Implementing Token-Based Obfuscation under (Ring) LWE

Cheng Chen[1], Nicholas Genise[2], Daniele Micciancio[2], Yuriy Polyakov[3,4], and Kurt Rohloff[3]

[1] MIT CSAIL
[2] UCSD
[3] NJIT
[4] Duality Technologies, Inc.

May 15, 2019

Abstract. Token-based obfuscation (TBO) is an interactive approach to cryptographic program obfuscation that was proposed by Goldwasser et al. as a potentially more practical alternative to conventional non-interactive security models, such as Virtual Black Box (VBB) and Indistinguishability Obfuscation. We introduce a query-revealing variant of TBO, and implement in PALISADE several optimized query-revealing TBO constructions based on (Ring) LWE covering a relatively broad spectrum of capabilities: linear functions, conjunctions, and branching programs.

The linear-function construction is first proposed in our work, and can be used to efficiently obfuscate binary classifiers by utilizing the token-based model where the number and format of queries can be restricted by the token generator. We also present an optimized TBO implementation for conjunctions based on an existing construction for constrained-hiding constrained PRF, which outperforms the prior recent implementation of distributional VBB conjunction obfuscator by one order of magnitude and reduces the program size by a factor of 3. The token-based model also provides protection against exhaustive search attacks the VBB implementation is prone to. The last group of TBO constructions implemented in our work deals with obfuscating permutation and general branching programs.

To enable efficient implementation of all these constructions, we developed many algorithmic and code-level optimizations that can also be applied to other lattice-based cryptography primitives, including VBB and Indistinguishability Obfuscation.

Keywords: Lattice-Based Cryptography · Token-Based Program Obfuscation · Lattice Trapdoors · Residue Number Systems · Software Implementation

1 Introduction

Program obfuscation has long been of interest to the cyber-security community. Obfuscated programs need to be difficult (computationally hard) to reverse engineer, and have to protect intellectual property contained in software

from theft. For many years, practical program obfuscation techniques have been heuristic and have not provided secure approaches to obfuscation based on the computational hardness of mathematical problems. In this regard, there have been multiple recent attempts to develop cryptographically secure approaches to program obfuscation based on the computational hardness of mathematical problems (see [9] for a survey of these approaches). There are multiple definitions used for cryptographically secure program obfuscation. Two prominent definitions are Virtual Black Box and Indistinguishability Obfuscation.

Virtual Black Box (VBB) obfuscation is an intuitive definition of secure program obfuscation where the obfuscated program reveals nothing more than black-box access to the program via an oracle [36]. VBB is known to have strong limitations [10, 13, 33]. The most significant limitation is that *general-purpose* VBB obfuscation is unachievable [10].

To address limitations of VBB, Barak *et al.* [10] define a weaker security notion of *Indistinguishability Obfuscation* (IO) for general-purpose program obfuscation. IO requires that the obfuscations of any two circuits (programs) of the same size and same functionality (namely, the same truth table) are computationally indistinguishable. The IO concept has been of current interest, with recent advances to identify candidate IO constructions based on multi-linear maps [29, 43, 44, 45]. There has also been recent work to implement multi-linear map constructions [4, 19, 24, 37, 42]. Recent results show that these constructions might not be secure [21, 23]. The only IO construction supporting general functions that is not subject to any attack to date is the work by Garg *et al.* [26]. These cryptographically secure program obfuscation capabilities have also been considered impractical due to their computational and storage inefficiencies.

There have also been attempts to securely obfuscate under the VBB model (and its variants) certain *special-purpose* functions, such as point, conjunction, and evasive functions, using potentially practical techniques. For example, there have been several approaches to obfuscating point functions [6, 11, 25]. Unfortunately, point functions have limited applicability.

Both VBB and IO are *non-interactive* models of program obfuscation where the obfuscated program is made available to a computationally bound adversary. The adversary can then run a large number of queries (bounded only by its computational power) against the obfuscated program. In many practical scenarios, e.g., classification problems, the obfuscated program can be potentially learned by analyzing input-output maps.

An alternative approach to program obfuscation involves interactions with a trusted party, which allows one to build program obfuscation systems where the number of queries is limited by the trusted party. The two main models for *interactive* program obfuscation are *Trusted-Hardware Obfuscation* (THO) and *Token-Based Obfuscation* (TBO). In the THO model, the user first executes the obfuscated program for a given input and then interacts with a trusted hardware to obtain the decryption of the result [14, 35]. In the TBO model, the user obtains a special token before executing the obfuscated program and then finds the decrypted result by herself [34]. The latter model is more flexible and

can support the use cases where the tokens are pre-generated offline, i.e., the trusted hardware does not need to be accessible to the user.

To illustrate TBO, consider an application where a vendor obfuscates an arbitrary program and provides tokens representing the rights to run this program on specific inputs. When a specific user wants to input a query x to this program, she also gets a token for x from the program owner, and then executes the obfuscated program.

1.1 Our Contributions

Our work introduces a query-revealing variant of TBO (where input queries are in the clear), which is more efficient than the query-hiding TBO model proposed in [34] based on functional encryption/reusable garbled circuits. Our variant is adequate for most obfuscation scenarios as program inputs are typically not hidden. Query-revealing TBO thus provides an efficient method to obfuscate the classes of functions that can be learned. We develop optimized constructions, implement them in PALISADE, and report experimental results for the TBO of several types of programs, including linear functions (binary classifiers), conjunctions [18], permutation branching programs [18], and general branching programs [22]. The constructions for linear functions, conjunctions, and permutation branching programs are all secure under standard assumptions, either LWE or Ring LWE (RLWE). The performance results for binary classifiers and conjunctions suggest that they are already practical. Further, our constructions based on constraint-hiding constrained PRFs from [18] drop the security's dependence on small-entry $\mathbf{A}$ RLWE [15] and instead rely on standard RLWE for provable security.

Concretely, we present a new TBO scheme for linear functions, which is based on an LWE secret-key scheme. We show how the TBO of linear functions can be used to efficiently obfuscate a binary classifier. We implement the scheme in PALISADE, a general-purpose lattice cryptography library. Our performance evaluation results suggest that this implementation is already practical. For instance, our implementation can evaluate obfuscated binary classifiers in less than 1 millisecond and requires a program size of only 8MB for the case of 16 2-byte features. The scheme can also be inverted to efficiently compute weighted sums over encrypted data, where the weights are in the clear. Our analysis shows that the space and computational complexity of our scheme is asymptotically better than that of a prior inner-product functional encryption construction based on LWE [1].

We develop an efficient variant of the TBO of conjunctions based on the constraint-hiding constrained PRF construction presented in [18], and implement it in PALISADE. Our optimizations compared to the original scheme include significantly improved key generation and evaluation algorithms for the token generator (both runtime and storage requirements are reduced by more than one order of magnitude), much tighter correctness constraints, and a larger alphabet for encoding binary patterns. Our implementation also uses the Residue-Number-System (RNS) representation for all operations of the scheme, i.e., we

present a full RNS variant of the scheme, which works only with native integers and can be easily parallelized. Our performance results suggest that this implementation is faster by one order of magnitude and requires a 3x smaller program size, as compared to the prior recent distributional VBB conjunction obfuscation implementation [24].

We present an efficient (full RNS) variant of the TBO for permutation branching programs based on the constraint-hiding constrained PRF construction presented in [18], and implement it in PALISADE. The optimizations w.r.t. the original construction are similar to those for the TBO of conjunctions.

We develop an efficient ring (full RNS) variant of the LWE construction for the TBO of general branching programs proposed in [22] as an extension to the constrained-hiding constrained PRF construction for permutation branching programs, and implement it in PALISADE. The development of the ring variant of the TBO for general branching programs also required new algorithms for lattice trapdoor sampling and a hardness proof for the non-uniform Ring LWE problem, presented in the Appendix. Our construction of TBO for general branching programs can be proved secure under RLWE with an increase in the secret dimension and secret distribution width. However, we chose the small parameter regime (similar to the TBO for permutation branching programs) in our implementation for efficiency.

All our implementations of TBO constructions and lower-level lattice algorithms are added as modules to PALISADE, thus effectively providing a TBO toolkit that will be included in one of the next public releases of PALISADE. Note that we use the same general framework for the TBO of both permutation and general branching programs.

1.2 Related Work

The TBO construction in [34] is formulated for the case of hidden queries using reusable garbled circuits, which in their turn can be built on top of a functional encryption (FE) scheme. This implies that a TBO scheme can be derived from an FE scheme by treating a secret key for evaluating a specific function on encrypted data as a token.

General FE constructions are currently impractical. One approach is based on a combination of key-policy attribute-based encryption and fully homomorphic encryption [34]. The state-of-the-art results in key-policy attribute encryption [28] suggest these schemes are still inefficient, and hence their use in FE where each attribute bit needs to be encrypted with FHE is currently not practical. Initial experimental results for multi-input FE are presented in [19] but they are far from practical.

However, more efficient constructions exist for simpler functions, such as inner products. For instance, Agrawal *et al.* proposed an FE for inner product predicates as an extension of identity-based encryption using dual Regev's public key encryption scheme (based on LWE) and lattice trapdoor sampling. Another FE construction for inner product predicates builds directly on top of

dual Regev's public key encryption scheme [1] (our construction for inner products is asymptotically smaller and faster as shown later in this work). Several works considered the scenario of function-hiding inner product encryption where the result of inner product is computed while keeping both input vectors hidden [12, 39]. The experimental results for the scheme based on the Symmetric External Diffie-Hellman (SDXH) assumption for bilinear groups are presented in [39].

The main difference between query-revealing TBO used in our work and FE (TBO model in [34]) is that the input queries in our model are in the clear, just like in the non-interactive program obfuscation models. This enables more efficient constructions for TBO. For instance, our linear function (inner product) construction based on LWE is significantly faster (by orders of magnitude) than function-hiding inner product encryption based on SDXH in [39], as can be seen from comparing Table 2 in our work with Table 1 in [39].

2 Preliminaries

We denote the integers modulo q as $\mathbb{Z}_q := \mathbb{Z}/q\mathbb{Z}$. Our implementation utilizes power-of-two cyclotomic polynomial rings $\mathcal{R} = \mathbb{Z}[x]/\langle x^n + 1\rangle$ and $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$, where n is a power of 2 and q is an integer modulus. An element in the ring is represented via the coefficient embedding, or by its vector of coefficients. Importantly, we measure the norm of a ring element, or a vector of ring elements, through this coefficient embedding. The norm is the Euclidean norm unless stated otherwise.

The discrete Gaussian distribution over a lattice $\Lambda \subset \mathbb{R}^n$ is defined with probability mass proportional to $\rho_{\mathbf{c},\sigma}(\mathbf{x}) = e^{-\pi\|\mathbf{x}-\mathbf{c}\|^2/\sigma^2}$ and is denoted as $\mathcal{D}_{\Lambda,\mathbf{c},\sigma}$, where $\mathbf{c} \in \mathbb{R}^n$ is the center and σ is the distribution parameter. At the most primitive level, the discrete Gaussian sampling algorithms work with discrete Gaussian distribution $\mathcal{D}_{\mathbb{Z},c,\sigma}$ over integers with floating-point center c and distribution parameter σ. If the center c is omitted, it is assumed to be set to zero. When discrete Gaussian sampling is applied to cyclotomic rings, we denote discrete Gaussian distribution as $\mathcal{D}_{\mathcal{R},\sigma}$. In this work, all discrete Gaussian sampling over rings is done in the coefficient representation.

We use $\mathcal{U}_q$ to denote discrete uniform distribution over $\mathbb{Z}_q$ and $\mathcal{R}_q$. We define $k = \lceil \log_2 q \rceil$ as the number of bits required to represent integers in $\mathbb{Z}_q$.

