscators, we describe our attack against a simple circuit obfuscator, which captures the circuit obfuscator of [19]. This simple circuit also captures the circuits obfuscators of [4, 37], hence the attack also applies to these obfuscators, when they are instantiated with the GGH13 multilinear map.

Impact and open problems. To our knowledge, the only GGH13-based branching program or circuit obfuscator still standing against quantum attackers is the [22] branching program obfuscator, when combined with the technique of [20] to prevent input partitioning. We summarize in Table 1 the current state of the art attacks against branching program or circuit obfuscators based on the GGH13 multilinear map. The obfuscators relying on the CLT multilinear map are already known to be insecure against quantum attackers, as the CLT multilinear map is known to be broken if we can factor some public modulus, and we have a quantum polynomial time algorithm for factoring integers [35]. Finally, the obfuscator of [24], based on the GGH15 multilinear map, has been proven insecure against quantum attackers in [14]. In light of this, an interesting question could be

to assess the post-quantum security of the obfuscator of [22] when combined with [20].

Obfuscator (instantiated with the GGH13 map)	Quantum attack	Classical attack
[22] without [20]	[14]	[14]
[22] combined with [20]	none	none
[2, 6, 32] [5, 30]	[3, 31] and this work	[3, 31]
[12]	[3, 31]	[3, 31]
[4, 19, 23, 37]	this work	none

Fig. 1. Attacks against GGH13-based branching program and circuit obfuscators

Also, we show that solving the short Principal Ideal Problem enables us to mount a classical attack against the candidate obfuscators of [19, 23]. We could wonder whether the opposite is true: can we base the security of these candidate obfuscators or variants thereof on the short Principal Ideal Problem?

Finally, it is interesting to note that the mixed-input attack described in this article crucially relies on the use of straddling set systems. This may seem paradoxical, as straddling set systems were introduced to build obfuscators secure in idealized models, hence supposedly more secure than the first candidates. The first candidate obfuscators [12, 22] tried to prevent mixed-input attacks by using so-called bundling scalars, but it was heuristic and came with no proof. On the contrary, the use of straddling set systems allows us to *prove* that the schemes are resistant to mixed-input attacks if the underlying multilinear map is somehow ideal, hence giving us a security proof in some idealized model. However, this comes at the cost of relying more on the security of the underlying multilinear map. So when the obfuscators are instantiated with the GGH13 multilinear map, which is known to have some weaknesses, this gives more possibilities to an attacker to transform these weaknesses of the multilinear map into weaknesses of the obfuscators. This is what we do is this article, by transforming a weakness of the GGH13 map into a concrete attack against obfuscators using straddling set systems. It also explains why our attack does not apply to the obfuscators of [12, 22], which did not use straddling set systems.

Roadmap. In Section 2, we recall the GGH13 multilinear map, and the notion of matrix branching programs. In Section 3, we define an abstract obfuscator, which captures all the obfuscators of [2, 5, 23, 30, 32], with both single input and dual input variants. We will then use this abstract obfuscator to present our attack in Section 4. This will prove Theorem 1, except for the obfuscators of [6,19]. We then discuss in Section 4.4 how to adapt the attack to the obfuscator of [6]. Finally, we describe in Section 5 the obfuscator of [19] and explain how to adapt the mixed-input attack to this obfuscator, hence completing the proof of Theorem 1.

Acknowledgments. The author is grateful to Damien Stehlé for helpful discussions and comments on the draft. The author was supported by an ERC Starting Grant ERC-2013-StG-335086-LATTAC.

2 Preliminaries

In this section, we first recall some mathematical background and define some notations. We then recall the settings of the GGH13 multilinear map and the definition of matrix branching programs. Finally, we recall recent results for the Principal Ideal Problem, that we will use in our attack.

2.1 Mathematical Background

Rings. Let R be the ring $\mathbb{Z}[X]/(X^n+1)$ for n a power of two, and $K = \mathbb{Q}[X]/(X^n+1)$ be its fraction field. We let $R^\times$ denote the set of invertible elements of R. For an element $x \in K$, we let x_i denote its coefficients when seen as a polynomial of degree less than n, that is $x = \sum_{i=0}^{n-1} x_i X^i$. An ideal of R is a subset $I \subseteq R$ which is stable by addition and by multiplication by an element of R. If $I = gR = \{gr | r \in R\}$ for some element $g \in R$, we say that I is a principal ideal generated by g, and we denote it by gR or $\langle g \rangle$. We denote by $\{\sigma_j\}_{j\in[n]}$ the complex embeddings of K in $\mathbb{C}$. We can write these embeddings as $\sigma_1, \cdots, \sigma_{n/2}, \overline{\sigma_1}, \cdots, \overline{\sigma_{n/2}}$, where $\bar{\cdot}$ denotes the complex conjugation. The (algebraic) norm of an element $x \in K$ is $\mathcal{N}(x) = \prod_{j\in[n]} \sigma_j(x) \in \mathbb{R}$. The norm of an ideal $I \subseteq R$ is $\mathcal{N}(I) = |R/I|$. If $I = gR$ is a principal ideal, then $\mathcal{N}(I) = \mathcal{N}(g)$. The product of two ideals $I, J \subseteq R$, denoted by $I \cdot J$, is the smallest ideal containing $\{ab \mid a \in I, b \in J\}$. We say that an ideal $I \subseteq R$ is prime if $I \neq R$ and if for all ideals $J_1, J_2 \subseteq R$ such that $I = J_1 \cdot J_2$, then we have either $J_1 = R$ or $J_2 = R$.

Lattices. We view the ring R as an n-dimensional lattice, where the elements of R are mapped to the vectors of their coefficients, when seen as polynomials of degree $n-1$. For $x, y \in K$, the inner product of x and y is $\langle x, y \rangle = \sum_i x_i y_i$. We also define the ℓ_2 norm (or Euclidean norm) of $x \in K$ by $\|x\| = \sqrt{\sum_i x_i^2}$ and the infinite norm of x by $\|x\|_\infty = \max_i(x_i)$. Recall the following properties, for any $x, y \in K$

$$\|x \cdot y\| \leq \sqrt{n} \cdot \|x\| \cdot \|y\| \tag{1}$$

$$\|x\|_\infty \leq \|x\| \leq \sqrt{n} \cdot \|x\|_\infty. \tag{2}$$

For $x \in K$, the Minkowski embeddings of x is $\sigma(x) := (\mathrm{Re}(\sigma_1(x)), \mathrm{Im}(\sigma_1(x)), \cdots, \mathrm{Re}(\sigma_{n/2}(x)), \mathrm{Im}(\sigma_{n/2}(x))) \in \mathbb{R}^n$. We define the inner product of the Minkowski embeddings of two elements $x, y \in K$ by the usual inner product over $\mathbb{R}^n$ of $\sigma(x)$ and $\sigma(y)$. As we are in a cyclotomic ring of order a power of two, the geometry induced by the coefficient embeddings is the same, up to scaling, as the one

induced by the Minkowski embeddings. This means that for any $x, y \in K$, we have

$$\langle \sigma(x), \sigma(y) \rangle = n/2 \cdot \langle x, y \rangle.$$

In particular, for all $x \in K$, we have

$$\|\sigma(x)\|_2 = \sqrt{n/2} \cdot \|x\|, \tag{3}$$

where $\|\sigma(x)\|_2 = \sqrt{\langle \sigma(x), \sigma(x) \rangle}$.
An ideal I can be seen as a sub-lattice of R, and hence described by a $\mathbb{Z}$-basis. The Principal Ideal Problem (PIP) is, given a basis of a principal ideal I, to recover a generator of I, that is an element $g \in R$ such that $I = \langle g \rangle$.
For any lattice L, real $\sigma > 0$ and point $c \in L$, we define the Gaussian weight function over L by

$$\rho_{L,\sigma,c}(x) = \exp\left(\frac{-\|x-c\|^2}{2\sigma^2}\right).$$

We define the discrete (spherical) Gaussian distribution over L of parameter σ and centered in c by

$$\forall x \in L,\ D_{L,\sigma,c}(x) = \frac{\rho_{L,\sigma,c}(x)}{\rho_{L,\sigma,c}(L)},$$

where $\rho_{L,\sigma,c}(L) = \sum_{x \in L} \rho_{L,\sigma,c}(x)$. We simplify $\rho_{L,\sigma,0}$ and $D_{L,\sigma,0}$ into $\rho_{L,\sigma}$ and $D_{L,\sigma}$, and say in that case that the distribution is centered.

2.2 The GGH13 multilinear map

We recall in this section the GGH13 multilinear map (or shortly GGH map) of [21], in its asymmetric setting. The GGH multilinear map allows to encode elements of ring. We can then homomorphically perform additions and multiplications on these elements, under some constraints. It also allows to publicly test if an encoding encodes zero. Let q be a large integer (usually taken exponential in n) and define $R_q = R/qR$. Let g be some small element of $R^\times$ chosen such that the ideal $\langle g \rangle$ is prime and has a prime norm. The plaintext space will be R/gR and the encoding space will be R_q.

Encodings. Let κ be some positive integer and $z_1, \cdots, z_\kappa$ be chosen randomly in $R_q^\times$.[8] These z_i's are chosen during the initialisation phase of the GGH map. Let S be a subset of $[\kappa]$ and $a + gR$ be an element of R/gR. An encoding of $a + gR$ at level S is an element of the form

$$u = c \cdot \prod_{i \in S} z_i^{-1} \bmod q,$$

[8] The distribution of the z_i's does not matter here.

where c is a small representative of $a + gR$ in R. We sometimes abuse notation by saying that u is an encoding of $a \in R$ instead of $a + gR \in R/gR$. We use the notation $[a]_S$ to denote an encoding of $a + gR$ at level S. When there is no ambiguity on the level of the encoding, we just write it $[a]$, with no subscript. We say that c is the numerator of the encoding u and $\prod_{i \in S} z_i$ is its denominator.

Operations on encodings. Let u_1 and u_2 be the encodings of two elements a_1 and a_2 at the same level S. Then $u_1 + u_2$ is an encoding of $a_1 + a_2$ at level S. Let u_1 and u_2 be the encodings of two elements u_1 and u_2 at level S_1 and S_2 respectively, with $S_1 \cap S_2 = \emptyset$. Then $u_1 \cdot u_2$ is an encoding of $a_1 \cdot a_2$ at level $S_1 \cup S_2$.[9]

Zero-testing. Let S_{zt} denote the set $[\kappa]$ and $z^* = \prod_{i \in S_{zt}} z_i$. Let h be some element in R of ℓ_2-norm approximately $\sqrt{q}$. We define $p_{zt} = hz^*g^{-1} \bmod q$ and call it the zero-testing parameter. To test if an encoding u at level S_{zt} is an encoding of zero or not (i.e., to test if the numerator of u is a multiple of g or not), compute $w = u \cdot p_{zt} \bmod q$. If this is smaller than $q^{3/4}$,[10] then u is an encoding of zero, otherwise it is not. Indeed, if $u = bg(z^*)^{-1} \bmod q$ (i.e., u is an encoding of zero), then $w = bh \bmod q$ and the parameters are set such that $||bh|| \leq q^{3/4}$ for a correct level-S_{zt} encoding. On the other hand, if u is not an encoding of zero, then the g^{-1} in the zero-testing parameter does not cancel out, and $g^{-1} \bmod q$ is very unlikely to be small compared to q. We can prove that in this case, w will never be smaller than $q^{3/4}$ (see [21] for more details).