2.1 Double-CRT (RNS) Representation

Our implementation utilizes the Chinese Remainder Theorem (referred to as integer CRT) representation to break multi-precision integers in $\mathbb{Z}_q$ into vectors of smaller integers to perform operations efficiently using native (64-bit) integer types. The integer CRT representation is also often referred to as the Residue-Number-System (RNS) representation. We use a chain of same-size prime moduli $q_0, q_1, q_2, \ldots$ satisfying $q_i \equiv 1 \bmod 2n$. Here, the modulus q is computed as $\prod_{i=0}^{l-1} q_i$, where l is the number of prime moduli needed to represent q. All

polynomial multiplications are performed on ring elements in polynomial CRT representation where all integer components are represented in the integer CRT basis. Using the notation proposed in [30], we refer to this representation of polynomials as "Double-CRT".

2.2 Ring Learning with Errors

The following distinguishing problem, originated by Regev and modified to an algebraic version [46], will be our source of cryptographic hardness.

Definition 1. *(Gaussian-secret, cyclotomic-RLWE). Let $\mathcal{R}$ be a power-of-two cyclotomic ring of dimension n over $\mathbb{Z}$, $q \geq 2$ be integer used as a modulus, and $m > 0$. Let $\mathcal{D}_{\mathcal{R},\sigma}$ be a discrete Gaussian distribution over $\mathcal{R}_q$ (sampled over $\mathcal{R}$, then taken modulo q). Then, the $(\mathcal{R}^l, m, q, \mathcal{D}_{\mathcal{R},\sigma}, \mathcal{D}_{\mathcal{R},\sigma})$RLWE problem is to distinguish between the following two distributions: $\{(\mathbf{A}, \mathbf{s}^T\mathbf{A}+\mathbf{e}^T)\}$ and $\{(\mathbf{A}, \mathbf{u}^T)\}$, where, $\mathbf{s} \leftarrow \mathcal{D}^l_{\mathcal{R},\sigma}$, $\mathbf{A} \leftarrow \mathcal{U}(\mathcal{R}_q)^{l\times m}$, $\mathbf{e} \leftarrow \mathcal{D}^m_{\mathcal{R},\sigma}$ and $\mathbf{u} \leftarrow \mathcal{U}({\mathcal{R}_q}^m)$*[5].

Note, the hardness of discrete-Gaussian-secret (R)LWE is usually called the *normal form* of (R)LWE [5, 47].

The LWE assumption is often extended to its multi-secret form. We will only need the multi-secret case when $\mathcal{R} = \mathbb{Z}$, and pseudorandomness follows from LWE [48, Lemma 2.9]. Specifically, the κ-secret $(n, q, m, \mathcal{D}_{\mathbb{Z},\sigma})$LWE distribution is $(\mathbf{A}, \mathbf{AS} + \mathbf{E})$ where $\mathbf{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{m\times n})$, $\mathbf{S} \leftarrow \mathcal{D}^{n\times\kappa}_{\mathbb{Z},\sigma}$, and $\mathbf{E} \leftarrow D^{m\times\kappa}_{\mathbb{Z},\sigma}$. We will need the hardness of multi-secret LWE when the noise distribution is $p \cdot \mathcal{D}_{\mathbb{Z},\sigma}$ instead of $\mathcal{D}_{\mathbb{Z},\sigma}$, where $p \not| q$. The pseudorandomness of LWE with error drawn from $p \cdot \mathcal{D}_{\mathbb{Z},\sigma}$ is equivalent to LWE with error from $\mathcal{D}_{\mathbb{Z},\sigma}$ (proved by multiplying by p and p^{-1}).

2.3 GGH15 Encoding

We will use the generalized GGH15 construction [29] given in [22], called γ-GGH15. For a complete description, see Section 2 of [22].

First, we give the parameters and variables. Fix some ring $\mathcal{R}_q$. Let $\ell > 0$ be a fixed computation length, $(\mathbf{M}_{i,b} \in {\mathcal{R}_q}^{w\times w})_{i\in[\ell],b\in\{0,1\}}$ be a collection of binary, scalar matrices to be used as a form of computation, e.g. a matrix-branching program, and let $(s_{i,b} \in \mathcal{R}_q)_{i\in[\ell],b\in\{0,1\}}$ be a tuple ring elements. Let $\gamma(\mathbf{M}, s)$ be a function mapping $(\mathbf{M}, s)$ to another matrix satisfying $\gamma(\mathbf{M}, s)\gamma(\mathbf{M}', s') = \gamma(\mathbf{MM}', ss')$. The three choices of γ we will use are $\gamma(\mathbf{M}, s) = s$, $\gamma(\mathbf{M}, s) = \mathbf{M} \otimes s$, and $\gamma(\mathbf{M}, s) = diag(s, \mathbf{M} \otimes s)$ where $diag(\cdot, \cdot)$ is a diagonal matrix. For an $\mathbf{x} \in \{0, 1\}^\ell$, define the matrix subset products $\mathbf{Z}_\mathbf{x} = \prod_{i=1}^{\ell} \mathbf{Z}_{i,x_i}$ given any tuple of matrices $(\mathbf{Z}_{i,b})_{i\in[\ell],b\in\{0,1\}}$.

The γ-GGH15 construction, given as input the matrices $(\mathbf{M}_{i,b}, s_{i,b})_{i\in[\ell],b\in\{0,1\}}$ along with an additional matrix $\mathbf{A}_\ell$, returns the matrix $\mathbf{A}_0$ as well as the tuple $(\mathbf{D}_{i\in[\ell],b\in\{0,1\}})$ satisfying $\mathbf{A}_0\mathbf{D}_\mathbf{x} \approx \gamma(\mathbf{M}_\mathbf{x}, s_\mathbf{x})\mathbf{A}_\ell \mod q$ for any $\mathbf{x} \in \{0, 1\}^\ell$.

[5] This problem is referred to as GLWE or MLWE in literature [16,41], though we refer to it as RLWE for succintness.

$\underline{\mathbf{Exp}^{\mathbf{real}}_{\mathbf{tOB,A}}(1^\lambda)\mathbf{:}}$

$\text{osk} \leftarrow \textsc{Setup}(1^\lambda)$

$(\text{C}, \text{st}_A) \leftarrow \text{A}_1(1^\lambda)$

$\text{O} \leftarrow \textsc{Obf}(\text{osk}, \text{C})$

$\alpha \leftarrow \text{A}_2^{\text{TG}(\text{osk},\cdot)}(\text{C}, \text{O}, \text{st}_A)$

Return α

$\underline{\mathbf{Exp}^{\mathbf{ideal}}_{\mathbf{tOB,A,S}}(1^\lambda)\mathbf{:}}$

$(\text{C}, \text{st}_A) \leftarrow \text{A}_1(1^\lambda)$

$(\text{O}^*, \text{st}_S) \leftarrow \text{S}_1(1^\lambda, 1^{|C|})$

$\alpha \leftarrow \text{A}_2^{\text{OS}(\cdot,C)[[\text{st}_S]]}(\text{C}, \text{O}^*, \text{st}_A)$

Return α

Fig. 1: TBO security game.

2.4 Query-Revealing Token-Based Obfuscation

Here we define token-based obfuscation with restricted queries. Our definition is similar to [34], though weaker since the input query x is in the clear. Let λ be a security parameter throughout the following two definitions.

Definition 2 (Query-Revealing TBO). *Let $n = n(\lambda) \in \mathbb{N}$. A query-revealing token-based obfuscation scheme for a class of circuits $\{\mathcal{C}_n\}_{n\in\mathbb{N}}$, where each $\mathcal{C}_n$ is a set of n-bit-input circuits, is a tuple of probabilistic polynomial-time algorithms* $(\textsc{Setup}, \textsc{Obfuscate}, \textsc{TokenGen})$ *with the following properties:*

- $\textsc{Setup}(1^\lambda)$ *takes as input a security parameter λ and returns a secret key* osk.
- $\textsc{Obfuscate}(\text{osk}, C \in \mathcal{C}_n)$ *takes as input a circuit C, a secret key* osk*, and outputs an obfuscated circuit O.*
- $\textsc{TokenGen}(\text{osk}, x)$ *takes as input the secret key* osk *and some input $x \in \{0,1\}^n$, and returns a token tk_x.*

We require $O(tk_x) = C(x)$ with all but negligible probability.

Next, we define the security game in Figure 1. We abbreviate $(\textsc{Obfuscate}, \textsc{TokenGen})$ as $(\textsc{Obf}, \text{TG})$. In Figure 1, $\text{OS}(\cdot, C)[[\text{st}_S]]$ is an oracle that on input x from A_2, runs S_2 with inputs $C(x)$, x (note that it was $1^{|x|}$ in the query-hiding TBO of [34]), and the current state of S, st_S. S_2 responds with a fake tk^*_x and a new state st'_S which OS will feed to S_2 on the next call. OS returns tk^*_x to A_2.

Definition 3 (Security). *The TBO scheme is secure if there exists a pair of PPT simulation algorithms (S_1, S_2) such that for all PPT adversaries (A_1, A_2), the two probabilistic experiments defined in Figure 1 are computationally indistinguishable* $\{\text{Exp}^{\text{real}}_{\text{tOB,A}}(1^\lambda)\} \approx_c \{\text{Exp}^{\text{ideal}}_{\text{tOB,A,S}}(1^\lambda)\}$.

3 Token-Based Obfuscation of Linear Functions

We first present an LWE secret-key scheme that can be used for the TBO of linear functions and then demonstrate how it can be applied for encoding binary

classifiers. We provide a discussion suggesting that the security of TBO for binary classifiers is determined by the statistical poperties of the classification rules rather than our obfuscation technique. The use of token-based approach allows one to bound the number of queries and restrict inputs based on the statistical properties of classification rules, thus overcoming one of the major limitations of non-interactive obfuscation security models, such as VBB and IO.

3.1 LWE Secret Key Scheme for Evaluating Linear Weighted Sums

The purpose of this scheme is to perform evaluation on obfuscated tests for linear functions. The evaluation function can be described as $\sum_{i=1}^{N} w_i x_i$, where $\mathbf{x} = (x_1, \ldots, x_N) \in \mathbb{Z}_p^N$ and $\mathbf{w} = (w_1, \ldots, w_N) \in \mathbb{Z}_p^N$ refer to data and weights, respectively. Here, N is the dataset size (number of variables in the linear function). In the obfuscation case, the weights are encrypted and data (inputs) are in clear. For each input, a new token is generated.

It is also possible to invert this problem by encrypting the inputs and storing the weights in the clear. Each vector of weights would then require a token. This formulation would apply to use cases where the function is public but data need to stay encrypted.

The scheme is a tuple of functions, which includes ParamGen, KeyGen, Obfuscate, TokenGen, and Eval, where the functions are defined as follows:

- $\textsc{ParamGen}(1^\lambda, N, p)$: Given a security parameter λ and system parameters N and p, select an integer modulus q, LWE security parameter n, and discrete Gaussian distribution χ with standard deviation $\sigma/\sqrt{2\pi}$.
- $\textsc{KeyGen}(N) \rightarrow \mathrm{SK}$. Generate N secret vectors $\mathbf{s}_i \in \mathbb{Z}_q^n$, where $N \geq 2$. For example, use a nonce K and define $\mathbf{s}_i$ to be a hash of K concatenated to the index i.
- $\textsc{Obfuscate}(\mathrm{SK}, \mathbf{w}) \rightarrow \mathbf{O}$. Choose a random vector $\mathbf{a} \in \mathbb{Z}_q^n$ and error values (numbers) $e_i \in \mathbb{Z}_q$ generated using χ. Compute the obfuscated program

$$\mathbf{O} := \left[\mathbf{a}, \mathbf{b} := \{\langle \mathbf{a}, \mathbf{s_i} \rangle + p e_i + w_i\}_{i=1}^{N}\right].$$

- $\textsc{TokenGen}(\mathrm{SK}, \mathbf{x}) \rightarrow \mathbf{t}$. Generate tokens for data $\mathbf{x}$ as $\mathbf{t} := \sum_{i=1}^{N} x_i \mathbf{s}_i \in \mathbb{Z}_q^n$. For each distinct data input, a separate token needs to be generated.
- $\textsc{Eval}(\mathbf{O}, \mathbf{t}, \mathbf{x}) \rightarrow \bar{\mu}$. Compute

$$\bar{\mu} := \sum_{i=1}^{N} b_i x_i - \langle \mathbf{a}, \mathbf{t} \rangle \mod p.$$

The scheme is correct, i.e., $\bar{\mu} = \sum_{i=1}^{N} w_i x_i$, as long as the noise does not cause a wrap-around w.r.t. mod$q$. Indeed,

$$\sum_{i=1}^{N} b_i x_i - \langle \mathbf{a}, \mathbf{t} \rangle = \sum_{i=1}^{N} x_i \cdot \langle \mathbf{a}, \mathbf{s}_i \rangle + p \cdot \sum_{i=1}^{N} x_i e_i$$
$$+ \sum_{i=1}^{N} w_i x_i - \langle \mathbf{a}, \sum_{i=1}^{N} x_i \mathbf{s}_i \rangle = p \cdot \sum_{i=1}^{N} x_i e_i + \sum_{i=1}^{N} w_i x_i,$$

where the first term in the result is eliminated after applying mod$p$. For the evaluation to be correct, the following correctness constraint has to be satisified:

$$\left\| p \cdot \sum_{i=1}^{N} x_i e_i \right\|_{\infty} < q/4.$$

Note on key generation. Another alternative for key generation is to compute secret keys as AES(K,i). More specifically, we can generate a secret key K for AES and do encryptions of a counter to generate 128-bit random sequences. These sequences would then be used for random numbers in $\mathbb{Z}_q$. In this scenario, we need to store only the nonce K and can generate a particular key on the fly. In other words, the space requirements for the secret key vectors are limited by the value of n (negligibly small from the practical perspective). This method is used in our implementation.

Security The obfuscated program $\mathbf{O}$ is secure under LWE. We prove the security of the TBO scheme using Definition 2.

Theorem 1. *The LWE secret-key scheme for evaluating linear weighted sums is secure under query-revealing TBO (Definition 2).*

Proof. The outline of the proof is as follows: first we sketch a transformation from our scheme to one which follows the definition of TBO given in the preliminaries (Definition 2), including a simulation mode for token generation, then we prove the distributions of $(C, \mathrm{st}_A, \mathrm{O}, \{x\}, \{\mathrm{tk}_x\})$ are computationally close in both games.

General transformation. First, we transform our construction to match the definition given in the preliminaries. We fold ParamGen and KeyGen into Setup. Since Eval is implicit, we now assume the scheme is described as a tuple of PPT algorithms (Setup, Obfuscate, TokenGen).

Simulators. Next, we define the simulator $S_1(1^\lambda, 1^{|C|})$ to sample a uniformly random $\mathbf{S} \in \mathbb{Z}_q^{n \times N}$, and encode the $\mathbf{0}$-linear function $\mathrm{O}^* \leftarrow \textsc{Obfuscate}(\mathbf{S}, \mathbf{0})$. We assume $\mathbf{a}$'s first entry, a_1, is invertible over $\mathbb{Z}_q$ (the proof works as long some index i has $a_i \in \mathbb{Z}_q^*$, which is true with high probability). The state st_s is the LWE public vector $\mathbf{a}$ and secret $\mathbf{S}$. $\mathrm{S}_2(\mathrm{C}(x), x, \mathrm{st}_s)$ simulates token keys by

returning

$$\mathbf{t}_x^* = \mathbf{S}\mathbf{x} - a_1^{-1} \begin{bmatrix} \mathbf{w}^T\mathbf{x} \\ \mathbf{0} \end{bmatrix} \mod q = \left(\mathbf{S} - a_1^{-1} \begin{bmatrix} \mathbf{w}^T \\ \mathbf{0} \end{bmatrix} \right) \mathbf{x} \mod q = \mathbf{S}'\mathbf{x}$$

along with $\mathrm{st}_S = \mathbf{S}$.

Indistinguishability. Now we argue the distributions

$$\{(C, \mathrm{state}_A, \mathrm{O}, \{x, \mathbf{t}_x\}) : \mathrm{Exp}^{\mathrm{real}}_{\mathrm{tOB,A}}(1^\lambda)\}$$

and

$$\{(C, \mathrm{state}_A, \mathrm{O}^*, \{x, \mathbf{t}_x^*\}) : \mathrm{Exp}^{\mathrm{ideal}}_{\mathrm{tOB,S,A}}(1^\lambda)\}$$

are computationally indistinguishable. The distribution of $(\mathrm{C}, \mathrm{st}_A)$ is the same in both games by definition. Next, the distribution of $(\mathrm{C}, \mathrm{st}_A, \mathrm{O})$ is computationally indistinguishable from $(\mathrm{C}, \mathrm{st}_A, \mathrm{O}^*)$ by the pseudorandomness of N-secret LWE. Indeed, the same is true for $(\mathrm{C}, \mathrm{st}_A, \mathrm{O}, x)$ and $(\mathrm{C}, \mathrm{st}_A, \mathrm{O}^*, x)$, where x is A_2's first token query. Finally, the token queries are computationally indistinguishable since both $\mathbf{S}$ and $\mathbf{S}'$ are uniformly random matrices conditioned on $\mathbf{b}^t - \mathbf{a}^t\mathbf{S} = p\mathbf{e}^t + \mathbf{w}^t$ in the real game, and $\mathbf{b}^{*t} - \mathbf{a}^t\mathbf{S}' = p\mathbf{e}^t + \mathbf{w}^t$ in the ideal game. This implies the evaluations $\mathbf{t}_x = \mathbf{S}\mathbf{x}$ and $\mathbf{t}_x^* = \mathbf{S}'\mathbf{x}$ are indistinguishable to the adversary. Therefore, the views of A_2 in the real and ideal games are computationally indistinguishable, which implies its respective outputs are computationally indistinguishable. □

The main practical security limitation comes from the use of linear functions. All weights can be found in at most N queries. When the weights vector $\mathbf{w}$ is sparse (especially if the locations of some zeros are known) or some weight components are correlated, the number of queries is even smaller. This implies that the maximum number of queries for which tokens can be generated should be selected based on the dimension N as well as sparsity and other possible special properties of the weights vector $\mathbf{w}$.

Comparison with Construction [1] Abdalla *et al.* propose a functional encryption scheme for inner products based on LWE that reveals nothing else other than the result of the inner product. This scheme can be inverted to instantiate a query-revealing TBO for inner products. In this case, the secret key can be treated as a "token", and the scheme complexity can be directly compared to our construction. Table 1 compares the asymptotic complexity of both constructions using the TBO notation. The table suggests that the size of the master public key and corresponding public-key encryption time grow quadratically with N, preventing the practical use of scheme [1] for large N. In contrast, the key requirements and obfuscation times of our construction are significantly smaller, and grow linearly with N.

Table 1: Asymptotic complexity of our construction vs. the LWE construction based on functional encryption of inner products in [1]; $m > (N + n + 1) \log q$; computational complexity is expressed as the number of modular integer multiplications and additions

	Space complexity				Computational complexity		
Scheme	MSK	$\mathbf{a}$/MPK	$\mathbf{t}$	$\mathbf{O}$	Obfuscate	TokenGen	Eval
Ours	$Nn \log q$	$n \log q$	$n \log q$	$(N + n) \log q$	$O(Nn)$	$O(Nn)$	$O(N + n)$
[1]	$Nn \log q$	$Nm \log q$	$n \log q$	$(N + n) \log q$	$O(mn + Nm)$	$O(Nn)$	$O(N + n)$

3.2 Token-Based Obfuscation of Binary Classifiers

The TBO of linear functions can be used to build obfuscated binary classifiers. For instance, it can be directly applied for prediction using linear classifiers. As an example, consider a linear Support Vector Machine classification model that can be represented as a vector of weights and a bias, and obfuscated using the linear TBO scheme. The scheme can also be used for certain more advanced binary classifiers (with some non-linear properties). In this work we consider two cases of more advanced classifiers: (1) an arbitrary classification rule for a single feature and (2) a conjunction of such classifiers for multiple features. Note that the single-feature scenario is introduced only to illustrate the encoding but it cannot be used in practice because each token query would reveal one of the secret key vectors.

Single feature Given an M-bit feature and an arbitrary binary classification rule, we can map every possible value of the feature to a different component in the weights vector and then compute a linear weighted sum over all possible values to find the result of the classification. In this case, the dimension N of $\mathbf{w}$ and $\mathbf{x}$ is equal to 2^M.

Let us set the weight component w_i to 0 for all matching M-bit patterns and to 1 for all non-matching patterns, and encode M-bit patterns in an ascending order. In other words, a binary 0 is mapped to w_1, a binary 1 to w_2, and a binary $111 \ldots 111$ to index w_{2^M}. The same indexing function is applied to inputs, i.e., we convert the binary representation of the input to decimal representation. We then set x_i at this computed index to 1 and all other components of $\mathbf{x}$ are set to 0.

With this setup, $\sum_{i=1}^{N} w_i x_i = 0$ for a matching value of the attribute and 1 otherwise. This technique can be generalized to support a binary classification based on a conjunction of multiple features, which is discussed next.

Conjunction of multiple features We can concatenate the weights and input vectors for individual features to provide a binary classifier for a conjunction of P features. If the inputs for all P features match, $\sum_{i=1}^{N} w_i x_i = 0$. If there is at least

one non-matching value for one of the features, the result will be in the range from 1 to P. To hide the number of non-matching features in this scenario, we can encode non-matching components of w_i as a random value between 1 and $p-1$. With this setup, we get $\sum_{i=1}^{N} w_i x_i \equiv 0 \mod p$ when there is a match and a random value between 1 and $p-1$ otherwise. The value p should be large enough to avoid a false positive with a high probability (we used $p = 2^{40}$ in our experiments). Alternatively, we can keep the result for a non-matching input unchanged (in the range from 1 to P) to provide a linear classifier functionality with a non-binary output.

One potential application of this classifier is image classification. Images can be matched against known classification rules. Each feature would represent a subset of an image, such as a pixel or a square of pixels.

From the practical perspective, the statistical properties of a classification rule need to be examined before obfuscation to determine the practical bound on the number of queries. Furthermore, the token generator may not allow certain inputs. This implies that the token-based obfuscation approach may address an inherent security limitation of non-interactive models, such as IO and VBB, w.r.t. classifiers by bounding the number of queries and restricting query inputs.

We discuss the storage requirements, computational complexity, and scalability of the TBO for binary classifiers in Appendix B.

4 Token-Based Obfuscation of Conjunctions

We next consider a construction for the token-based obfuscation of conjunctions based on Ring LWE. Our TBO construction is a significantly optimized variant of the bit-fixing construction for constraint-hiding constrained PRFs proposed in Section 5.1 of [18]. We chose the example of conjunctions to give a fair comparison with a prior recent non-interactive (distributional VBB) conjunction obfuscation construction implemented in [24] and introduce several major optimizations used in the next section for the TBO of more general programs, i.e., branching programs.

Compared to the non-interactive conjunction obfuscation construction implemented in [24] (and originally formulated in [17]), the TBO construction has several advantages w.r.t. both security and efficiency. The construction [17,24] is secure under entropic (non-standard) Ring LWE while the current construction is secure under LWE. The token-based security model allows one to limit the number of queries versus the unbounded number of queries in the case of [24] (the latter would allow the adversary to learn the full pattern unless a relatively long pattern with high entropy is used). Our complexity analysis (and experimental results later in the paper) show that the program size and evaluation runtime in the case of TBO are significantly smaller. The only drawback of TBO is the need to have a trusted party generating tokens (either in advance or for each query on demand).