The elements (n, q, κ, p_{zt}) of the multilinear map are public, while the parameters $(h, g, \{z_i\}_{i \in [\kappa]})$ are secret. In our case, the obfuscator generates the multilinear maps and retains these secret elements. Note that to encode an element, we need to know the secret parameters g and $\{z_i\}_i$. This means that only the obfuscator will be able to create encodings from scratch. An encoding generated by the obfuscator, using the secret parameters, is called a fresh encoding, by opposition to the encodings obtained by adding or multiplying other encodings.

Size of the parameters. The size of the parameters of the GGH multilinear map may vary depending on the obfuscator. We present here the size recommended in the original article [21], with a small change for the size of q, due to the fact that we use the multilinear map in a different way for obfuscators than what was described in [21].

- The dimension n of R should be taken such that $n = \Omega(\kappa\lambda^2)$, where λ is the security parameter of the scheme. Taking a lower bound in λ^2 was the original choice of [21] to avoid some lattice attacks. It was reduced to $n = \Omega(\kappa\lambda \log(\lambda))$ in [28]. However, with the recent sub-exponential algorithm of [9] to solve PIP, it should be increased back to $\Omega(\kappa\lambda^2)$. Looking

[9] Even if $S_1 \cap S_2 \neq \emptyset$, we can still see $u_1 \cdot u_2$ as an encoding of $a_1 \cdot a_2$ at level $S_1 \cup S_2$, where $S_1 \cup S_2$ is a multiset, that is we keep multiple copies of elements that appear both in S_1 and S_2.

[10] This bound is the one chosen in [21], but it is flexible.

ahead, the attack we describe in Section 4 has a classical variant which is sub-exponential in the dimension n of the lattice (it has a complexity $O(2^{\sqrt{n}+o(1)})$). However, as $n \geq \Omega(\lambda^2)$, this remains exponential in the security parameter λ.

- The secret element g is sampled using a Gaussian distribution, with rejection, such that $\|g\| = O(n)$ and $\|1/g\| = O(n^2)$.
- The modulus q is chosen such that $q \geq n^{O(\kappa)}$. In the original GGH scheme, the modulus q was chosen greater than $2^{8\kappa\lambda} \cdot n^{O(\kappa)}$. This extra factor $2^{8\kappa\lambda}$ came from the re-randomisation procedure used originally to publicly generate level-1 encodings. In the case of obfuscators, as the one that generates encodings knows the secret parameters, it can generates the fresh encodings with a numerator of size $O(\text{poly}(n))$ instead of $O(2^{\lambda}\text{poly}(n))$, and hence get ride of this factor $2^{8\kappa\lambda}$. In all the obfuscators described here, except [19], the modulus q is exponential in λ. In [19], the obfuscator is built such that q remains polynomial in λ (even if κ is polynomial in λ, the authors managed to obtain a polynomial modulus q).
- The secret element h is sampled using a centered Gaussian distribution of parameter $\sqrt{q}$, so that $\|h\| = \Theta(\sqrt{n} \cdot \sqrt{q})$. In [21, Section 6.4], the authors suggest to sample h according to a non spherical Gaussian distribution instead of a spherical one. In the following we will always assume that h is sampled according to a spherical Gaussian distribution. We discuss the case of non spherical distributions in the full version [33].

2.3 Matrix Branching Programs

We recall in this section the definition of matrix branching programs, and we introduce some notation that will be used throughout the article. A branching program is defined over a ring $\mathcal{R}$.

Definition 1 (d-ary Matrix Branching Program [3]). *A d-ary matrix branching program $\boldsymbol{A}$ of length ℓ and width w over m-bit inputs is given by a sequence of square matrices*

$$\{A_{i,\boldsymbol{b}}\}_{i\in[\ell],\boldsymbol{b}\in\{0,1\}^d} \in \mathcal{R}^{w\times w},$$

two bookend vectors

$$A_0 \in \mathcal{R}^{1\times w} \text{ and } A_{\ell+1} \in \mathcal{R}^{w\times 1},$$

and an input function $\boldsymbol{inp} : [\ell] \to [m]^d$.

Let $x \in \{0,1\}^m$ and let x_i denote the i-th bit of x, for i in $[m]$. We will use the notation $x[\boldsymbol{inp}(i)] = (x_{\boldsymbol{inp}(i)_1}, x_{\boldsymbol{inp}(i)_2}, \cdots, x_{\boldsymbol{inp}(i)_d}) \in \{0,1\}^d$, where $\boldsymbol{inp}(i) = (\boldsymbol{inp}(i)_1, \cdots, \boldsymbol{inp}(i)_d) \in [m]^d$.

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

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

Remark. A branching program with $d = 1$ (respectively with $d = 2$) is also called a single input (respectively dual input) branching program. In the following, we will not distinguish between the single input and dual input cases, as our attack works in the same way in both cases (and even for higher arity d).

We say that two branching programs are equivalent if they compute the same function. We also introduce a notion of strong equivalence between branching programs, which will be useful later for the description of the abstract obfuscator and our attack.

Definition 2 (Strongly equivalent branching programs). *We say that two d-ary matrix branching programs $\boldsymbol{A} = (A_0, \{A_{i,\boldsymbol{b}}\}_{i\in[\ell],\boldsymbol{b}\in\{0,1\}^d}, A_{\ell+1})$ and $\boldsymbol{A}' = (A'_0, \{A'_{i,\boldsymbol{b}}\}_{i\in[\ell],\boldsymbol{b}\in\{0,1\}^d}, A'_{\ell+1})$, with the same length ℓ and the same input function $\mathtt{inp}$ (but not necessarily defined over the same rings) are strongly equivalent if, for all $\{\boldsymbol{b}_i\}_{i\in[\ell]} \in (\{0,1\}^d)^\ell$, we have*

$$A_0 \cdot \prod_{i\in[\ell]} A_{i,\boldsymbol{b}_i} \cdot A_{\ell+1} = 0 \iff A'_0 \cdot \prod_{i\in[\ell]} A'_{i,\boldsymbol{b}_i} \cdot A'_{\ell+1} = 0. \tag{4}$$

Remark. This notion is stronger than simple equivalence between branching programs, because we ask that (4) holds for all possible choices of $\{\mathbf{b}_i\}_{i\in[\ell]}$, and not only for the ones of the form $\{x[\mathtt{inp}(i)]\}_{i\in[\ell]}$ for some input x (corresponding to an honest evaluation of the branching program on x). The pair of branching programs described in Section 4.3 gives an example of equivalent branching programs that are not strongly equivalent.

2.4 The short Principal Ideal Problem

We define the short Principal Ideal Problem in the following way.

Definition 3 (Short Principal Ideal Problem). *Let $h \in R$ be sampled according to some distribution D. The short Principal Ideal Problem is, given any basis of the ideal $\langle h\rangle$ (when seen as a sub-lattice of R), to recover $\pm X^i \cdot h$ for some $i \in [n]$.*

For cyclotomic fields of order a power of two, when D is a discrete Gaussian distribution, this problem can be solved in quantum polynomial time, using the results of [10, 13, 18]. In [10], the authors show that given any basis of $\langle h\rangle$, an attacker can recover a generator $\widetilde{h}$ of the ideal $\langle h\rangle$ in quantum polynomial time.[11] Then, the authors of [18], based on an observation of [13], proved that from any generator $\widetilde{h}$ of $\langle h\rangle$, if h has been sampled using a discrete Gaussian distribution, then an attacker can recover $\pm X^i \cdot h$, for some $i \in [n]$, in (classical) polynomial time. This second part (recovering $\pm X^i \cdot h$ from $\widetilde{h}$) relies on the conjecture that the set of cyclotomic units of R is equal to $R^\times$ for power-of-two cyclotomic fields. We summarise this in the following theorem.

[11] Note that there also exists a classical sub-exponential time algorithm to recover $\widetilde{h}$, due to [9]. However, their algorithm runs in time $O(2^{\sqrt{n}+o(1)})$, but we chose $n \geq \Omega(\lambda^2)$, so this algorithm is exponential in the security parameter λ.

Theorem 2 (adapted from [10, 18]). *Let $h \in R$ be sampled according to a discrete spherical Gaussian distribution of parameter larger than $200 \cdot n^{1.5}$. Then, under Conjecture 1, there is a quantum polynomial time algorithm such that, given any basis of the ideal $\langle h \rangle$, it recovers $\pm X^i \cdot h$ for some $i \in [n]$, with constant probability close to 1 over the choice of h.*

Conjecture 1. The set of cyclotomic units of R is equal to $R^\times$ (see [18] for a definition of cyclotomic units and a discussion of this conjecture).

3 An Abstract obfuscator

Following an idea of Miles, Sahai and Zhandry in [31], we define here an abstract obfuscation scheme. This abstract obfuscator is inspired by the one of [31] but is a bit simpler and more general. In particular, it captures all the obfuscators of Theorem 1, except the ones of [6] and [19]. We will then show in Section 4 how to apply our quantum attack to this abstract obfuscator, resulting in an attack against the obfuscators of [2, 5, 23, 30, 32] and we will explain how to adapt the attack to the branching program obfuscator of [6] (which is just slightly different from the abstract obfuscator defined in this section). The case of the [19] obfuscator is postponed in Section 5 as it is not a branching program obfuscator, and so the formalism of the abstract branching program obfuscator does not apply to it.

The abstract obfuscator takes as input a polynomial size d-ary matrix branching program $\mathbf{A}$ (for some integer $d > 0$), over the ring of integers $\mathbb{Z}$,[12] with a fixed input function $\mathtt{inp}$ and with coefficients in $\{0, 1\}$. Usually, the obfuscators pad the branching program with identity matrices, to ensure that the input function has the desired structure. Here, to simplify the obfuscator, we will assume that the obfuscator only accepts branching programs with the desired $\mathtt{inp}$ function (the user has to pad the branching program himself before giving it to the obfuscator). For the attack to work, we ask that there exist two different integers j_1 and j_2 such that $\mathtt{inp}(j_1) \cap \mathtt{inp}(j_2) \neq \emptyset$ (meaning that there is a bit of the input which is inspected at least twice during the evaluation of the branching program). This can be assumed for all the obfuscators of Theorem 1.[13] Let w be the width of $\mathbf{A}$, ℓ be its length, A_0, $A_{\ell+1}$ be its bookend vectors and $\{A_{i,\mathbf{b}}\}_{i\in[\ell],\mathbf{b}\in\{0,1\}^d} \in \{0,1\}^{w\times w}$ be its square matrices. Recall that the function computed by the branching program $\mathbf{A}$ is defined by

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

The abstract obfuscator then proceeds as follows.

[12] Most of the time, the matrices of the branching program will be permutation matrices, and the underlying ring will have no importance.