4.1 Definition of Conjunctions

We define a conjunction as a function on L-bit inputs, specified as $f(x_1, \ldots, x_L) = \bigwedge_{i \in I} y_i$, where y_i is either x_i or $\neg x_i$ and $I \subseteq [L]$. The conjunction program checks that the values $x_i : i \in I$ match some fixed pattern while the values with indices outside I can be arbitrary. We represent conjunctions further in the paper as vectors $\mathbf{v} \in \{0, 1, \star\}^L$, where we define $F_{\mathbf{v}}(x_1, \ldots, x_L) = 1$ iff for all $i \in [L]$ we have $x_i = v_i$ or $v_i = \star$. We refer to $\star$ as a "wildcard".

This type of conjunctions is used in machine learning to execute or approximate classes of classifiers [40, 52]. Conjunctions can be used to encode binary classifiers (similar to the approach discussed in Section 3.2) but with some additional restrictions due to the wild-card-based (rather than arbitrary) format of patterns. A more detailed discussion on conjunctions and their applications is presented in [24].

4.2 Conceptual Model

The scheme for the TBO of conjunctions includes the same tuple of functions as the TBO for linear functions (Section 3.1) but the concept of token is used differently. The conceptual workflow is defined as follows:

- ParamGen: Generate lattice parameters based on the length of the pattern and security level.
- KeyGen: Generate trapdoor key pairs and an unconstrained master secret key. The unconstrained key corresponds to a pattern of all wild cards (which accepts any pattern).
- Obfuscate: An obfuscated program (constrained key) for a given pattern is generated by replacing the master key elements with random samples where a specific bit is fixed (no changes are made for wild cards in the input pattern).
- TokenGen: Compute a vector $\mathbf{y}' \in \mathcal{R}_p^{1 \times m}$, which is a result of evaluating the PRF, to generate the token using the master (unconstrained) key.
- Eval: Evaluate the obfuscated program using the constrained key (obfuscated program) and output a vector $\mathbf{y} \in \mathcal{R}_p^{1 \times m}$, where $\mathcal{R}_p = \mathbb{Z}_p[x] / \langle x^n + 1 \rangle$. Compare $\mathbf{y}$ with $\mathbf{y}'$; if they match, output 1 (True), otherwise output 0 (False).

The output of TokenGen is the PRF value, and is used as the "token" in this case. If the token for the unconstrained key (master seret key) matches the output for the constrained key (obfuscated program), the result is 1 (True).

The TokenGen procedure is executed for each input by a trusted party. The Eval operation is executed by a public (untrusted) party. ParamGen, KeyGen, and Obfuscate are offline operations. EvalToken and Eval are online operations in the scenario where a token generator is available to generate a token for each input on demand.

We next describe the algorithms for each function.

4.3 Algorithms for TBO Functions

The building blocks of the TBO construction for conjuctions, such as lattice trapdoor sampling and GGH15 directed encoding, are the same as for the distributional VBB conjunction obfuscation construction implemented in [24], which makes it possible to provide a fair comparison of both constructions. In this section we provide the pseudocode for the algorithms, focusing on the differences between the constructions and our optimizations w.r.t. to the theoretical bit-fixing constraint-hiding constrained PRF construction proposed in [18].

The main difference of the TBO model as compared to the distributional VBB model [24] is the interaction between untrusted and trusted components of the system. This bounds the number of evaluation queries and prevents exhaustive search attacks that the distributional VBB construction is amenable to.

The main optimizations w.r.t. the construction in [18] include the use of a larger (non-binary) alphabet for encoding words of the pattern, an asymptotically and practically faster procedure (with much smaller storage requirements) for generating the tokens, and significantly tighter correctness constraints.

Algorithm 1 Key generation

function KeyGen(1^λ)
  for `i = 0..`$\mathcal{L}$ **do**
    $\mathbf{A}_i, \widetilde{\mathbf{T}}_i :=$ TrapGen(1^λ)
  for `i = 1..`$\mathcal{L}$ **do**
    for b `= 0..`2^w`-1` **do**
      $s_{i,b} \leftarrow \mathcal{D}_{\mathcal{R},\sigma}$
  return $K_{MSK} := \left(\{s_{i,b}\}_{i\in\{1,..,\mathcal{L}\},b\in\{0,..,2^w-1\}}, \{\mathbf{A}_i, \widetilde{\mathbf{T}}_i\}_{i\in\{0,..,\mathcal{L}\}}\right)$

The key generation algorithm is listed in Algorithm 1. The prameter $\mathcal{L} = \lceil L/w \rceil$ is the effective length of conjunction pattern, w is the number of bits per word of the pattern, $s_{i,b} \in \mathcal{R}$ is the i-th word secret-key component for the b-th value of the current word, $\mathbf{A}_i \in \mathcal{R}_q^{1\times m}$ is the public key for the i-th word, $\widetilde{\mathbf{T}}_i \in \mathcal{R}_q^{2\times\kappa}$ is the trapdoor for the i-th word, κ is the number of digits used in Gaussian sampling, and $m = 2 + \kappa$. The key generation procedure includes two major steps: generating $\mathcal{L}$ trapdoors (the definition of TrapGen is given in Appendix F) and computing the unconstrained key as $\mathcal{L} \times b$ short ring elements.

As compared to the construction in [18], we optimized the master secret key generation to only sample short ring elements $s_{i,b}$ (without calling complex lattice trapdoor sampling for these short ring elements), which reduces the storage and speed complexity for the unconstrained key by a factor of $O(m^2)$. In the original construction, the size of the master key was approximately the same as the obfuscated program. In summary, the storage requirement for the keys in our construction is $O(\mathcal{L}bn) + O(\mathcal{L}(m + 2\kappa)n)$ integers in $\mathbb{Z}_q$ versus $O(m^2\mathcal{L}bn) + O(\mathcal{L}(m + 2\kappa)n)$ integers in the original construction. Storing the

secret keys rather their GGH15 encodings does not effect the security of the construction in the TBO model as the trusted party is allowed to have access to the secret keys by definition.

Algorithm 2 Obfuscation

function OBFUSCATE($\mathbf{v} \in \{0,1,*\}^L, K_{MSK}, \sigma$)
  for `i = 1..`$\mathcal{L}$ **do**
    `Build binary mask` M (0's correspond to wild-card bits, 1's correspond to fixed bits)
    for b `= 0..`2^w`-1` **do**
      if $(b \wedge M) \neq (v \wedge M)$ **then**
        $r_{i,b} \leftarrow \mathcal{D}_{\mathcal{R},\sigma}$
      else
        $r_{i,b} := s_{i,b}$
      $\mathbf{D}_{i,b} := \mathsf{Encode}_{\mathbf{A}_{i-1} \to \mathbf{A}_i}(\widetilde{\mathbf{T}}_{i-1}, r_{i,b}, \sigma)$
  $\Pi_v := \left(\mathbf{A}_0, \{\mathbf{D}_{i,b}\}_{i \in [\mathcal{L}], b \in \{0,\ldots,2^w-1\}}\right)$
  return Π_v

Algorithm 2 lists the pseudocode for the main obfuscation function OBFUSCATE. We encode words of conjunction pattern $\mathbf{v} \in \{0,1,\star\}^L$ rather than bits as in the original construction [18]. Each word is w bits long, and 2^w is the number of encoding matrices for each encoded word of the pattern. The actual pattern length L gets replaced with the effective length $\mathcal{L} = \lceil L/w \rceil$ to reduce the number of encoding levels (multi-linearity degree). When the fixed bits in the encoded word match the fixed bits in the pattern being obfuscated, the obfuscated program uses the short ring elements $s_{i,b}$ from the unconstrained key. Otherwise, new short ring elements $r_{i,b}$ specific to the obfuscated program are generated.

The OBFUSCATE procedure relies on an ENCODE algorithm for the directed-encoding ring instantiation to encode each word of the conjunction pattern. The ENCODE algorithm is depicted in Algorithm 3 and is the same GGH15 directed encoding procedure as described in [24]. The lattice trapdoor sampling procedure GAUSSSAMP is described in Appendix F.

The storage requirement for the obfuscated program Π_v is $O(\mathcal{L}bm^2n)$.

Algorithm 3 Directed encoding

function $\mathsf{Encode}_{\mathbf{A}_i \to \mathbf{A}_{i+1}}(\mathbf{T}_i, r, \sigma)$
  $\mathbf{e}_{i+1} \leftarrow \mathcal{D}_{\mathcal{R},\sigma} \in {\mathcal{R}_q}^{1 \times m}$.
  $\mathbf{b}_{i+1} := r\mathbf{A}_{i+1} + \mathbf{e}_{i+1} \in \mathcal{R}_q^{1 \times m}$
  $\mathbf{R}_{i+1} := \mathsf{GaussSamp}(\mathbf{A}_i, \mathbf{T}_i, \mathbf{b}_{i+1}, \sigma_t, \sigma_s) \in \mathcal{R}_q^{m \times m}$
  return $\mathbf{R}_{i+1}$

Algorithm 4 Token Generation: Evaluation by a trusted party (using the master secret key)

function TokenGen($\mathbf{x} \in \{0,1\}^L$, K_{MSK})
  $\Delta := \mathbf{A}_{\mathcal{L}}[1] \prod_{i=1}^{\mathcal{L}} s_{i,x[1+(i-1)w\,:\,iw]}$
  $\mathbf{y}' := \lfloor \frac{2}{q} \Delta \rceil \in \mathbb{Z}_2^n$
  return $\mathbf{y}'$

Algorithm 4 lists the pseudocode for TokenGen, i.e., the evaluation of the input using unconstrained key. Our variant is significantly optimized compared to the construction in [18]: it multiplies short ring elements followed by a single scalar product with the second ring element of the public key $\mathbf{A}_0$ (in contrast to vector-matrix products in [18]), which reduces the computational complexity by a factor of $O(m^2)$.

Algorithm 5 shows the psedocode for the evaluation of a given input using the obfuscated program (constrained key).

Algorithm 5 Evaluation using the obfuscated program

function Eval($\mathbf{x} \in \{0,1\}^L$, Π_v, $\mathbf{y}'$)
  $\mathbf{D}_{\mathbf{\Pi}} := \mathbf{A}_0$
  for `i = 1..`$\mathcal{L}$ **do**
    $\mathbf{D}_{\mathbf{\Pi}} := \mathbf{D}_{\mathbf{\Pi}}\, \mathbf{D}_{i,x[1+(i-1)w\,:\,iw]} \in {\mathcal{R}_q}^{1\times m}$
  $\mathbf{y} := \lfloor \frac{2}{q} \mathbf{D}_{\mathbf{\Pi}}[1] \rceil \in \mathbb{Z}_2^n$
  return $(\mathbf{y} = \mathbf{y}')$

In this case, the token $\mathbf{y}'$ is generated using a lattice PRF. We do not need to perform the comparison of all polynomial coefficients in $\mathbf{y}'$ and $\mathbf{y}$. Instead we can perform it for the number of coefficients that makes the probability of a false positive negligibly small. In our experiments, we chose this number to be 128. Dropping a fixed number of bits from a PRF retains all security measures.

Next, we note that the probability of comparison error is linear in the number of coefficients compared under the heuristic that the coefficients are independent and uniformly distributed over $\mathbb{Z}_q$. Let B be our bound on the GGH15 noise. Then, the probability of a rounding error in a comparison of the entire output is less than $(nmd)\frac{4B}{q}$ since there are two "bad" regions of $\mathbb{Z}_q$ of size $2B$ corresponding to rounding errors and there are nmd $\mathbb{Z}_q$-coefficients being rounded to bits (nmd bits)[6]. By only comparing α bits, we can replace this by $\alpha \cdot \frac{4B}{q}$. The choice of α and the probability upper-bound for a comparison error will affect the modulus size (Appendix C).

[6] This analysis is nearly identical to the original LWE to LWR reduction in [8] for $p = 2$.

4.4 Security

The TBO construction for conjunctions is secure under Definition 2 for query-revealing TBO under Ring LWE. The proof showing that the existence of constraint-hiding constrained PRF implies an the existence of a query-revealing TBO scheme is presented in Appendix A. Further, we are able to base security on plain RLWE instead of small-entry **A** RLWE [15]. See Remark 1 in Appendix A for more details.