[13] This is even mandatory for the dual input version of the obfuscators, as it is usually required that all pairs (s, t) (or (t, s)) appear in the $\mathtt{inp}$ function, for any $s, t \in [m]$ with $s \neq t$.

- It instantiates the GGH multilinear map and retains its secret parameters $(g, h, \{z_i\}_{i\in[\kappa]})$ and its public parameters (n, q, κ, p_{zt}). The choice of the parameters of the GGH map depends on the the parameters ℓ, w and d of the branching program $\mathbf{A}$.
- It transforms the matrices of branching program $\mathbf{A}$ to obtain a new branching program $\hat{\mathbf{A}}$, with the same parameters w, d, ℓ, the same input function $\mathtt{inp}$, and which is strongly equivalent to $\mathbf{A}$. We denote by $\{\hat{A}_{i,\mathbf{b}}\}_{i\in[\ell],\mathbf{b}\in\{0,1\}^d} \in (R/gR)^{w\times w}$ and $\hat{A}_0 \in (R/gR)^{1\times w}, \hat{A}_{\ell+1} \in (R/gR)^{w\times 1}$ the matrices and bookend vectors of $\hat{\mathbf{A}}$. Note that this new matrix branching programs has its coefficients in the ring R/gR and not in $\{0,1\}$. Recall that strong equivalence means that

$$A_0 \cdot \prod_{i\in[\ell]} A_{i,\mathbf{b}_i} \cdot A_{\ell+1} = 0 \iff \hat{A}_0 \cdot \prod_{i\in[\ell]} \hat{A}_{i,\mathbf{b}_i} \cdot \hat{A}_{\ell+1} = 0 \text{ (in } R/gR) \qquad (5)$$

  for all choices of $\mathbf{b}_i \in \{0,1\}^d$, with $i \in [\ell]$. This condition is required for our attack to work, and is satisfied by all the obfuscators of [2, 5, 23, 30, 32]. To transform the initial branching program $\mathbf{A}$ into this new branching program $\hat{\mathbf{A}}$, the obfuscators of [2,5,23,30,32] first embed the matrices of $\mathbf{A}$ into the ring R/gR (this is possible since the coefficients of the matrices are 0 and 1). Then, they use various tools, taken among the following.[14]

  1. Transform the matrices $A_{i,\mathbf{b}}$ into block-diagonal matrices $\begin{pmatrix} A_{i,\mathbf{b}} & \\ & B_{i,\mathbf{b}} \end{pmatrix}$, were $B_{i,\mathbf{b}}$ are square $w' \times w'$ matrices in R/gR, chosen arbitrarily (they can be fixed, or chosen at random, this will have no importance for us), with w' polynomial in the security parameter λ. In order to cancel the extra diagonal block, the vector A_0 is transformed into $\begin{pmatrix} A_0 & 0 \end{pmatrix}$, with a block of zeros of size $1 \times w'$. The vector $A_{\ell+1}$ is transformed into $\begin{pmatrix} A_{\ell+1} \\ B_{\ell+1} \end{pmatrix}$, with $B_{\ell+1}$ an arbitrary $w' \times 1$ vector.
  2. Use Killian randomisation, that is, choose $\ell + 1$ non singular matrices $\{R_i\}_{i\in[\ell+1]} \in (R/gR)^{w\times w}$ and transform $A_{i,\mathbf{b}}$ into $R_i \cdot A_{i,\mathbf{b}} \cdot R_{i+1}^{\text{adj}}$, where R_{i+1}^{adj} is the adjugate matrix of R_{i+1}, i.e., $R_{i+1} \cdot R_{i+1}^{\text{adj}} = \det(R_{i+1}) \cdot I_n$. Transform also A_0 into $A_0 \cdot R_1^{\text{adj}}$ and $A_{\ell+1}$ into $R_{\ell+1} \cdot A_{\ell+1}$.
  3. Multiply by random scalars, i.e., multiply each matrix $A_{i,\mathbf{b}}$ by some random scalar $\alpha_{i,\mathbf{b}} \in (R/gR)^\times$. Also multiply A_0 and $A_{\ell+1}$ by α_0 and $\alpha_{\ell+1}$ respectively.

  We can check that all the transformations described above output a branching program which is strongly equivalent to the one given in input, so the final branching program $\hat{\mathbf{A}}$ is also strongly equivalent to $\mathbf{A}$ (as in (5)). In the following, we will only be interested in (5), not in the details of the transformation.

[14] The obfuscators of [23, 32] use the three tools while the ones of [2, 5, 30] use Tools 2 and 3 only.

- Finally, the obfuscator encodes the matrices $\{\hat{A}_{i,\mathbf{b}}\}_{i,\mathbf{b}}$, $\hat{A}_0$ and $\hat{A}_{\ell+1}$ at some level $\{S_{i,\mathbf{b}}\}_{i,\mathbf{b}}$, S_0 and $S_{\ell+1}$ respectively, using the GGH multilinear map. The choice of these levels (called a straddling set system) depends on the obfuscators, but will have no importance in the following. The only property that we need, and that is fulfilled by the above obfuscators, is that for any entry x, the sets $S_0, S_{\ell+1}$ and $S_{i,x[inp(i)]}$ for $i \in [l]$ are disjoint and we have

$$S_0 \cup \left(\cup_{i\in[l]} S_{i,x[inp(i)]}\right) \cup S_{\ell+1} = S_{zt}. \tag{6}$$

  This means that every honest evaluation of the encoded branching program outputs an element at level S_{zt}, that can be zero-tested. This condition is necessary for the above obfuscators to be correct (otherwise we cannot evaluate the obfuscated branching program).

- The obfuscator then outputs the elements $[\hat{A}_0]_{S_0}$, $\{[\hat{A}_{i,\mathbf{b}}]_{S_{i,\mathbf{b}}}\}_{i\in[l],\mathbf{b}\in\{0,1\}^d}$, $[\hat{A}_{\ell+1}]_{S_{\ell+1}}$ and the public parameters of the GGH map (n, q, κ, p_{zt}).

  To evaluate the obfuscated branching program on input x, compute

$$u_x = [\hat{A}_0]_{S_0} \times \prod_{i\in[\ell]} [\hat{A}_{i,x[\mathtt{inp}(i)]}]_{S_{i,x[\mathtt{inp}(i)]}} \times [\hat{A}_{\ell+1}]_{S_{\ell+1}}.$$

By Property (5), this is an encoding of zero if and only if the output of the original branching program was zero. And by Property (6), this encoding is at level S_{zt}. So using p_{zt}, we can perform a zero-test and output 0 if this is an encoding of 0 and 1 otherwise. In the following, we will sometimes simplify notations and forget about the subscripts $S_{i,\mathbf{b}}$, as the levels of the encodings are entirely determined by the encoded matrices $A_{i,\mathbf{b}}$.

For our attack to work, we will need to assume that if we evaluate the obfuscated branching program on enough inputs for which the output is zero, then we can recover a basis of the ideal $\langle h\rangle$ (where h is a secret element of the GGH13 map, as described in Section 2.2). More formally, we make the following heuristic assumption.

Heuristic 1. Let X_0 be the set of inputs on which the branching program evaluates to 0 and let $x \in X_0$. If we evaluate the obfuscated branching program on x and zero-test the final encoding, we obtain a ring element of the form $r_x \cdot h \in R$. We assume that the set of all $r_x \cdot h$ for $x \in X_0$ spans the ideal $\langle h\rangle$ (and not a smaller ideal contained in $\langle h\rangle$). We also assume that if x is chosen uniformly in X_0, then we can obtain a basis of $\langle h\rangle$ with a polynomial number of samples.

Discussion about Heuristic 1. We make the heuristic assumption above to simplify the description of our attack. This heuristic assumption is coherent with the numerical experiments we made (see the full version [33] for a description of the experimental results). Moreover, we also observe that, even if we recover an ideal $J \subseteq \langle h\rangle$ instead of the ideal $\langle h\rangle$, we can still handle it if $\langle h\rangle$ has a constant number of prime factors (see the full version for more details).

This completes the definition of our abstract obfuscator, which captures the obfuscators of [2, 5, 23, 30, 32]. In the next section, we describe a mixed-input attack against this abstract obfuscator, where all we use is that it satisfies Properties (5) and (6).

4 The main Attack

We will now prove our main theorem.

Theorem 3. *Let $\mathcal{O}$ be any of the obfuscators in [2, 5, 6, 23, 30, 32], on single or dual input branching programs, instantiated with the GGH13 multilinear map [21] (respectively, let $\mathcal{O}$ be any of the circuit obfuscators in [4, 19, 37]). Assume the secret parameter h of the GGH13 multilinear map is sampled using a spherical Gaussian distribution (as in Section 2.2). Then, there exist two explicit equivalent branching programs (respectively, two equivalent circuits) $\boldsymbol{A}$ and $\boldsymbol{A}'$ such that $\mathcal{O}(\boldsymbol{A})$ and $\mathcal{O}(\boldsymbol{A}')$ can be distinguished in quantum polynomial time, under Conjecture 1 and Heuristic 1.*

The limitation to the case where h is sampled according to a spherical Gaussian distribution is discussed in the full version. We show that if q is large enough, or if h is a product of a small number of spherical Gaussian distributions, then our result still holds. We leave as an open problem to show that the attack goes through for every efficient way of sampling h, or to find a way that allows to thwart the attack (although we lean towards the former rather than the latter). The necessity for h being sampled according to a spherical Gaussian distribution appears in Theorem 2, to solve the short Principal Ideal Problem and recover the secret element h. It is not used anywhere else in the attack, in particular, it is not used in the mixed-input part of the attack (see Section 4.2).

To prove Theorem 3, we present a quantum polynomial time attack against the abstract obfuscator described in Section 3. This results into an attack against the iO security of the branching program obfuscators of [2, 5, 23, 30, 32]. We then explain how to slightly modify this attack to use it against the obfuscator of [6], whose structure is very close to the one of the abstract obfuscator. Finally, adapting the attack to the circuit obfuscator of [19] will require more work, because its structure is further away from the abstract obfuscator.

The attack works in two steps. We first recover the secret element h of the GGH multilinear map. Using the results of [10, 13, 18], recalled in Section 2.4, this can be done in quantum polynomial time. Knowing this secret element h, we are able to construct a zero-testing parameter p'_{zt} at a higher level than S_{zt}. We can then use this new parameter p'_{zt} to mount a (classical) polynomial time mixed-input attack against the abstract obfuscator.

4.1 Creating a new zero-testing parameter in quantum polynomial time

We first explain in this section how we can recover the secret parameter h of the multilinear map in quantum polynomial time. We then describe how to construct

a new zero-testing parameter at a level higher than S_{zt}, using h. Note that the following is folklore, we recall it for the sake of completeness.

The first step is to recover sufficiently many multiples of h, to obtain a basis of the ideal $\langle h \rangle$ (when seen as a sub-lattice of R). This part of the attack was already described in the original article [21], and can be done in classical polynomial time, under Heuristic 1. Observe that for each top-level encoding that pass the zero-test, we obtain a multiple of h. We make the heuristic assumption 1 to ensure that we indeed recover a basis of the ideal $\langle h \rangle$, by zero-testing sufficiently many top-level encodings of zero. For this step to work, we need that the branching program evaluates sufficiently often to 0, to obtain sufficiently many encodings of 0. In the following, we will choose branching programs that compute the always zero function, hence the condition on the number of encodings that pass the zero-test will be satisfied.

We then recover $\pm X^i h$ from the basis of the ideal $\langle h \rangle$, using Theorem 2. This can be done in quantum polynomial time, under Conjecture 1, as h is sampled according to a Gaussian distribution of parameter larger than $200 \cdot n^{1.5}$. The fact that we recover $\pm X^j h$ instead of h will have no importance for our attack,[15] so in the following we will assume that we recovered h exactly. In [21, Section 6.4], the authors propose another distribution for the secret parameter h (the element h is sampled according to a non spherical Gaussian distribution). Theorem 2 does not apply as it in this case, but we show in the full version that our attack can be extended to some other distributions of h.

We now explain how to use h to create a new zero-testing parameter p'_{zt} at a higher level than S_{zt}. A close variant of this step was already mentioned in [21, Section 6.3.3]. The authors explained how to use a small multiple of $1/h$ and a low level encoding of zero to create a new zero-testing parameter that enabled to test at a higher level whether the numerator of an encoding was a multiple of g or not (i.e., if the encoding was an encoding of zero or not). In our case, the situation is a little different, as we do not know any low level encoding of zero. Hence, we only manage to create a new zero-testing parameter that enables us to determine whether the numerator of an encoding is a multiple of g^2 or not. In the following, we will say that an encoding is an encoding at level $2S_{zt}$ if its denominator is $(z^*)^2$. For instance, such an encoding can be obtained by multiplying two level S_{zt} encodings. We see the level $2S_{zt}$ as a multiset containing all the elements of S_{zt} twice. We use the secret h to compute a new zero-testing parameter p'_{zt} at level $2S_{zt}$. Recall that $p_{zt} = hz^*g^{-1} \bmod q$. We then define

$$p'_{zt} = p_{zt}^2 h^{-2} \bmod q = (z^*)^2 \cdot g^{-2} \bmod q.$$

Again, note that even if we call it a new zero-testing parameter, p'_{zt} only enables us to test whether the numerator of a level $2S_{zt}$ encoding is a multiple of g^2, and not g, as our original zero-test parameter p_{zt} did. But still, being able to test at a level higher than S_{zt} if the numerator is a multiple of g^2 will enable us to mount a mixed-input attack against the abstract obfuscator of Section 3. We describe this mixed-input attack in the next subsection.

[15] This is because both X^j and its inverse $-X^{n-j}$ have euclidean norm 1.

4.2 The mixed-input attack

We now assume that we have built a new pseudo-zero-test parameter p'_{zt}, as in Subsection 4.1 (in quantum polynomial time), and that we are given an obfuscated branching program $([\hat{A}_0]_{S_0}, \{[\hat{A}_{i,\mathbf{b}}]_{S_{i,\mathbf{b}}}\}_{i\in[l],\mathbf{b}\in\{0,1\}^d}, [\hat{A}_{\ell+1}]_{S_{\ell+1}})$, obtained by using our abstract obfuscator defined in Section 3.

Let x and y be two different inputs of the branching program. A mixed-input attack consists in changing the value of some bits of the input during the evaluation of the obfuscated branching program. For instance, the way we will do it is by taking some matrix $[\hat{A}_{i,y[\mathtt{inp}(i)]}]_{S_{i,y[\mathtt{inp}(i)]}}$, instead of $[\hat{A}_{i,x[\mathtt{inp}(i)]}]_{S_{i,x[\mathtt{inp}(i)]}}$, while evaluating the program on x. Such mixed-input attack can leak information on the program being obfuscated (see the specific choice of branching programs described in the next subsection). In order to prevent mixed-input attack, the abstract obfuscator uses a straddling set system. The intuition is that if the attacker tries to mix the matrices $[\hat{A}_{i,x[\mathtt{inp}(i)]}]_{S_{i,x[\mathtt{inp}(i)]}}$ and $[\hat{A}_{i,y[\mathtt{inp}(i)]}]_{S_{i,y[\mathtt{inp}(i)]}}$, it will not get an encoding at level S_{zt} at the end of the computation and hence it cannot zero-test it. However, we can use our new zero-testing parameter p'_{zt} to handle this difficulty.

Let $j \in [\ell]$ and compute

$$\tilde{u}_{x,j} = [\hat{A}_0] \cdot \prod_{i<j} [\hat{A}_{i,x[\mathtt{inp}(i)]}] \cdot [\hat{A}_{j,y[\mathtt{inp}(j)]}] \cdot \prod_{j<i\leq\ell} [\hat{A}_{i,x[\mathtt{inp}(i)]}] \cdot [\hat{A}_{\ell+1}]$$
$$\tilde{u}_{y,j} = [\hat{A}_0] \cdot \prod_{i<j} [\hat{A}_{i,y[\mathtt{inp}(i)]}] \cdot [\hat{A}_{j,x[\mathtt{inp}(j)]}] \cdot \prod_{j<i\leq\ell} [\hat{A}_{i,y[\mathtt{inp}(i)]}] \cdot [\hat{A}_{\ell+1}],$$

that is, we exchange $[\hat{A}_{j,x[\mathtt{inp}(j)]}]_{S_{j,x[\mathtt{inp}(j)]}}$ and $[\hat{A}_{j,y[\mathtt{inp}(j)]}]_{S_{j,y[\mathtt{inp}(j)]}}$ in the honest evaluations of the obfuscated branching program on x and y.

The encodings $\tilde{u}_{x,j}$ and $\tilde{u}_{y,j}$ will have illegal levels S_x and S_y that are different from S_{zt}. But as we only exchange two matrices between correct evaluations, we know that $\tilde{u}_{x,j} \cdot \tilde{u}_{y,j}$ will be encoded at the same level as $u_x \cdot u_y$ where u_x and u_y are the correct evaluations of the obfuscated branching program on x and y. As u_x and u_y are correct evaluations, using Property (6), we know that they are encoded at level S_{zt}. Hence $\tilde{u}_{x,j} \cdot \tilde{u}_{y,j}$ is encoded at level $2S_{zt}$, and we can zero-test $\tilde{u}_{x,j} \cdot \tilde{u}_{y,j}$ using p'_{zt}.

Remember that an encoding will pass this zero-test only if its numerator is a multiple of g^2 and not only g. A simple way to ensure that $\tilde{u}_{x,j} \cdot \tilde{u}_{y,j}$ has a numerator which is a multiple of g^2 is to choose x and y such that $\tilde{u}_{x,j}$ and $\tilde{u}_{y,j}$ are both encodings of 0 (i.e., their numerator are both multiples of g, and hence their product has a numerator which is a multiple of g^2). Using Property (5) of our abstract obfuscator, we know that $\tilde{u}_{x,j}$ is an encoding of 0 if and only if

$$A_0 \cdot \prod_{i<j} A_{i,x[\mathtt{inp}(i)]} \cdot A_{j,y[\mathtt{inp}(j)]} \cdot \prod_{j<i\leq\ell} A_{i,x[\mathtt{inp}(i)]} \cdot A_{\ell+1} = 0.$$

We denote by $\tilde{a}_{x,j}$ the left hand side of this equation. In the same way, we define

$$\tilde{a}_{y,j} = A_0 \cdot \prod_{i<j} A_{i,y[\mathtt{inp}(i)]} \cdot A_{j,x[\mathtt{inp}(j)]} \cdot \prod_{j<i\leq\ell} A_{i,y[\mathtt{inp}(i)]} \cdot A_{\ell+1},$$

and we have that $\tilde{u}_{y,j}$ is an encoding of 0 if and only if $\tilde{a}_{y,j} = 0$.

To conclude, if we manage to find two equivalent branching programs $\mathbf{A}$ and $\mathbf{A}'$, two inputs x and y and an integer $j \in [\ell]$ such that $\tilde{a}_{x,j} = \tilde{a}_{y,j} = 0$ for $\mathbf{A}$ but $\tilde{a}'_{x,j} \neq 0$ and $\tilde{a}'_{y,j} \neq 0$ for $\mathbf{A}'$, then we can distinguish between the obfuscation of $\mathbf{A}$ and the one of $\mathbf{A}'$. Indeed, the numerator of $\tilde{u}_{x,j} \cdot \tilde{u}_{y,j}$ will be a multiple of g^2 in the case of $\mathbf{A}$ but the numerator of $\tilde{u}'_{x,j} \cdot \tilde{u}'_{y,j}$ will not be a multiple of g in the case of $\mathbf{A}'$ (and therefore not a multiple of g^2 either). Hence, using p'_{zt}, we can determine which of the branching program $\mathbf{A}$ or $\mathbf{A}'$ has been obfuscated.

In the next subsection, we present two possible branching programs $\mathbf{A}$ and $\mathbf{A}'$ and inputs x and y that satisfy the condition above. We note that this condition is easily satisfied and it should be possible to find a lot of other branching programs satisfying it. We just propose here a simple example of such branching programs, in order to complete the proof of Theorem 1.

4.3 A concrete example of branching programs

In this section, we present an example of two branching programs $\mathbf{A}$ and $\mathbf{A}'$ that are equivalent, but such that their obfuscated versions, obtained using the abstract obfuscator, can be distinguished using the framework described above, hence attacking the iO security of the obfuscator.

Remember that for the first step of our attack (recovering h and creating p'_{zt}, see Section 4.1), we need to have a sufficient number of inputs x that evaluate to zero. Here, we choose branching programs that compute the always zero function. We now show how to satisfy the conditions for the second part of the attack (Section 4.2).

Let $I = I_w \in \{0,1\}^{w \times w}$ be the identity matrix and $J \in \{0,1\}^{w \times w}$ be a matrix of order two (i.e., $J \neq I$ and $J^2 = I$). One could for example take $J = \begin{pmatrix} 0 & 1 & & \\ 1 & 0 & & \\ & & I_{w-2} & \end{pmatrix}$. Our first branching program will consist in identity matrices only. We will build our second branching program such that when evaluating it on input x, we have a product of ℓ matrices I (when we forget about the bookend vectors), but on input y we have a product of $\ell - 2$ matrices I and 2 matrices J.[16] We will then exchange one of these J matrices with an I matrix in the evaluation on x. The resulting products will then be equal to matrix J instead of matrix I (as it is the case for the first branching program). We describe the two branching programs more precisely below.

Input selection function. Recall that the input selection function $\texttt{inp}$ is fixed and is such that there are at least two distinct integers j_1 and j_2 such that $\texttt{inp}(j_1) \cap \texttt{inp}(j_2) \neq \emptyset$. Let s be such that $s \in \texttt{inp}(j_1) \cap \texttt{inp}(j_2)$. This means that when evaluating the branching program on some input, the j_1-th and the j_2-th matrices of the product both depend on the s-th bit of the input. Without loss of

[16] As J has order 2, the resulting product will still be the identity matrix.

generality, we assume that $\texttt{inp}(j_1) = (s, s_2, \cdots, s_d)$ and $\texttt{inp}(j_2) = (s, t_2, \cdots, t_d)$ for some integers s_i and t_i in $[m]$.