4.5 Setting the Parameters

Ring-LWE trapdoor construction. The trapdoor secret polynomials are generated with a noise width σ, which is at least the smoothing parameter estimated as $\sqrt{\ln(2n_m/\epsilon)/\pi}$, where n_m is the maximum ring dimension and ϵ is the bound on the statistical error introduced by each randomized-rounding operation [49]. For $n_m \leq 2^{14}$ and $\epsilon \leq 2^{-80}$, we choose a value of $\sigma \approx 4.578$.

Short Ring Elements in Directed Encoding. For short ring elements $s_{i,b}, r_{i,b}$, we use error distribution with the distribution parameter σ. This implies that we rely on Ring-LWE for directed encoding.

Directed Encoding. To encode short ring elements, we use the error distribution with noise width σ (for the noise polynomials).

G-Sampling. Our G-sampling procedure requires that $\sigma_t = (t+1)\sigma$. This guarantees that all integer sampling operations (noise widths) inside G-sampling are at least the smoothing parameter σ, which is sufficient to approximate the continuous Gaussian distribution with a negligible error.

Spectral norm σ_s. Parameter σ_s is the spectral norm used in computing the Cholesky decomposition matrix (it guarantees that the perturbation covariance matrix is well-defined). To bound σ_s, we use inequality $\sigma_s > s_1(\mathbf{X})\,\sigma_t$, where $\mathbf{X}$ is a sub-Gaussian random matrix with parameter σ [49]. Lemma 2.9 of [49] states that $s_1(\mathbf{X}) \leq C_0\sigma\left(\sqrt{n\kappa} + \sqrt{2n} + C_1\right)$, where C_0 is a constant and C_1 is at most 4.7. We can now rewrite σ_s as $\sigma_s > C_0\sigma\sigma_t\left(\sqrt{n\kappa} + \sqrt{2n} + 4.7\right)$. In our experiments we used $C_0 = 1.3$, which was found empirically.

Modulus q. The correctness constraint for a conjunction pattern with $\mathcal{L}$ words ($\mathcal{L} \geq 2$) is expressed as $q > 2^{10}P_e^{-1}B_e\left(\beta\sigma_s\sqrt{mn}\right)^{\mathcal{L}-1}$, where $B_e = 6\sigma, \beta = 6$, $P_e = 2^{-20}$, and all other parameters are the same as in [24]. The derivation details are presented in Appendix C.

Ring Dimension n. All of the security proofs presented in [18] for the constraint-hiding constrained PRF directly apply to our construction, which implies that the TBO of conjunctions is secure under Ring LWE. To choose the ring dimension, we run the LWE security estimator[7] (commit a2296b8) [3] to find the lowest security levels for the uSVP, decoding, and dual attacks following the standard homomorphic encryption security recommendations [20]. We choose the least value of λ for all 3 attacks on classical computers based on the estimates for the BKZ sieve reduction cost model, and then multiply it by the number

[7] https://bitbucket.org/malb/lwe-estimator

of encoded matrices, corresponding to the number of Ring LWE problems that need to be solved.

Dimension m. The dimension m was set to $2 + \kappa$ following the logic described in [24].

Word size w. We found $w = 8$ to be the optimal value for all our experiments, using the same procedure as described in [24].

4.6 Comparison with Construction in [24]

As the building blocks and many underlying parameters for the TBO construction are the same as for the distributional VBB constructon [24], we can directly compare them. The noise constraints are approximately the same as the smaller depth in the TBO construction (by 1) is compensated by the extra factor of approximately $2^5 P_e^{-1}$ introduced by the rounding. The construction in [24] requires computing two product chains versus just one product chain in our TBO construction. All other parameters are the same. This implies that the TBO construction should be at least twice faster in obfuscation and evaluation, and requires 2x smaller storage for the obfuscated program. We provide their experimental comparison later in the paper.

From the security perspective, the TBO model can be used to bound the number of queries and restrict the format of inputs, thus overcoming the main security limitation of the conjunction obfuscation construction discussed in [24].

5 Token-Based Obfuscation of Branching Programs

In this section we present a construction for the TBO of more general classes of programs, namely permutation and general branching programs. For permutation branching programs, we develop an optimized variant of the constrained-hiding constrained PRF construction presented in Section 5.2 of [18]. For general branching programs, we adapt the private constrained PRF[8] construction of [22] (Section 7.2) to rings and add several optimizations to it. Both classes of branching programs are integrated in the same framework, hence we deal with one general construction for the TBO of branching programs. The TBO construction is secure under Ring LWE.

The construction for the TBO of branching programs builds on top of the same procedures as the TBO for conjunctions discussed in Section 4 and then adds an extra layer dealing with matrix branching programs. Conceptually speaking, the TBO of conjunctions may be considered as a simple special case of the TBO for branching programs. In this section we focus on the aspects specific to branching programs, implying that all other underlying building blocks and parameters are the same as for the TBO of conjunctions.

[8] Private constrained PRF and constrained-hiding constrained PRF are two interchangeable terms referring to the same capability.

Compared to the constructions in [22] and [18], our construction includes the following optimizations: (1) significantly improved key generation and evaluation algorithms for the token generator (both runtime and storage requirements are dramatically reduced), (2) much tighter correctness constraints (using lower values of main parameters and Central Limit Theorem/subgaussian analysis), and (3) a larger alphabet for encoding input bits.

5.1 Matrix Branching Programs

First, we provide the main definitions of branching programs supported by our construction.

Definition 4. *(Matrix branching programs [22]) Let $l, L \in N$ be the bit-length of the input $\mathbf{x} \in \{0,1\}^l$ and the index of the branching program. Let $f : \{0,1\}^l \to \{0,1\}^L$ be the input-to-index map and $F : \{0,1\}^L \to \{0,1\}^l$ be the index-to-input map.*

A dimension-u, length-L matrix branching program over l-bit inputs consists of an input-to-index map f, a sequence of pairs of 0-1 *matrices, and two disjoint sets of target matrices $\mathbf{P}_0$ and $\mathbf{P}_1$:*

$$\Gamma = \left\{ f, \{\mathbf{M}_{i,b} \in \{0,1\}^{u \times u}\}_{i \in [L], b \in \{0,1\}}, \mathbf{P}_0, \mathbf{P}_1 \right\}.$$

This branching program decides the language $\mathbf{L} \subseteq \{0,1\}^l$ defined as

$$\mathbf{L}(x) = \begin{cases} 0 \ \mathbf{M}_{f(x)} = \prod_{i \in [L]} \mathbf{M}_{i,F(i)} \in \mathbf{P}_0, \\ 1 \ \mathbf{M}_{f(x)} = \prod_{i \in [L]} \mathbf{M}_{i,F(i)} \in \mathbf{P}_1. \end{cases}$$

The dimension u and length L are typically referred to as the width and length of a matrix branching program.

Looking ahead, the applications in this paper may require additional constraint on the target sets $\mathbf{P}_0, \mathbf{P}_1$ to perform the correct functionality.

The following 2 types are supported by our TBO construction.

Definition 5. *(Permutation branching programs: Type II branching programs in [22])*

1. *$\mathbf{M}_{i,b}$'s are permutation matrices*
2. *The target sets $\mathbf{P}_0, \mathbf{P}_1$ satisfy $\mathbf{e}_1 \cdot \mathbf{P}_1 = \{\mathbf{e}_1\}$; $\mathbf{e}_1 \cdot \mathbf{P}_0 = \{\mathbf{e}_2\}$, where $\mathbf{e}_i \in \{0,1\}^{1 \times u}$ denotes the unit vector with the i^{th} coordinate being 1, and the rest being 0.*

Permutation branching programs can be used to represent NC^1 circuits. Barrington's theorem converts any depth-δ Boolean circuits into an oblivious branching program of length $L \leq 4^\delta$ composed of permutation matrices $\{\mathbf{M}_{i,b}\}_{i \in [L], b \in \{0,1\}}$ of dimension u (by default $u = 5$). Evaluation is done by multiplying the matrices selected by input bits, with the final output $\mathbf{I}^{u \times u}$ or a u-cycle $\mathbf{P}_i$, where

$i \in \{0,1\}$, recognizing 0 and 1, respectively. In practice, we can manually construct branching programs with shorter length L and smaller width u than those provided by the general conversion of Barrington's Theorem.

Note that the branching programs obtained by Barrington's theorem directly satisfy Definition 5.

Definition 6. *(General branching programs: Type I branching programs in [22]). For vector* $\mathbf{v} \in \{0,1\}^{1\times u}$*, the target sets* $\mathbf{P}_0, \mathbf{P}_1$ *satisfy* $\mathbf{v}\cdot\mathbf{P}_1 = \{0^{1\times u}\}$*;* $\mathbf{v}\cdot\mathbf{P}_0 \subseteq \{0,1\}^{1\times u} \setminus \{0^{1\times u}\}$.

General branching programs can be used to represent formulas in Conjunctive Normal Form (CNF) (see [22] for two specific representations of CNFs).

The relationships between these two types of branching programs are discussed in [22].

5.2 TBO Construction

At a high level, the TBO construction for branching programs has the same functions as the one for the TBO of conjunctions. The main difference is in how the programs are encoded.

Algorithm 6 Key generation for branching programs

function KeyGen(1^λ)
  for `i = 0..`$\mathcal{L}$ **do**
    $\mathbf{A}_i, \widetilde{\mathbf{T}}_i :=$ TrapGen(1^λ), $\mathbf{A}_i \in {\mathcal{R}_q}^{d\times dm}$
  $\mathbf{J} := \mathbf{e}_1$; $\mathbf{A}_\mathbf{J} := \mathbf{J}\mathbf{A}_0$
  for `i = 1..`$\mathcal{L}$ **do**
    for b `= 0..`2^w`-1` **do**
      $s_{i,b} \leftarrow \mathcal{D}_{\mathcal{R},\sigma}$
  return $K_{MSK} := \left(\{s_{i,b}\}_{i\in\{1,..,\mathcal{L}\},b\in\{0,..,2^w-1\}}, \{\mathbf{A}_i, \widetilde{\mathbf{T}}_i\}_{i\in\{0,..,\mathcal{L}\}}, \mathbf{A}_J\right)$

In the case of conjunctions, each bit is encoded as a short ring element s (we ignore here for simplicity the larger-alphabet optimization). For branching programs, each bit is encoded as a square matrix of ring elements, which is a tensor product of a matrix with 0's and 1's by a random short ring element.

We define the encoding function as $\gamma(\mathbf{M}, s)$. For permutation programs, we have $\gamma(\mathbf{M}, s) = \mathbf{M} \otimes s$. For general branching programs, $\gamma(\mathbf{M}, s) = diag(s, \mathbf{M} \otimes s)$, where *diag* refers to a function building a diagonal matrix. If u is the dimension of the matrix $\mathbf{M}$, then $\gamma(\mathbf{M}, s)$ for permutation branching programs is a $u \times u$ square matrix of ring elements, and $\gamma(\mathbf{M}, s)$ for general branching programs is a $(u+1)\times(u+1)$ square matrix of ring elements.