Matrices. Our first branching program $\mathbf{A}$ consists in identity matrices only, i.e., $A_{i,\mathbf{b}} = I$ for all $i \in [\ell]$ and $\mathbf{b} \in \{0,1\}^d$. For our second branching program $\mathbf{A}'$, we take

$$A'_{i,\mathbf{b}} = \begin{cases} I \text{ if } i \notin \{j_1, j_2\} \text{ or } b_1 = 0 \\ J \text{ if } i \in \{j_1, j_2\} \text{ and } b_1 = 1, \end{cases}$$

where $\mathbf{b} = (b_1, \cdots, b_d)$. This means that when evaluating the branching program $\mathbf{A}'$ on some input x, if $x_s = 0$, then all the matrices of the product are identity matrices. And if $x_s = 1$, then the j_1-th and j_2-th matrices of the product are J matrices and the others are I matrices. As J has order two, the product will always be the identity.

Bookend vectors. We take A_0 and $A_{\ell+1}$ to be two vectors such that $A_0 I A_{\ell+1} = 0$ but $A_0 J A_{\ell+1} \neq 0$. For instance, with the choice of $J = \begin{pmatrix} 0\ 1 & & \\ 1\ 0 & & \\ & & I_{w-2} \end{pmatrix}$, we can take $A_0 = \begin{pmatrix} 1\ 0 \dots\ 0 \end{pmatrix}$ and $A_{\ell+1} = \begin{pmatrix} 0\ 1\ 0 \dots\ 0 \end{pmatrix}^T$, where A^T denotes the transpose of A for any matrix A. These bookend vectors are the same for both branching programs, i.e., $A'_0 = A_0$ and $A'_{\ell+1} = A_{\ell+1}$.

These two branching programs $\mathbf{A}$ and $\mathbf{A}'$ are equivalent as they both compute the always zero function. Now, take $x = 0 \dots 0$ and $y = 0 \dots 010 \dots 0$ where the 1 is at the s-th position, and let $j = j_1$. Let us compute $\tilde{a}_{x,j}, \tilde{a}_{y,j}$ for branching program $\mathbf{A}$ and $\tilde{a}'_{x,j}, \tilde{a}'_{y,j}$ for branching program $\mathbf{A}'$.

Branching program A. As all matrices are identity matrices in $\mathbf{A}$, exchanging two matrices does not change the product and we still have

$$\tilde{a}_{x,j} = A_0 \cdot \prod_{i<j} A_{i,x[\texttt{inp}(i)]} \cdot A_{j,y[\texttt{inp}(j)]} \cdot \prod_{j<i\leq\ell} A_{i,x[\texttt{inp}(i)]} \cdot A_{\ell+1} = A_0 \cdot I \cdot A_{\ell+1} = 0,$$
$$\tilde{a}_{y,j} = A_0 \cdot \prod_{i<j} A_{i,y[\texttt{inp}(i)]} \cdot A_{j,x[\texttt{inp}(j)]} \cdot \prod_{j<i\leq\ell} A_{i,y[\texttt{inp}(i)]} \cdot A_{\ell+1} = A_0 \cdot I \cdot A_{\ell+1} = 0.$$

Branching program $\mathbf{A}'$. Here, we chose our parameters so that an honest evaluation of $\mathbf{A}'$ on x leads to a product of only I matrices and an honest evaluation of $\mathbf{A}'$ on y leads to a product of $\ell - 2$ matrices I and 2 matrices J. We also chose j so that we exchange a J matrix with a I matrix. Hence, we have

$$\tilde{a}'_{x,j} = A'_0 \cdot \prod_{i<j} A'_{i,x[\texttt{inp}(i)]} \cdot A'_{j,y[\texttt{inp}(j)]} \cdot \prod_{j<i\leq\ell} A'_{i,x[\texttt{inp}(i)]} \cdot A'_{\ell+1} = A_0 \cdot J \cdot A_{\ell+1} \neq 0,$$
$$\tilde{a}'_{y,j} = A'_0 \cdot \prod_{i<j} A'_{i,y[\texttt{inp}(i)]} \cdot A'_{j,x[\texttt{inp}(j)]} \cdot \prod_{j<i\leq\ell} A'_{i,y[\texttt{inp}(i)]} \cdot A'_{\ell+1} = A_0 \cdot J \cdot A_{\ell+1} \neq 0.$$

To conclude, this gives us the desired condition of Section 4.2. Indeed, for the branching program $\mathbf{A}$, the numerator of $\tilde{u}_{x,j} \cdot \tilde{u}_{y,j}$ is a multiple of g^2, hence zero-testing it with the parameter p'_{zt} gives a positive result. Oppositely, for the branching program $\mathbf{A}'$, the numerator of $\tilde{u}_{x,j} \cdot \tilde{u}_{y,j}$ is not a multiple of g,[17] hence zero-testing it with the parameter p'_{zt} gives a negative result. We can then distinguish between the obfuscations of $\mathbf{A}$ and $\mathbf{A}'$. This completes the proof of Theorem 1 for the obfuscators of [2, 5, 23, 30, 32].

4.4 Other branching program obfuscators

We now discuss the possible extension of this attack to other branching program obfuscators that are not captured by the abstract obfuscator of Section 3.

Obfuscator of [6]. This obfuscator is close to the one described in the abstract model, except that it obfuscates a slightly different definition of branching programs. In [6], a branching program $\mathbf{A}$ comes with an additional value q_{acc}, and we have $\mathbf{A}(x) = 0$ if and only if $A_0 \cdot \prod_{i\in[l]} A_{i,x[\mathtt{inp}(i)]} \cdot A_{\ell+1} = q_{acc}$. The only difference with the definition of branching programs given in Section 2.3 is that q_{acc} may be non-zero. Hence, when multiplying by the scalars $\alpha_{i,\mathbf{b}}$ in the obfuscator (see Tool 3), we may change the output of the function. To enable correct evaluation of the obfuscated branching program, the obfuscator of [6] also publishes encodings of the scalars $\alpha_{i,\mathbf{b}}$ at level $S_{i,\mathbf{b}}$.

More formally, the obfuscator of [6] uses Tools 2 and 3 of Section 3. In Tool 2, the authors use R_{i+1}^{-1} instead of R_{i+1}^{adj}, in order to keep the same product (otherwise the product would be multiplied by the determinants of the R_i matrices). Let $\hat{A}_{i,\mathbf{b}} = \alpha_{i,\mathbf{b}} R_i A_{i,\mathbf{b}} R_{i+1}^{-1}$ be the matrices obtained after re-randomization (using Tools 2 and 3). Let $\hat{A}_0 = A_0 R_1^{-1}$ and $\hat{A}_{\ell+1} = R_{\ell+1} A_{\ell+1}$. The obfuscator provides encodings of the matrices $\hat{A}_0$, $\{\hat{A}_{i,\mathbf{b}}\}_{i,\mathbf{b}}$ and $\hat{A}_{\ell+1}$ at levels S_0, $\{S_{i,\mathbf{b}}\}_{i,\mathbf{b}}$ and $S_{\ell+1}$, respectively. It also provides encodings of the $\{\alpha_{i,\mathbf{b}}\}_{i,\mathbf{b}}$ at levels $\{S_{i,\mathbf{b}}\}_{i,\mathbf{b}}$ and an encoding of q_{acc} at level $S_0 \cup S_{\ell+1}$. Then, to evaluate the obfuscated branching program on input x, one computes

$$[\hat{A}_0]_{S_0} \cdot \prod_{i\in[\ell]} [\hat{A}_{i,x[\mathtt{inp}(i)]}]_{S_{i,x[\mathtt{inp}(i)]}} \cdot [\hat{A}_{\ell+1}]_{S_{\ell+1}} - [q_{acc}]_{S_0 \cup S_{\ell+1}} \cdot \prod_{i\in[\ell]} [\alpha_{i,x[\mathtt{inp}(i)]}]_{S_{i,x[\mathtt{inp}(i)]}},$$

and tests whether this is an encoding of 0 or not. By construction, this will be an encoding of 0 at level S_{zt} if and only if $\mathbf{A}(x) = 0$.

The first part of our attack (recovering h and p'_{zt}) still goes through. We slightly modify the mixed-input part. Instead of exchanging only the j-th matrix between the evaluations of x and y, we will also exchange the corresponding $\alpha_{j,\mathbf{b}}$ in the second product. Doing so, we ensure that the product of the $\alpha_{i,\mathbf{b}}$'s remains the same in both sides of the difference. This also ensures that the level of both sides will be the same after the exchange, and hence we can still subtract them.

[17] The ideal $\langle g \rangle$ is chosen prime in GGH so the product of two elements that are not divisible by g is also not divisible by g.

The same example as in Section 4.3 will then work also for this obfuscator. This gives us a way to distinguish in quantum polynomial time between the obfuscated versions of two equivalent branching programs, hence attacking the iO security of the obfuscator of [6].

Obfuscators of [12,22]. Our attack does not seem to extend to the obfuscators of [12, 22]. The obstacle is that the security of these obfuscators against mixed-input attacks does not rely on the GGH map but on the scalars $\alpha_{i,\mathbf{b}}$, which are chosen with a specific structure to ensure that the branching program is correctly evaluated.

More precisely, these obfuscators use (single input) branching programs with a slightly different definition, where the product of matrices (with the bookend vectors) is never 0. For instance, the branching programs are chosen such that the product of the matrices (on honest evaluations) is either 1 or 2, in which cases we say that the output of the branching program is respectively 0 or 1. Hence, when evaluating the obfuscated branching program on input x, the user obtains a top-level encoding of either $\prod_i \alpha_{i,x_i}$ or $2\prod_i \alpha_{i,x_i}$ depending on the output of the branching program. In order for the user to determine which one of the two encodings it has obtained, the obfuscated branching program also provide him (via a so-called dummy branching program) with a top-level encoding $\prod_i \alpha_{i,x_i}$. The user then only has to subtract the two top-level encodings and zero-test to determine whether $\mathbf{A}(x) = 0$ or 1. Now, if the user tries to mix the inputs, it can obtain a top-level encoding of $(\alpha_{j,y_j} \cdot \prod_{i\neq j} \alpha_{i,x_i}) \cdot a_{x,j}$ for instance (where $a_{x,j} = 1$ or 2 is the product of the corresponding matrices). But, as it is not an honest evaluation, it will not have a top-level encoding of $\alpha_{j,y_j} \cdot \prod_{i\neq j} \alpha_{i,x_i}$ to compare it with.

Following the same idea as for the mixed-input attack described above, the attacker could compute two top-level encodings of $(\alpha_{j,y_j} \cdot \prod_{i\neq j} \alpha_{i,x_i}) \cdot a_{x,j}$ and $(\alpha_{j,x_j} \cdot \prod_{i\neq j} \alpha_{i,y_i}) \cdot a_{y,j}$ and then multiply them to obtain an encoding of $(\prod_i \alpha_{i,x_i} \cdot \prod_i \alpha_{i,y_i}) \cdot a_{x,j} \cdot a_{y,j}$ at level $2S_{zt}$. Now, using the top-level encodings of $\prod_i \alpha_{i,x_i}$ and $\prod_i \alpha_{i,y_i}$ that are provided by the obfuscated branching program, one can also obtain an encoding of $(\prod_i \alpha_{i,x_i} \cdot \prod_i \alpha_{i,y_i})$ at level $2S_{zt}$. So if we could zero-test at level $2S_{zt}$, then we could distinguish between a branching program where $a_{x,j} \cdot a_{y,j} = 1$ and one where $a_{x,j} \cdot a_{y,j} \neq 1$. But we cannot zero-test at level $2S_{zt}$: our new zero-testing parameter p'_{zt} only enables us to determine whether the numerator of an encoding is a multiple of g^2 or not. Here, we subtract two level-$2S_{zt}$ encodings of the same value, so the numerator of the result will be a multiple of g, but it is very unlikely to be a multiple of g^2. Hence, we do not learn anything by using p'_{zt}. Because of the final subtraction, we did not manage to obtain an encoding at level $2S_{zt}$ whose numerator was a multiple of g^2, and so we did not manage to adapt the mixed-input attack described above to the obfuscators of [12, 22].

5 Adapting the attack to the obfuscator of [19]

Unlike the abstract branching program described in Section 3, the obfuscator of [19] does not obfuscate branching programs, but it obfuscates circuits directly. The structure of this obfuscator is very different from the abstract obfuscator described in Section 3 and so the attack described in Section 4 cannot be directly applied to it. However, similarly to the other obfuscators described above, the obfuscator of [19] also uses the levels of the GGH multilinear map to prevent mixed-input attacks. This is the weakness we exploited to mount a mixed-input attack against the abstract obfuscator, and here again, this will enable us to attack the [19] obfuscator, by attacking the underlying GGH multilinear map. In this section, we first describe in a simplified way the obfuscator of [19] (this simplified version also captures the obfuscators of [4, 37]). We then show how to adapt our attack to mount a quantum polynomial-time mixed-input attack against this candidate obfuscator.

5.1 The obfuscator

The obfuscator of [19] uses the GGH multilinear map [21] in its asymmetric version, but with a composite g. More concretely, sample three elements $g_1, g_2, g_3 \in R$ as for the original g in the GGH map, that is $\|g_i\| = O(n)$, $\|1/g_i\| = O(n^2)$ and such that $\mathcal{N}(g_i)$ is a prime integer, for all $i \in [3]$. Then, let $g = g_1 g_2 g_3$. If we denote by $R_i = R/g_i R$ the quotient rings for $i \in [3]$, then using the Chinese reminder theorem we know that the encoding space R/gR is isomorphic to $R_1 \times R_2 \times R_3$. In the following, it will be useful to choose this point of view, as we will encode triplets of elements $(a_1, a_2, a_3) \in R_1 \times R_2 \times R_3$, using the GGH map.

Let Σ be some subset of $\{0,1\}^l$ with both l and $|\Sigma|$ that are polynomial in the security parameter λ. We will be interested into arithmetic circuits $C : \Sigma \to \{0,1\}$. By arithmetic circuits, we mean that C performs addition, multiplication and subtraction over the bits of the element of Σ (i.e., C is an arithmetic circuit from $\{0,1\}^l$ to $\{0,1\}$, but we are only interested in its restriction to $\Sigma \subseteq \{0,1\}^l$). The operations over the bits are performed over $\mathbb{Z}$ but we only consider circuits whose output is in $\{0,1\}$. Let $\mathscr{C}$ be a class of such circuits, whose size is bounded by some polynomial (the properties of this class of circuit will not be interesting for our attack) and let U be a universal circuit for the class $\mathscr{C}$. The size of U is also bounded by some polynomial in the security parameter. We abuse notation by denoting by C both a circuit of $\mathscr{C}$ and its bit representation, that is we have $U(\sigma, C) = C(\sigma)$ for any $\sigma \in \Sigma$ (the first C denotes the bit representation of the circuit while the second one represent the function computed by the circuit).

To obfuscate a circuit C of the class $\mathscr{C}$, the main idea of [19] is that the obfuscator will produce GGH encodings of the bits of C and of the bits of all the possible inputs $\sigma \in \Sigma$. Then, to evaluate the obfuscated circuit, it suffices to homomorphically evaluate the universal circuit U on these encodings and to test whether the result is 0 or not. In order to prove the security of their obfuscators, the authors of [19] added other gadgets to their obfuscator. The first idea is

to encode the useful information only in the second slot of the GGH map (in the ring R_2) and to use the two other slots to prevent some mixed-input attack (where we mix the bits of two circuits). They also use straddling set systems, like the abstract obfuscator defined in Section 3, to prevent other kind of mixed input attacks (where we mix the bits of two inputs). We describe below in more details how the obfuscator of [19] obfuscates a circuit $C \in \mathscr{C}$. In order to help understanding what is happening, we also describe in parallel how to evaluate the obfuscated circuit.

1. First, we encode each bit of all the possible inputs $\sigma \in \Sigma$ (recall that we chose $|\Sigma|$ to be polynomial in the security parameter, so it is possible to enumerate all the elements of Σ). For each symbol $\sigma \in \Sigma$ and each bit position $i \in [l]$, define $W_{i,\sigma}^{(1)} = [r_\sigma^{(1)} \cdot w_{i,\sigma}^{(1)}]_{S_\sigma^{(1)}}$ and $R_\sigma^{(1)} = [r_\sigma^{(1)}]_{S_\sigma^{(1)}}$, where $r_\sigma^{(1)}$ is sampled uniformly in $R/gR^\times$ (and only depends on σ) and

$$w_{i,\sigma}^{(1)} = (y_i^{(1)}, \sigma_i, \rho_{i,\sigma}^{(1)}) \in R_1 \times R_2 \times R_3,$$

   for σ_i the i-th bit of σ and $y_i^{(1)}$ and $\rho_{i,\sigma}^{(1)}$ sampled uniformly in R_1 and R_3 respectively. The level $S_\sigma^{(1)}$ of the encoding will be chosen to prevent mixed-input attacks. We will go into more details about the levels of the encodings later. These encodings $W_{i,\sigma}^{(1)}$ and $R_\sigma^{(1)}$ are made public, for $i \in [l]$ and $\sigma \in \Sigma$. Note that $y_i^{(1)}$ is the same for all symbols σ, this will be necessary for correctness.
2. Second, we encode the bits of the representation of the circuit $C \in \mathscr{C}$. We denote by $|C|$ the size of the bit representation of C. For each $1 \leq j \leq |C|$, define $W_j^{(2)} = [r^{(2)} \cdot w_j^{(2)}]_{S^{(2)}}$ and $R^{(2)} = [r^{(2)}]_{S^{(2)}}$, where $r^{(2)}$ is sampled uniformly in $R/gR^\times$ and

$$w_j^{(2)} = (y_j^{(2)}, C_j, \rho_j^{(2)}) \in R_1 \times R_2 \times R_3,$$

   for C_j the j-th bit of the representation of C and $y_j^{(2)}$ and $\rho_j^{(2)}$ sampled uniformly in R_1 and R_3 respectively. Again, the level $S^{(2)}$ of the encoding will be described later. These encodings $W_j^{(2)}$ and $R^{(2)}$ are made public, for $1 \leq j \leq |C|$.

Once we have encodings for the bits of C and for all the possible input values $\sigma \in \Sigma$, as the universal circuit U only performs additions, subtractions and multiplications, we can homomorphically evaluate it on the encodings. We can always perform multiplications of encodings, it will only increase the level of the encodings. However, there is a subtlety for addition and subtraction, as we can only add and subtract encodings at the same level. To circumvent this difficulty, the authors of [19] use the encodings $R^{(2)}$ and $R_\sigma^{(1)}$. During the evaluation of the universal circuit U on the encodings, we will perform computations so that for all intermediate encodings we compute, we always have encodings of the form $[r \cdot w]_S$

and $[r]_S$, with the same level S. At the beginning, all the encodings described above have the desired form $[r \cdot w]_S$ and $[r]_S$. If we want to multiply $[r_1 \cdot w_1]_{S_1}$ and $[r_2 \cdot w_2]_{S_2}$, we just compute the product of the encodings to get $[r_1 r_2 \cdot w_1 w_2]_{S_1 \cup S_2}$ and we also compute the product of the r part to obtain $[r_1 r_2]_{S_1 \cup S_2}$. Note that here, the union of the two sets $S_1 \cup S_2$ keeps multiple copies of the elements that appear both in S_1 and in S_2 (i.e., $S_1 \cup S_2$ is a multiset). If we want to add $[r_1 \cdot w_1]_{S_1}$ and $[r_2 \cdot w_2]_{S_2}$, then two cases appear. If $r_1 = r_2$ and $S_1 = S_2$, then add both encodings to get $[r_1 \cdot (w_1 + w_2)]_{S_1}$ and keep $[r_1]_{S_1}$. Otherwise, compute $[r_1]_{S_1} \cdot [r_2 \cdot w_2]_{S_2} + [r_2]_{S_2} \cdot [r_1 \cdot w_1]_{S_1} = [r_1 r_2 \cdot (w_1 + w_2)]_{S_1 \cup S_2}$ and compute the product $[r_1 r_2]_{S_1 \cup S_2}$. We proceed similarly for subtraction.

With this technique, we can evaluate the circuit U on the encodings provided by the obfuscator, independently of the levels used to encode them. Assume we evaluate it honestly on the encodings of C and of some input $\sigma \in \Sigma$, we then obtain encodings $W_\sigma = [r_\sigma \cdot w_\sigma]_{S_\sigma}$ and $R_\sigma = [r_\sigma]_{S_\sigma}$ at some level S_σ, for some $r_\sigma \in R/gR$, where

$$w_\sigma = (y^*, C(\sigma), \rho_\sigma) \in R_1 \times R_2 \times R_3,$$

for some $y^* \in R_1$ and $\rho_\sigma \in R_3$. Note that, as the $y_i^{(1)}$'s do not depend on the input σ, the value y^* is the same for all σ's. We then want to annihilate the values in the extra slots (that is y^* and ρ_σ) to recover the value of $C(\sigma)$ by zero-testing. To do that, the obfuscator provides two more encodings.

3. To annihilate the value in the third slot, the obfuscator output encodings $\widehat{W_\sigma} = [\widehat{r_\sigma} \cdot \widehat{w}]_{\widehat{S_\sigma}}$ and $\widehat{R_\sigma} = [\widehat{r_\sigma}]_{\widehat{S_\sigma}}$, for all $\sigma \in \Sigma$, where $\widehat{r_\sigma}$ is sampled uniformly in $R/gR^\times$ and
$$\widehat{w} = (\widehat{y}, \widehat{\alpha}, 0),$$
for $\widehat{y}$ and $\widehat{\alpha}$ uniformly chosen in R_1 and $R_2^\times$, respectively.