Next we describe the TBO algorithms focusing on the discussion of differences brought about by the encoding of matrix branching programs. To present the

Algorithm 7 Obfuscation for branching programs

function $\textsc{Obfuscate}(\{\mathbf{M}_{i,b}\}_{i\in[L],b\in\{0,1\}}, K_{MSK}, \sigma)$
  for `i = 1..`$\mathcal{L}$ **do**
    for b `= 0..`2^w`-1` **do**
      $\widehat{\mathbf{M}}_{i,b} = \prod_{j=1}^{w} \mathbf{M}_{(i-1)w+j,b_j} \in \mathbf{R}_q^{d\times d}$
      $\mathbf{D}_{i,b} := \mathsf{Encode}_{\mathbf{A}_{i-1}\to\mathbf{A}_i}(\widetilde{\mathbf{T}}_{i-1}, \gamma(\widehat{\mathbf{M}}_{i,b}, s_{i,b}), \sigma)$
  $\Pi_v := \left(\mathbf{A}_{\mathbf{J}}, \{\mathbf{D}_{i,b}\}_{i\in[\mathcal{L}],b\in\{0,\ldots,2^w-1\}}\right)$
  return Π_v

Algorithm 8 Directed encoding for matrices

function $\mathsf{Encode}_{\mathbf{A}_i\to\mathbf{A}_{i+1}}(\widetilde{\mathbf{T}}_i, \mathbf{S}\in {\mathcal{R}_q}^{d\times d}, \sigma)$
  $\mathbf{E}_{i+1} \leftarrow \mathcal{D}_{\mathcal{R},\sigma}^{d\times dm} \in {\mathcal{R}_q}^{d\times dm}$.
  $\mathbf{B}_{i+1} := \mathbf{S}\mathbf{A}_{i+1} + \mathbf{E}_{i+1} \in \mathcal{R}_q^{d\times dm}$
  $\mathbf{R}_{i+1} := \mathsf{GaussSamp}(\mathbf{A}_i, \widetilde{\mathbf{T}}_i, \mathbf{B}_{i+1}, \sigma_t, \sigma_s) \in \mathcal{R}_q^{dm\times dm}$
  return $\mathbf{R}_{i+1}$

same procedures for both types of branching programs, we use d as the dimension of $\gamma(\mathbf{M}, s)$ rather than the dimension u of the underlying matrix $\mathbf{M}$.

The key generation algorithm is listed in Algorithm 6. The main differences compared to Algorithm 1 are (1) the computation of $\mathbf{A}_J$ term, which is needed for the security of the construction for general branching programs proposed in [22], and (2) the increased dimensions for both public key and secret trapdoors (a square $d\times d$ increase as compared to the conjunction case). Note that $\mathbf{J} := (1, \mathbf{v})$ for general branching programs and $\mathbf{J} := \mathbf{I}_d$ for permutation programs. The TrapGen algorithm used in this case is a generalization for the module-LWE probem, which is discussed in Section 6.1 and Appendix E.

Algorithm 9 Evaluation by a trusted party (using the master secret key)

function $\textsc{TokenGen}(\mathbf{x}\in\{0,1\}^L, K_{MSK})$
  $\Delta := \mathbf{A}_{\mathcal{L}}[1] \prod_{i=1}^{\mathcal{L}} s_{i,x[1+(i-1)w\,:\,iw]}$
  $\mathbf{y}' := \lfloor \frac{2}{q}\Delta \rceil \in \mathbb{Z}_2^n$
  return $\mathbf{y}'$

The obfuscation and encoding procedures are presented in Algorithms 7 and 8. Conceptually the obfuscation procedure is similar to Algorithm 2 but deals with the encoding of matrices of $d\times d$ short ring elements corresponding to the matrix branching program, rather than individual short ring elements in the conjunction construction. This implies that the storage requirements are at least d^2 larger as compared to conjunctions (they are actually more due to increased noise requirements). The $\widehat{\mathbf{M}}_{i,b}$ is introduced to support a larger alphabet (word

size) when encoding the program, which is a major optimization compared to the constructons in [22] and [18].

Algorithm 9 lists the pseudocode for TokenGen, the evaluation using unconstrained key. The computational complexity is the same as for conjunctions, and $O(dm)$ smaller than for the original branching program construction [22].

Algorithm 10 Evaluation using the obfuscated program

function Eval($\mathbf{x} \in \{0,1\}^L$, Π_v, $\mathbf{y}'$)
  $\mathbf{D_\Pi} := \mathbf{A_J}$
  for `i = 1..`$\mathcal{L}$ **do**
    $\mathbf{D_\Pi} := \mathbf{D_\Pi}\,\mathbf{D}_{i,x[1+(i-1)w\,:\,iw]} \in {\mathcal{R}_q}^{1\times dm}$
  $\mathbf{y} := \lfloor \frac{2}{q}\mathbf{D_\Pi}[1] \rceil \in \mathbb{Z}_2^n$
  return $(\mathbf{y} = \mathbf{y}')$

Algorithm 10 shows the pseudocode for the evaluation using the obfuscated program (constrained key). The main difference compared to Algorithm 5 for conjunctions is that we multiply by $\mathbf{A}_J$ rather than $\mathbf{A}_0$ to satisfy the security requirements for the TBO of general branching programs. The computational complexity is $O(d^2)$ higher than in the case of conjunctions.

5.3 Security

The TBO construction for permutation branching programs is secure under Definition 2 for query-revealing TBO under Ring LWE. The proof showing that the existence of constraint-hiding constrained PRF (also referred to as private constrained PRF) implies the existence of a query-revealing TBO scheme is presented in Appendix A, and we can rely on plain RLWE since the construction from [18] only needs small-entry $\mathbf{A}$ RLWE for pseudorandomness, which is not a requirement in Definition 2. See Remark 1 in Appendix A for more details.

The security of general branching programs can be provable based on RLWE with an increase in the secret dimension and secret distribution width, from $s_{i,b} \in \mathcal{R}$ to $\mathbf{S}_{i,b} \in \mathcal{R}^{z\times z}$ for $z = \log_t(q)$, where t is the Gaussian width of $\mathbf{S}_{i,b}$. See Remark 2 in Appendix G for more details.

5.4 Parameter Selection for Matrix Branching Programs

The correctness constraint for branching programs with $\mathcal{L}$ words ($\mathcal{L} \geq 2$) is expressed as $q > 2^{10} P_e^{-1} B_J B_e \left(6\sigma_s\sqrt{dmn}\right)^{\mathcal{L}-1}$, where $B_j = d$ for general branching programs and $B_j = 1$ for permutation branching programs, and $\sigma_s = C_0\sigma\sigma_t\left(\sqrt{dn\kappa} + \sqrt{2n} + 4.7\right)$. All other parameters are the same as for the TBO of conjunctions. The derivation details are presented in Appendix D.

5.5 Efficiency of Permutation and General Branching Programs

The general branching program represention is typically significantly more efficient than the permutation representation [22]. The programs with l-bit input can be represented as general branching programs of length l. In the case of permutation programs, the length of branching programs typically has to be at least l^2 or the width has to be set to at least 2^l [22], which leads to a dramatic performance degradation when the length l is increased and makes the permutation branching program approach nonviable for most useful practical scenarios. Hence in this work we present experimental results only for general branching programs.

5.6 Application: Hamming Distance

To illustrate the TBO of general branching programs, we consider an example of obfuscating a procedure to find whether the Hamming distance between two strings of equal length K is below a certain threshold T. The Hamming distance is defined as the number of positions at which the corresponding symbols of the strings are different. We denote as $\phi \in \{0, 1, \star\}^l$ the l-bit string to be obfuscated. Note that wildcard values are allowed.

The following branching program can be used to represent this problem:

1. Initialization, for all $i \in [K]$, $b \in \{0, 1\}$, let $\mathbf{M}_{i,b} := \mathbf{I}_{T+1}$.
2. If $\phi_i = 0$, set $\mathbf{M}_{i,1} := \mathbf{N}$.
3. If $\phi_i = 1$, set $\mathbf{M}_{i,0} := \mathbf{N}$.
4. For $b \in \{0, 1\}$, set $\mathbf{M}_{l,b} := \mathbf{M}_{l,b}\mathbf{R}$.

Here, $\mathbf{N} \in \{0, 1\}^{(T+1)\times(T+1)}$ is a matrix where $N_{i,i+1} = 1$, $N_{T+1,T+1} = 1$ and all other values are set to 0; $\mathbf{R} \in \{0, 1\}^{(T+1)\times(T+1)}$ is a matrix where $R_{T+1,T+1} = 1$ and all other values are set to 0. The vector $\mathbf{v} \in \{0, 1\}^{T+1}$ is $[1\,0\,0\,\ldots\,0]$.

This branching program has the length of K and width of $T + 1$.

6 Efficient Lattice Trapdoor Algorithms

In this section, we describe the underlying lattice trapdoor mechanism used in our construction and its efficient algorithms. The trapdoor technique is an optimized instantiation of the MP12 framework [49] (which in turn is an optimized instantiation of [32]). In addition, we introduce an algorithm, SampleMat, used to sample a perturbation of arbitrary dimension over $\mathcal{R}$ efficiently. This algorithm may be of independent interest and is described in Appendix E.

The pseudocode for trapdoor generation and sampling is given in Appendix E. In short, TrapGen takes as input a security parameter and outputs a pseudorandom matrix $\mathbf{A}$ over $\mathcal{R}_q$ along with a trapdoor matrix $\mathbf{T}$ with small entries over $\mathcal{R}$. This trapdoor $\mathbf{T}$ allows us to sample discrete Gaussian preimage vectors $\mathbf{x}$ over $\mathcal{R}$ such that $\mathbf{Ax} \mod q = \mathbf{u}$ for $\mathbf{u}$ given as an input. Sampling a discrete Gaussian matrix $\mathbf{X}$ over $\mathcal{R}$ where $\mathbf{AX} = \mathbf{U} \mod q$ is done by sampling each column of $\mathbf{X}$ independently.

6.1 Perturbation Sampling for the General Covariance Matrices of Ring Elements

We now discuss SampleMat, needed to extend the efficient perturbation sampling methods of [27] to the broad, module-LWE setting. Specifically, SampleMat replaces [27]'s algorithm Sample2z. This new algorithm may be of independent interest since it samples a discrete Gaussian perturbation with a covariance described as a matrix of any dimension over the ring $\mathcal{R}$ via the Schur complement method of [27]. We remark the proof of SampleMat's statistical correctness follows from [27, Theorem 4.1], whose proof only depends on the lattice dimension and is oblivious to the underlying algebraic structure or module dimension.

6.2 RNS Algorithms

We implemented all procedures for the TBO constructions of conjunctions and branching programs in the Double-CRT (RNS) representation, which supports parallel operations over vectors of fast native integers (64-bit for x86-64 architectures). There are many benefits of using the Double-CRT representation, and new algorithms have recently been proposed [2, 7, 28, 38]. The two procedures that require special handling are the lattice trapdoor sampling in Encode and the scale-and-round operation in TokenGen and Eval.

Lattice trapdoor sampling calls digit decomposition for each polynomial coefficient in the G-sampling step. The conventional digit decomposition is not compatible with RNS, and requires expensive conversion to the positional (multi-precision) format to extract the digits. Instead, we use a CRT representation of the gadget matrix that was recently proposed in [28], which allows us to perform "digit" decomposition directly in RNS. We discuss the changes introduced by the use of CRT representation for the gadget matrix, as compared to the trapdoor algorithms in [24], in Appendix F.

For the scale-and-round operation, we utilize the RNS scaling procedure proposed in [38] for the decryption in the Brakerski/Fan-Vercauteren homomorphic encryption scheme. The technique is based on the use of floating-point operations for some intermediate computations.

7 Implementation and Results

7.1 Software Implementation

We implemented the TBO constructions in PALISADE v1.3.1 [50], an open-source lattice cryptography library. PALISADE uses a layered approach with four software layers, each including a collection of C++ classes to provide encapsulation, low inter-class coupling and high intra-class cohesion. The software layers are as follows:

1. The cryptographic layer supports cryptographic protocols such as homomorphic encryption schemes through calls to lower layers.

Table 2: Execution times and program size for a 16-feature binary classifier; n=2048, $\lceil \log_2 q \rceil = 53$, $\lambda \geq 128$.

Feature Size [bits]	Program size [KB]	OBFUSCATE [s]	TOKENGEN [ms]	EVAL [ms]
# threads = 1				
8	48	0.76	2.09	0.041
16	8,208	185	2.11	0.047
# threads = 28				
8	48	0.09	0.42	0.029
16	8,208	15.7	0.39	0.032
24	2,097,168	4,061	1.30	0.069

Table 3: Execution times and program size for conjunction obfuscation; $\lambda \geq 80$.