Multiplying the encoding of $w_\sigma = (y^*, C(\sigma), \rho_\sigma)$ obtained above, by this encoding of $\widehat{w} = (\widehat{y}, \widehat{\alpha}, 0)$ enables us to cancel the last slot and to obtain an encoding of $\widehat{w}_\sigma := (\widehat{y} \cdot y^*, \widehat{\alpha} \cdot C(\sigma), 0)$. We also multiply the r parts, as described above. Note that to cancel this third slot, the obfuscator outputs one pair of encodings for each symbol $\sigma \in \Sigma$. While this may seem useless because each encoding encodes the same $\widehat{w}$, this is in fact required to standardise the levels of the encodings. Indeed, after evaluating the universal circuit on the encodings of C and σ, we obtain an encoding whose level depends on σ. By multiplying with an encoding at a complementary level at this step, we can then ensure that the level of the product is independent of σ. This property will be important, because to zero-test the final encoding, we need it to be at the maximal level S_{zt}, independently of the input σ.

4. Finally, to cancel the first slot, the obfuscator provides two encodings $\bar{W} = [\bar{r} \cdot \bar{w}]_{\bar{S}}$ and $\bar{R} = [\bar{r}]_{\bar{S}}$, where $\bar{r}$ is sampled uniformly in $R/gR^\times$ and
$$\bar{w} = (\widehat{y} \cdot y^*, 0, 0).$$

Note that $\widehat{w}_\sigma - \bar{w} = 0$ if and only if $C(\sigma) = 0$. Hence, it suffices to subtract the corresponding encodings (using the r part, because the levels of the encodings will not match) and to zero-test the obtained encoding to determine whether $C(\sigma) = 0$ or 1.

This completes the description of the obfuscator, together with the correctness proof of the evaluation of the obfuscated program. Before describing the mixed-input attack, we would like to insist on some properties of the obfuscator described above.

- The levels of the encodings output by the obfuscator are chosen such that all honest evaluations of the obfuscated circuit on some input $\sigma \in \Sigma$ produce encodings with the same level. This level is then chosen to be the maximal level of the GGH map, and will be denoted by S_{zt}. The obfuscator also provides a zero-test parameter p_{zt} to enable zero-test at level S_{zt}. In the following, the only thing that will be interesting for our attack is that a honest evaluation of the obfuscated circuit on any input $\sigma \in \Sigma$ outputs an encoding at level S_{zt}, so we do not go into more details about the levels of the encodings.
- As we already noted, the value y^* obtained in the first slot after evaluating the universal circuit on the encodings of C and σ does not depend on σ. This is needed for the last step, where we subtract $\widehat{y} \cdot y^*$. As we want this to output 0 for any input (to cancel out the first slot), the value y^* has to be independent of σ. This first slot prevents us from mixing the bits of the circuit C, but does not prevent us from mixing the bits of the input σ (i.e., changing the value of some bit during the evaluation). Mixing the bits of the input is only prevented by the GGH map and the straddling set system (recall that the levels of the encodings depend on the input σ). This is the kind on mixed-input attack we will be able to perform after recovering the secret element h of the GGH map.

Differences between the DGGMM obfuscator and our simplification above. The obfuscator of [19] obfuscates circuits from Σ^c to $\{0,1\}$ for some constant c, instead of circuits from Σ to $\{0,1\}$ as described above. However, for our attack, we can take the constant c to be equal to 1, so we simplified a bit the description of the obfuscator and forgot about this constant c. If needed, the attack can be easily adapted to the case where c is a constant different from 1.

Also, the obfuscator of [19] uses an extra slot where it computes a PRF, and which is cancelled out before zero-testing by multiplying by an encoding of 0 in this slot (the principle is the same as for cancelling the third slot of the obfuscator described here). This extra slot is used only in the proof of security and does not interfere with our mixed-input attack, so we removed it from the description above.[18]

[18] This extra slot can be captured by the simplification above by taking g_3 to be a product of two prime elements and changing the distribution of the elements ρ in the third slot of the encodings. This has no impact on our attack.

Finally, in the obfuscator of [19], we have $\bar{w} = (\widehat{y} \cdot y^*, \widehat{\alpha}, 0)$ instead of $\bar{w} = (\widehat{y} \cdot y^*, 0, 0)$. So when subtracting, we obtain at the end an encoding of $(0, \hat{\alpha}(1 - C(\sigma)), 0)$, which is 0 if and only if $C(\sigma) = 1$, instead of 0 if and only if $C(\sigma) = 1$ as in our simplification. However, both versions are equivalent, as we can always negate the output of the circuit. In order to be consistent with the other obfuscators described in this article, we decided to stick with the fact that obtaining an encoding of 0 means that the circuit outputs 0.

The DGGMM obfuscator was designed to obtain a candidate iO obfuscator from low noise multilinear maps. To do so, the class of circuit $\mathscr{C}$ targeted by the obfuscator described above is a very restrictive one (among other things, it requires that the circuits have a constant depth and a polynomial number of inputs). The authors then use a theorem from [29] to bootstrap their construction for this restricted class of circuit $\mathscr{C}$ to an obfuscator for all circuits in P/poly.

Remark. The DGGMM obfuscator is very similar to the previous circuit obfuscators of [4,37], and the simple circuit obfuscator described above also captures these obfuscators. Hence, the attack described below also applies to the obfuscators of [4,37], when instantiated with the GGH13 multilinear map (these obfuscators were originally instantiated with the CLT multilinear map, as they require composite-order multilinear maps, but they can also be instantiated with a modified version of the GGH13 map, as observed in [19]).

5.2 The mixed-input attack

As mentioned above, the attack will consist in modifying a bit of the input σ during the computation. The idea is the same as for the attack of Section 4. We start by recovering the secret element h of the GGH map in quantum polynomial time, using the works of [10,13,18]. As above, we can obtain top level encodings of 0 each time the circuit evaluates to 0, so by choosing a circuit that evaluates to 0 sufficiently often, we can recover a basis of the ideal $\langle h \rangle$ (under Heuristic 1) and then recover h exactly (under Conjecture 1). We then construct a new zero-testing parameter p'_{zt} at level $2S_{zt}$ (testing whether the numerator of an encoding is a multiple of g^2, and not only g). This first step of the attack works exactly as described in Section 4.1 and we do not re-explain it here.

The second part of the attack (using p'_{zt} to mount a mixed input attack) will differ from the one for the abstract branching program obfuscator. The first difference is that in the abstract branching program obfuscator, we only computed products of matrices. So by changing a matrix, we just changed the final level of the encodings but all the operations remained possible (products of encodings are always possible, whatever their levels are). Here, as we evaluate a circuit with additions and multiplications, we must be careful. Indeed, if we change the level of one encoding of a sum but not the other one, we will not be able to perform the sum anymore. To circumvent this difficulty, we will use a specific universal circuit, which ends up by a multiplication. Let U be a universal circuit for the class of circuit $\mathscr{C}$. We define a new circuit $\tilde{U}$, which takes as input a concatenation of the description of two circuits in $\mathscr{C}$ and an input $\sigma \in \Sigma$ and

computes the product of the evaluations of the two circuits on input σ. More formally, we define

$$\tilde{U}(\sigma, C_1 \cdot C_2) = U(\sigma, C_1) \cdot U(\sigma, C_2).$$

The circuit $\tilde{U}$ is a universal circuit for the class $\mathscr{C} \cdot \mathscr{C}$. Note that when evaluating the circuit $\tilde{U}$, we finish the evaluation with a multiplication. To perform our mixed input attack, we will evaluate $U(\cdot, C_1)$ and $U(\cdot, C_2)$ honestly on different inputs σ_1 and σ_2. As each partial evaluation is honest, we can perform all the required operations on the encodings. The dishonest computation will be the last multiplication only.

Let σ_1 and σ_2 be two distinct elements of Σ. Let C_{00} be a circuit that evaluates always to 0 on Σ. We also let C_{10} be a circuit that evaluates to 1 on σ_1 and to 0 otherwise and C_{01} be a circuit that evaluates to 1 on σ_2 and to 0 otherwise. The functions computed by $C_{00} \cdot C_{00}$ and by $C_{01} \cdot C_{10}$ are the same, so these circuits are equivalent. We will now show how to distinguish the obfuscated versions of $C_{00} \cdot C_{00}$ and $C_{01} \cdot C_{10}$, when using the universal circuit $\tilde{U}$. As both circuits are equivalent, this will result into an attack against the iO security of the obfuscator.

Objective: The obfuscator obfuscates the circuit $C_1 \cdot C_2 \in \{C_{00} \cdot C_{00}, C_{01} \cdot C_{10}\}$, and we want to distinguish whether $C_1 \cdot C_2 = C_{00} \cdot C_{00}$ or $C_1 \cdot C_2 = C_{01} \cdot C_{10}$.

1. The obfuscator encodes the bits of C_1 and C_2 under the GGH map, as well as the bits of all possible inputs $\sigma \in \Sigma$. In particular, we have encodings for σ_1 and σ_2. We homomorphically evaluate U on the encodings of C_1 and σ_1, C_1 and σ_2, C_2 and σ_1 and C_2 and σ_2.[19] These are honest partial evaluations of the circuit $\tilde{U}$ on input σ_1 and σ_2, so we can perform these evaluations (in particular, there will not be incompatibilities of encodings levels). We obtain four pairs of encodings $(R_{b_1b_2} = [r_{b_1b_2}]_{S_{b_1b_2}}, W_{b_1b_2} = [r_{b_1b_2} \cdot w_{b_1b_2}]_{S_{b_1b_2}})$, for $b_1, b_2 \in \{1, 2\}^2$, where

$$w_{b_1b_2} = (y_{b_1}, C_{b_1}(\sigma_{b_2}), \rho_{b_1b_2}).$$

   Recall that the y part of the encoding does not depend on the input σ, so this is independent of b_2 for our notations.
2. A honest evaluator of the obfuscated program would then multiply the encodings W_{11} and W_{21} (of $C_1(\sigma_1)$ and $C_2(\sigma_1)$) and the encodings W_{12} and W_{22} (of $C_1(\sigma_2)$ and $C_2(\sigma_2)$). However, in order to distinguish which circuit has been obfuscated, we do not perform these honest computations. Instead, following the idea of the mixed input attack described in Section 4.2, we compute $W_{11} \cdot W_{22}$ and $W_{12} \cdot W_{21}$ (and we do the same for the r part). We then obtain two encodings $\widetilde{W}_1$ and $\widetilde{W}_2$ of

$$\tilde{w}_1 := (y^*, C_1(\sigma_1) \cdot C_2(\sigma_2), \rho_{11}\rho_{22})$$
$$\text{and} \quad \tilde{w}_2 := (y^*, C_1(\sigma_2) \cdot C_2(\sigma_1), \rho_{12}\rho_{21})$$

[19] Recall that $U(\sigma, C) = C(\sigma)$ and the universal circuit we chose is $\tilde{U}(\sigma, C_1 \cdot C_2) = U(\sigma, C_1) \cdot U(\sigma, C_2)$.

at levels $S_{11} \cup S_{22}$ and $S_{12} \cup S_{21}$ respectively. Note that the first slot of the encodings contains y^*, as it would for a honest evaluation.

3. We then complete the computation as if $\widetilde{W}_1$ was an honest evaluation on σ_1 and $\widetilde{W}_2$ was an honest evaluation on σ_2. That is, we first multiply $\widetilde{W}_1$ by $\widehat{W_{\sigma_1}}$ and $\widetilde{W}_2$ by $\widehat{W_{\sigma_2}}$ to cancel the third slot. We obtain two encodings $\widehat{W}_1$ and $\widehat{W}_2$ of

$$\begin{aligned} \widehat{w}_1 &:= (y^* \cdot \widehat{y}, \widehat{\alpha} \cdot C_1(\sigma_1) \cdot C_2(\sigma_2), 0) \\ \text{and} \quad \widehat{w}_2 &:= (y^* \cdot \widehat{y}, \widehat{\alpha} \cdot C_1(\sigma_2) \cdot C_2(\sigma_1), 0) \end{aligned}$$

at levels $S_{11} \cup S_{22} \cup \widehat{S_{\sigma_1}}$ and $S_{12} \cup S_{21} \cup \widehat{S_{\sigma_2}}$, respectively.

4. Finally, we cancel the first slot by subtracting $\bar{W}$ to the encodings $\widehat{W}_1$ and $\widehat{W}_2$ obtained above. Note that this subtraction is between encodings that are not at the same level (for both honest and dishonest evaluations), so the resulting level is the union of the levels of both parts of the subtraction. We obtain two encodings $\bar{W}_1$ and $\bar{W}_2$ of

$$\begin{aligned} \bar{w}_1 &:= (0, \widehat{\alpha} \cdot C_1(\sigma_1) \cdot C_2(\sigma_2), 0) \\ \text{and} \quad \bar{w}_2 &:= (0, \widehat{\alpha} \cdot C_1(\sigma_2) \cdot C_2(\sigma_1), 0) \end{aligned}$$

at levels $S_{11} \cup S_{22} \cup \widehat{S_{\sigma_1}} \cup \bar{S}$ and $S_{12} \cup S_{21} \cup \widehat{S_{\sigma_2}} \cup \bar{S}$, respectively.

5. Now, we would like to zero-test the encodings $\bar{W}_1$ and $\bar{W}_2$ obtained above, but because we mixed the inputs, the levels of the encodings are unlikely to be S_{zt} and we are not able to zero-test. However, we know that $S_{11} \cup S_{21} \cup \widehat{S_{\sigma_1}} \cup \bar{S} = S_{zt}$, because the encoding obtained by honestly evaluating the obfuscated program on σ_1 has this level. In the same way, we know that $S_{12} \cup S_{22} \cup \widehat{S_{\sigma_2}} \cup \bar{S} = S_{zt}$. Hence, the level of the product $\bar{W}_1 \cdot \bar{W}_2$ is $2S_{zt}$. Using our p'_{zt} parameter, we can then test whether its numerator is a multiple of g^2 or not.
   - In the case where $C_1 \cdot C_2 = C_{00} \cdot C_{00}$, we have $\bar{w}_1 = 0 \bmod g$ and $\bar{w}_2 = 0 \bmod g$. Hence, their product is a multiple of g^2. So the numerator of $\bar{W}_1 \cdot \bar{W}_2$ is a multiple of g^2, and the zero-test using p'_{zt} answers positively.
   - In the case where $C_1 \cdot C_2 = C_{01} \cdot C_{10}$, we have $\bar{w}_1 = 0 \bmod g$ and $\bar{w}_2 \neq 0 \bmod g$. So the product is a multiple of g^2 if and only if $\bar{w}_1$ is a multiple of g^2, which is very unlikely ($\bar{w}_1$ is obtained by subtracting two values that are equal modulo g_1, so this is a multiple of g_1 but this is unlikely to be a multiple of g_1^2).[20] Hence, the numerator of $\bar{W}_1 \cdot \bar{W}_2$ will not be a multiple of g^2 (with high probability), and the zero-test using p'_{zt} will fail.

   We can then distinguish between the obfuscated versions of $C_{00} \cdot C_{00}$ and $C_{01} \cdot C_{10}$ in (classical) polynomial time, using our new zero-testing parameter p'_{zt} obtained in quantum polynomial time.

This completes our quantum attack against the obfuscators of [4,19,37] and the proof of Theorem 1.

[20] Note that even if $\bar{w}_1$ were a multiple of g^2, then, by taking $p'_{zt} = (z^* \cdot g^{-1})^3 \bmod q$, we could mount the same kind of attack, at level $3S_{zt}$ instead of $2S_{zt}$.

References

1. M. R. Albrecht, S. Bai, and L. Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. In *CRYPTO 2016*, pages 153–178. Springer, Heidelberg, Aug. 2016.
2. P. V. Ananth, D. Gupta, Y. Ishai, and A. Sahai. Optimizing obfuscation: Avoiding Barrington's theorem. In *ACM CCS 14: 21st Conference on Computer and Communications Security*, pages 646–658. ACM Press, Nov. 2014.
3. D. Apon, N. Döttling, S. Garg, and P. Mukherjee. Cryptanalysis of indistinguishability obfuscations of circuits over GGH13. In *International Colloquium on Automata, Languages, and Programming*. Springer, Berlin, Heidelberg, 2017.
4. B. Applebaum and Z. Brakerski. Obfuscating circuits via composite-order graded encoding. In *TCC 2015*, pages 528–556. Springer, Heidelberg, Mar. 2015.
5. S. Badrinarayanan, E. Miles, A. Sahai, and M. Zhandry. Post-zeroizing obfuscation: New mathematical tools, and the case of evasive circuits. In *EUROCRYPT 2016*, pages 764–791. Springer, Heidelberg, May 2016.
6. B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai. Protecting obfuscation against algebraic attacks. In *EUROCRYPT 2014*, pages 221–238. Springer, Heidelberg, May 2014.
7. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO 2001*, pages 1–18. Springer, Heidelberg, Aug. 2001.
8. D. A. M. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. In *18th Annual ACM Symposium on Theory of Computing*, pages 1–5. ACM Press, May 1986.
9. J.-F. Biasse, T. Espitau, P.-A. Fouque, A. Gélin, and P. Kirchner. Computing generator in cyclotomic integer rings. In *EUROCRYPT 2017*, pages 60–88. Springer, 2017.
10. J.-F. Biasse and F. Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 893–902. Society for Industrial and Applied Mathematics, 2016.
11. Z. Brakerski and G. N. Rothblum. Obfuscating conjunctions. In *CRYPTO 2013*, pages 416–434. Springer, Heidelberg, Aug. 2013.
12. Z. Brakerski and G. N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In *TCC 2014*, pages 1–25. Springer, Heidelberg, Feb. 2014.
13. P. Campbell, M. Groves, and D. Shepherd. Soliloquy: A cautionary tale. In *ETSI 2nd Quantum-Safe Crypto Workshop*, pages 1–9, 2014.
14. Y. Chen, C. Gentry, and S. Halevi. Cryptanalyses of candidate branching program obfuscators. In *EUROCRYPT 2017*, pages 278–307. Springer, 2017.
15. J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehlé. Cryptanalysis of the multilinear map over the integers. In *EUROCRYPT 2015*, pages 3–12. Springer, Heidelberg, Apr. 2015.
16. J. H. Cheon, J. Jeong, and C. Lee. An algorithm for ntru problems and cryptanalysis of the ggh multilinear map without a low-level encoding of zero. *LMS Journal of Computation and Mathematics*, 19(A):255–266, 2016.
17. J.-S. Coron, T. Lepoint, and M. Tibouchi. Practical multilinear maps over the integers. In *CRYPTO 2013*, pages 476–493. Springer, Heidelberg, Aug. 2013.

18. R. Cramer, L. Ducas, C. Peikert, and O. Regev. Recovering short generators of principal ideals in cyclotomic rings. In *EUROCRYPT 2016*, pages 559–585. Springer, Heidelberg, May 2016.
19. N. Döttling, S. Garg, D. Gupta, P. Miao, and P. Mukherjee. Obfuscation from low noise multilinear maps. Cryptology ePrint Archive, Report 2016/599, 2016. http://eprint.iacr.org/2016/599.
20. R. Fernando, P. Rasmussen, and A. Sahai. Preventing CLT attacks on obfuscation with linear overhead. In *ASIACRYPT 2017*, pages 242–271. Springer, Heidelberg, Dec. 2017.
21. S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In *EUROCRYPT 2013*, pages 1–17. Springer, Heidelberg, May 2013.
22. S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE, Oct. 2013.
23. S. Garg, E. Miles, P. Mukherjee, A. Sahai, A. Srinivasan, and M. Zhandry. Secure obfuscation in a weak multilinear map model. In *TCC 2016-B*, pages 241–268. Springer, Heidelberg, Oct. / Nov. 2016.
24. C. Gentry, S. Gorbunov, and S. Halevi. Graph-induced multilinear maps from lattices. In *TCC 2015*, pages 498–527. Springer, Heidelberg, Mar. 2015.
25. R. Goyal, V. Koppula, and B. Waters. Lockable obfuscation. In *FOCS 2017*, pages 612–621. IEEE, 2017.
26. Y. Hu and H. Jia. Cryptanalysis of GGH map. In *EUROCRYPT 2016*, pages 537–565. Springer, Heidelberg, May 2016.
27. P. Kirchner and P.-A. Fouque. Revisiting lattice attacks on overstretched ntru parameters. In *EUROCRYPT 2017*, pages 3–26. Springer, 2017.
28. A. Langlois, D. Stehlé, and R. Steinfeld. GGHLite: More efficient multilinear maps from ideal lattices. In *EUROCRYPT 2014*, pages 239–256. Springer, Heidelberg, May 2014.
29. H. Lin. Indistinguishability obfuscation from constant-degree graded encoding schemes. In *EUROCRYPT 2016*, pages 28–57. Springer, Heidelberg, May 2016.
30. E. Miles, A. Sahai, and M. Weiss. Protecting obfuscation against arithmetic attacks. Cryptology ePrint Archive, Report 2014/878, 2014. http://eprint.iacr.org/2014/878.
31. E. Miles, A. Sahai, and M. Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In *CRYPTO 2016*, pages 629–658. Springer, Heidelberg, Aug. 2016.
32. R. Pass, K. Seth, and S. Telang. Indistinguishability obfuscation from semantically-secure multilinear encodings. In *CRYPTO 2014*, pages 500–517. Springer, Heidelberg, Aug. 2014.
33. A. Pellet-Mary. Quantum attacks against indistinguishablility obfuscators proved secure in the weak multilinear map model. Cryptology ePrint Archive, 2018. http://eprint.iacr.org/
34. A. Sahai and B. Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *46th Annual ACM Symposium on Theory of Computing*, pages 475–484. ACM Press, May / June 2014.
35. P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *FOCS 1994*, pages 124–134. IEEE, 1994.
36. D. Wichs and G. Zirdelis. Obfuscating compute-and-compare programs under LWE. In *FOCS 2017*, pages 600–611. IEEE, 2017.
37. J. Zimmerman. How to obfuscate programs directly. In *EUROCRYPT 2015*, pages 439–467. Springer, Heidelberg, Apr. 2015.