L [bits]	# threads	n	$\lceil \log_2 q \rceil$	$\log_2 t$	Program size [GB]	OBFUSCATE [min]	TOKENGEN [ms]	EVAL [ms]	EVALTOTAL [ms]
Token-Based Obfuscation									
32	1	4096	180	20	11.6	23.5	1.3	75.8	77.1
32	14	4096	180	20	11.6	5.1	0.6	11.0	11.6
64	28	8192	360	20	300	52.5	4.0	269.9	273.9
Optimized Distributional VBB Obfuscation [24]									
32	14	4096	180	15	36.8	12.4	–	–	53.0

2. The encoding layer supports plaintext encodings for cryptographic schemes.
3. The lattice constructs layer supports power-of-two and arbitrary cyclotomic rings (coefficient, CRT, and double-CRT representations). Lattice operations are decomposed into primitive arithmetic operations on integers, vectors, and matrices here.
4. The arithmetic layer provides basic modular operations (multiple multiprecision and native math backends are supported), implementations of Number-Theoretic Transform (NTT), Negacyclic Convolution NTT, and Bluestein FFT. The integer distribution samplers are implemented in this layer.

Our TBO toolkit is a new PALISADE module called "tbo", which includes the following new features broken down by layer:

– TBO of linear functions (binary classifiers), conjunctions, and branching programs in the cryptographic layer.
– Variants of GGH15 encoding in the encoding layer.
– Trapdoor sampling for matrices of ring elements in the lattice layer.

Several lattice-layer and arithmetic-layer optimizations are also applied for runtime improvements. OpenMP loop parallelization is used to achieve speedup in the multi-threaded mode.

Table 4: Execution times and program size for the obfuscation of the branching program that checks whether two 24-bit strings (one of them is obfuscated) have a Hamming distance less than T; # threads = 28, $n = 4096$, $\lceil \log_2 q \rceil = 180$, $\log_2 t = 20$, $\lambda \geq 80$.

T	d	Program size [GB]	Obfuscate [min]	TokenGen [ms]	Eval [ms]
1	3	76.6	26.9	0.6	55.0
2	4	136	44.8	0.6	66.6
3	5	213	72.6	0.9	133

7.2 Experimental Testbed

Experiments were performed using a server computing node with 2 sockets of Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, each with 14 cores. 500GB of RAM was accessible for the experiments. The node had Fedora 26 OS and g++ (GCC) 7.1.1 installed.

7.3 TBO of Linear Functions (Binary Classifiers)

Table 2 shows both single- and multi-threaded results for the obfuscation of a binary classifier representing a conjunction of 16 features, with the feature size being varied from 8 to 24 bits. This particular classifier can be interpreted as an image classification algorithm for 16 blocks of pixels, with the block size being varied from 1 to 3 bytes. For the underlying linear-function TBO construction, we used the distibution parameter of $8/\sqrt{2\pi}$ and $p = 2^{40}$.

The evaluation runtime, which is a sum of TokenGen and Eval, is of the order of 1 millisecond (note that it can be further improved by using a faster implementation of AES-CTR for generating secret keys on the fly) even for the single-threaded case. As the evaluation runtime depends only on the number of features, it remains almost the same when the feature size is increased from 8 to 24 bits.

The program size grows linearly with the dimension N (as predicted by our complexity analysis) and supports a highly efficient obfuscation for 16-bit features, with the obfuscation runtime of 16 seconds and program size of 8MB. The obfuscation runtime gets a relatively high speed-up in the multi-threaded mode, namely speed-ups of 8.4 and 11.8 on a 28-core machine for 8-bit and 16-bit features, respectively.

The above results imply that the obfuscation of binary classifiers is already practical, as long as the number of queries (and possibly the format of query inputs) is adequately restricted by the token generator based on the statistical properties of the classifiers being obfuscated.

7.4 TBO of Conjunctions

Table 3 presents the peformance results for the TBO of 32-bit and 64-bit conjunctions, along with the results for an optimized implementation of the distributional VBB obfuscation [17, 24] of 32-bit conjunctions for comparison.

The TBO of 32-bit conjunctions is close to being practical, with a total evaluation runtime of 11.6 milliseconds, obfuscation runtime of 5.1 minutes, and program size of 11.6 GB for a setting with more than 80 bits of security. As compared to the distributional VBB results presented in [24] for the same lattice parameters, the evaluation is 10.1x faster, obfuscation is 7.4x faster, and program size is 3.3x smaller. As TBO provides a mechanism for bounding the number of queries, this construction is also more secure. For a more complete picture, we also ran experiments for the optimized distributional VBB implementation (using the same RNS and low-level optimizations as in our TBO implementation) to provide a fair comparison of the runtimes for the TBO and distributional VBB security models. The experimental speed-ups due to the use of the TBO model are 4.6x for evaluation time and 2.4x for obfuscation time, which are somewhat higher than predicted by our high-level complexity analysis in Section 4.

We also examined the effect of OpenMP loop parallelization optimizations by comparing the results for single- and multi-threaded scenarios (Table 3). Here, we chose 14 (matching the number of cores per socket) as the number of threads as the main parallelization dimension in both evaluation and obfuscation is $m = 11$, and increasing the number of threads further than that degrades the performance due to multi-threading overhead. The speed-ups in the evaluation and obfuscation runtimes are 6.6x and 4.6x, respectively, with the maximum theoretical limit for this case being 11. This suggests there is room for further loop parallelization optimizations.

Our 64-bit conjunction obfuscation results are much further from being practical, mainly due to the large program size requirement of 300 GB. On the other hand, they are significantly better than prior distributional VBB results for the same lattice parameters. For instance, the evaluation is 9x faster, obfuscation is 7.7x faster, and program size is 2.5x smaller.

7.5 TBO of Branching Programs

Table 4 shows the performance results for the TBO of general branching programs using the Hamming distance problem as an example application. Note that $d = 5$ corresponds to the classical Barrington's theorem permutation branching program case. Hence these results can be used for benchmarking the TBO of both permutation and general branching programs of length $L = 24$ bits.

The results suggest that the program size is the main efficiency limitation of the TBO for branching programs, which is due to the large size of the GGH15 encoding matrices (in this case, we have $3d^2 \times 256$ of $m \times m$ matrices with ring elements of dimension n). Even for the case of the Hamming distance threshold of 3 and 24-bit strings, the TBO construction requires 213 GB to store the obfuscated program. At the same time, the evaluation and obfuscation runtimes are much closer to being practical.

Comparison with [37] The best prior results for general branching programs are provided by Halevi *et al.* for general read-once branching programs [37]. Although this construction was subsequently broken using a rank attack in [22], it can be used as a benchmark for comparison because the construction uses many similar building blocks (but for the case of matrices rather than rings), such as GGH15 encoding and Micciancio-Peikert lattice trapdoors. The obfuscation and evaluation times for a 24-bit program with about 100 states are 67 minutes and 13 seconds, respectively [37]. Our results for a Hamming distance program obfuscation with more than 500 states on a comparable system are 72.6 minutes and 0.13 seconds, respectively. The total storage requirements appear to be similar, but they are harder to compare due to implementation differences. In summary, our implementation evaluates more complex branching programs about two orders of magnitude faster, and is not vulnerable to known attacks.

8 Conclusion

We have presented implementation results for several TBO constructions. Some of these constructions are practical (binary classifiers based on linear functions) or close to being practical (conjunctions) while the more advanced (general branching program) constructions still need to be further improved. The important benefit of the TBO model is the ability to support the obfuscation of certain programs, such as some binary classifiers, that can be learned under the non-interactive models of VBB or IO. This is solely from the token generator's ability to limit the number of queries and restrict allowed inputs.

A potential approach for improving our results for general branching programs is to explore alternative obfuscation designs, possibly not based on multilinear maps, such as the recent method using tensor products proposed by Gentry *et al.* [31].

9 Acknowledgements

We gratefully acknowledge the input and feedback from Vinod Vaikuntanathan and Shafi Goldwasser. This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Army Research Laboratory (ARL) under Contract Numbers W911NF-15-C-0226 and W911NF-15-C-0233. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the Department of Defense or the U.S. Government.

References

1. Abdalla, M., Bourse, F., De Caro, A., Pointcheval, D.: Simple functional encryption schemes for inner products. In: Katz, J. (ed.) Public-Key Cryptography – PKC 2015. pp. 733–751. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

2. Al Badawi, A., Polyakov, Y., Aung, K.M.M., Veeravalli, B., Rohloff, K.: Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme. IEEE Transactions on Emerging Topics in Computing (2019)
3. Albrecht, M., Scott, S., Player, R.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology **9**(3), 169–203 (10 2015)
4. Apon, D., Huang, Y., Katz, J., Malozemoff, A.J.: Implementing cryptographic program obfuscation. Cryptology ePrint Archive, Report 2014/779 (2014)
5. Applebaum, B., Cash, D., Peikert, C., Sahai, A.: Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In: Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings. pp. 595–618 (2009). https://doi.org/10.1007/978-3-642-03356-8_35, `https://doi.org/10.1007/978-3-642-03356-8\_35`
6. Bahler, L., Di Crescenzo, G., Polyakov, Y., Rohloff, K., Cousins, D.B.: Practical implementation of lattice-based program obfuscators for point functions. In: HPCS 2017. pp. 761–768 (2017)
7. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full rns variant of fv like somewhat homomorphic encryption schemes. In: Avanzi, R., Heys, H. (eds.) Selected Areas in Cryptography – SAC 2016. pp. 423–442 (2017)
8. Banerjee, A., Peikert, C., Rosen, A.: Pseudorandom functions and lattices. In: EUROCRYPT 2012. pp. 719–737 (2012)
9. Barak, B.: Hopes, fears, and software obfuscation. Commun. ACM **59**(3), 88–96 (Feb 2016)
10. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. J. ACM **59**(2), 6:1–6:48 (May 2012)
11. Bellare, M., Stepanovs, I.: Point-function obfuscation: A framework and generic constructions. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. pp. 565–594 (2016)
12. Bishop, A., Jain, A., Kowalczyk, L.: Function-hiding inner product encryption. In: ASIACRYPT 2015. pp. 470–491 (2015)
13. Bitansky, N., Canetti, R., Cohn, H., Goldwasser, S., Kalai, Y.T., Paneth, O., Rosen, A.: The impossibility of obfuscation with auxiliary input or a universal simulator. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. pp. 71–89 (2014)
14. Bitansky, N., Canetti, R., Goldwasser, S., Halevi, S., Kalai, Y.T., Rothblum, G.N.: Program obfuscation with leaky hardware. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. pp. 722–739 (2011)
15. Boneh, D., Lewi, K., Montgomery, H.W., Raghunathan, A.: Key homomorphic prfs and their applications. In: CRYPTO 2013. pp. 410–428 (2013)
16. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Innovations in Theoretical Computer Science 2012. pp. 309–325 (2012)
17. Brakerski, Z., Vaikuntanathan, V., Wee, H., Wichs, D.: Obfuscating conjunctions under entropic ring lwe. In: Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science. pp. 147–156. ITCS '16 (2016)
18. Canetti, R., Chen, Y.: Constraint-hiding constrained prfs for nc1 from lwe. Cryptology ePrint Archive, Report 2017/143 (2017), `https://eprint.iacr.org/2017/143`
19. Carmer, B., Malozemoff, A.J., Raykova, M.: 5gen-c: Multi-input functional encryption and program obfuscation for arithmetic circuits. In: ACM CCS'17. pp. 747–764 (2017)

20. Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Hoffstein, J., Lauter, K., Lokam, S., Moody, D., Morrison, T., Sahai, A., Vaikuntanathan, V.: Security of homomorphic encryption. Tech. rep., HomomorphicEncryption.org, Redmond WA (July 2017)
21. Chen, Y., Gentry, C., Halevi, S.: Cryptanalyses of candidate branching program obfuscators. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017. pp. 278–307 (2017)
22. Chen, Y., Vaikuntanathan, V., Wee, H.: Ggh15 beyond permutation branching programs: Proofs, attacks, and candidates. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. pp. 577–607 (2018)
23. Coron, J.S., Lee, M.S., Lepoint, T., Tibouchi, M.: Zeroizing attacks on indistinguishability obfuscation over clt13. In: Fehr, S. (ed.) PKC 2017. pp. 41–58 (2017)
24. Cousins, D.B., Crescenzo, G.D., Gür, K.D., King, K., Polyakov, Y., Rohloff, K., Ryan, G.W., Savas, E.: Implementing conjunction obfuscation under entropic ring lwe. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 354–371 (2018)
25. Crescenzo, G.D., Bahler, L., Coan, B.A., Polyakov, Y., Rohloff, K., Cousins, D.B.: Practical implementations of program obfuscators for point functions. In: HPCS 2016. pp. 460–467 (2016)
26. Garg, S., Miles, E., Mukherjee, P., Sahai, A., Srinivasan, A., Zhandry, M.: Secure obfuscation in a weak multilinear map model. In: Hirt, M., Smith, A. (eds.) TCC 2016. pp. 241–268 (2016)
27. Genise, N., Micciancio, D.: Faster gaussian sampling for trapdoor lattices with arbitrary modulus. In: EUROCRYPT 2018. pp. 174–203 (2018)
28. Genise, N., Micciancio, D., Polyakov, Y.: Building an efficient lattice gadget toolkit: Subgaussian sampling and more. Cryptology ePrint Archive, Report 2018/946 (2018), `https://eprint.iacr.org/2018/946`
29. Gentry, C., Gorbunov, S., Halevi, S.: Graph-induced multilinear maps from lattices. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. pp. 498–527 (2015)
30. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. pp. 850–867 (2012)
31. Gentry, C., Jutla, C.S., Kane, D.: Obfuscation using tensor products. Cryptology ePrint Archive, Report 2018/756 (2018), `https://eprint.iacr.org/2018/756`
32. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: STOC '08. pp. 197–206 (2008)
33. Goldwasser, S., Kalai, Y.T.: On the impossibility of obfuscation with auxiliary input. In: FOCS'05. pp. 553–562 (Oct 2005)
34. Goldwasser, S., Kalai, Y., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: STOC '13. pp. 555–564 (2013)
35. Goyal, V., Ishai, Y., Sahai, A., Venkatesan, R., Wadia, A.: Founding cryptography on tamper-proof hardware tokens. In: Micciancio, D. (ed.) Theory of Cryptography. pp. 308–326. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
36. Hada, S.: Zero-knowledge and code obfuscation. In: Okamoto, T. (ed.) ASIACRYPT 2000. pp. 443–457 (2000)
37. Halevi, S., Halevi, T., Shoup, V., Stephens-Davidowitz, N.: Implementing bp-obfuscation using graph-induced encoding. In: ACM CCS '17. pp. 783–798 (2017)
38. Halevi, S., Polyakov, Y., Shoup, V.: An improved rns variant of the bfv homomorphic encryption scheme. In: Matsui, M. (ed.) CT-RSA 2019. pp. 83–105 (2019)
39. Kim, S., Lewi, K., Mandal, A., Montgomery, H., Roy, A., Wu, D.J.: Function-hiding inner product encryption is practical. In: Catalano, D., De Prisco, R. (eds.) Security and Cryptography for Networks. pp. 544–562 (2018)

40. Kubat, M.: An Introduction to Machine Learning. Springer Publishing Company, Incorporated, 1st edn. (2015)
41. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Des. Codes Cryptography **75**(3), 565–599 (2015)
42. Lewi, K., Malozemoff, A.J., Apon, D., Carmer, B., Foltzer, A., Wagner, D., Archer, D.W., Boneh, D., Katz, J., Raykova, M.: 5gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In: ACM CCS '16. pp. 981–992 (2016)
43. Lin, H.: Indistinguishability obfuscation from sxdh on 5-linear maps and locality-5 prgs. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. pp. 599–629 (2017)
44. Lin, H., Pass, R., Seth, K., Telang, S.: Indistinguishability obfuscation with non-trivial efficiency. In: Cheng, C.M., Chung, K.M., Persiano, G., Yang, B.Y. (eds.) PKC 2016. pp. 447–462 (2016)
45. Lin, H., Tessaro, S.: Indistinguishability obfuscation from trilinear maps and block-wise local prgs. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. pp. 630–660 (2017)
46. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. pp. 1–23 (2010)
47. Lyubashevsky, V., Peikert, C., Regev, O.: A toolkit for ring-LWE cryptography. In: EUROCRYPT. vol. 7881, pp. 35–54. Springer (2013)
48. Micciancio, D.: On the hardness of learning with errors with binary secrets. Theory of Computing **14**(1), 1–17 (2018)
49. Micciancio, D., Peikert, C.: Trapdoors for lattices: Simpler, tighter, faster, smaller. In: Advances in Cryptology–EUROCRYPT 2012, pp. 700–718. Springer (2012)
50. Polyakov, Y., Rohloff, K., Ryan, G.W.: PALISADE lattice cryptography library. `https://git.njit.edu/palisade/PALISADE` (Accessed November 2018)
51. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6), 34:1–34:40 (2009)
52. Xiao, Y., Mehrotra, K.G., Mohan, C.K.: Efficient classification of binary data stream with concept drifting using conjunction rule based boolean classifier. In: Ali, M., Kwon, Y.S., Lee, C.H., Kim, J., Kim, Y. (eds.) IEA/AIE 2015. pp. 457–467 (2015)

Auxiliary Supporting Material

A Query-Revealing TBO from Constraint-Hiding Constrained PRF

Here we prove that the existence of a constraint-hiding pseudorandom function (CHPRF) implies the existence of a query-revealing TBO scheme. First, we recall the definition of a (one-key) CHPRF [18, 22].

Definition 7. *Consider a family of functions* $\{\mathcal{F}_\lambda\}$, $\mathcal{F}_\lambda = \{F_k : D_\lambda \to R_\lambda\}$ *is a set of keyed functions, and a constraint family* $\mathcal{C} = \{\mathcal{C}_\lambda\}$ *where* $\mathcal{C}_\lambda = \{C : D_\lambda \to \{0,1\}\}$ *is a set of circuits. Let*

$$(\textsc{Gen}, \textsc{Constrain}, \textsc{Eval}, \textsc{Constrain.Eval})$$

be a tuple of algorithms such that Eval *and* Constrain.Eval *are deterministic, the PPT* $\textsc{Gen}(1^\lambda)$ *returns a master secret key* msk*, and the PPT returns a constrained key* CK_C*. Further,* $\textsc{Eval}(\mathrm{msk}, x) = F_{\mathrm{msk}}(x)$ *and* $\textsc{Constrain.Eval}(\mathrm{CK}_C, x) = F_{\mathrm{CK}_C}(x)$*. We say the tuple of efficient algorithms* (Gen, Constrain, Eval, Constrain.Eval) *is a CHPRF for* $\{\mathcal{F}_\lambda\}$ *if:*

1. *For all inputs* $x \in D_\lambda$ *s.t.* $C(x) = 1$*, we have* $\Pr\{\textsc{Eval}(\mathrm{msk}, x) = \textsc{Constrain.Eval}(\mathrm{CK}_C, x)\} \geq 1 - \mathrm{neglible}(\lambda)$ *where the probability is taken over* Gen *and* Constrain*'s random coins.*
2. *There exists a PPT simulator pair* $(\mathrm{S}_1, \mathrm{S}_2)$ *such that for all PPT adversaries* $(\mathrm{A}_1, \mathrm{A}_2)$*, the outputs of* $\mathrm{Real}_{\mathrm{CHPRF,A}}(1^\lambda)$ *and* $\mathrm{Ideal}_{\mathrm{CHPRF,A,S}}(1^\lambda)$ *are computationally indistinguishable (Figure 2).*

The simulator S_1 *takes as input the security parameter and the circuit size, and outputs a fake constrained key* CK^*_C *as well as a state* st_S*. Next, the simulator* S_2 *takes as input an* $x \in D_\lambda$*,* $C(x)$*, where* $C : D_\lambda \to \{0,1\}$ *is the circuit chosen by the adversary, a state* st_S*, and it returns a fake evaluation* y *and an updated state* st'_S*. The oracle* $\mathrm{OP}(\cdot, C)[[\mathrm{st}_S]]$ *takes as input* $x \in D_\lambda$ *and runs* $(y, \mathrm{st}'_S) \leftarrow \mathrm{S}_2(x, C(x), \mathrm{st}_S)$*. Then, it updates its internal state to* st'_S *and returns* y *if* $C(x) = 1$*, or it returns a uniformly sampled element in the range,* $u \leftarrow \mathcal{U}(R_\lambda)$*, if* $C(x) = 0$*. Further, we assume* Constrain.Eval *is implicit given* CK_C*.*

Theorem 2. *The existence of a one-key CHPRF scheme for a class of circuits* $\{\mathcal{C}_\lambda\}$ *and an* R_λ *with* $|R_\lambda| = \lambda^{\omega(1)}$ *implies the existence of a query-revealing token-based obfuscation scheme for the same class of circuits,* $\{\mathcal{C}_\lambda\}$*.*

Proof. **Functionality.** Given a CHPRF scheme (Gen, Constrain, Eval, Constrain.Eval), construct the TBO scheme

$$(\textsc{Setup}', \textsc{Obfuscate}', \textsc{TokenGen}')$$

as follows:

$\underline{\mathbf{Real}_{\mathbf{CHPRF,A}}(1^\lambda)\mathbf{:}}$

$\mathrm{msk} \leftarrow \textsc{Gen}(1^\lambda)$
$(\mathrm{C}, \mathrm{st}_A) \leftarrow \mathrm{A}_1(1^\lambda)$
$\mathrm{CK}_C \leftarrow \textsc{Constrain}(\mathrm{msk}, \mathrm{C})$
$\alpha \leftarrow \mathrm{A}_2^{\textsc{Eval}(\mathrm{msk},\cdot)}(\mathrm{C}, \mathrm{CK}_C, \mathrm{st}_A)$
Return α

$\underline{\mathbf{Ideal}_{\mathbf{CHPRF,A,S}}(1^\lambda)\mathbf{:}}$

$(\mathrm{C}, \mathrm{st}_A) \leftarrow \mathrm{A}_1(1^\lambda)$
$(\mathrm{CK}_C^*, \mathrm{st}_S) \leftarrow \mathrm{S}_1(1^\lambda, 1^{|C|})$
$\alpha \leftarrow \mathrm{A}_2^{\mathrm{OP}(\cdot,C)[[\mathrm{st}_S]]}(\mathrm{C}, \mathrm{CK}_C^*, \mathrm{st}_A)$
Return α

Fig. 2: The one-key CHPRF security games.

- $\textsc{Setup}'(1^\lambda)$ returns $\mathrm{osk} \leftarrow \textsc{Gen}(1^\lambda)$.
- Given a secret key, osk, and a circuit, $\textsc{Obfuscate}'(\mathrm{osk}, \mathrm{C})$ returns a constrained key as the obfuscated circuit $\mathrm{O} \leftarrow \textsc{Contrain}(\mathrm{osk}, \mathrm{C})$.
- Next, we define $\textsc{TokenGen}'(\mathrm{osk}, x)$ to return the function evaluation as the token $\mathrm{tk}_x \leftarrow \textsc{Eval}(\mathrm{osk}, x)$.
- Finally, we evaluate the obfuscated circuit on x by checking

$$\textsc{Constrain.Eval}(\mathrm{CK}_C, x) = \mathrm{tk}_x$$

$(F_{\mathrm{osk}}(x) = F_{\mathrm{CK}_C}(x))$.

By the correctness of the CHPRF scheme, we have $O(tk_x) = 1 = C(x)$ with $1 - \mathrm{negl}(\lambda)$ probability whenever $C(x) = 1$. Further, we have $F_{\mathrm{osk}}(x) \neq F_{\mathrm{CK}_C}(x)$ when $C(x) = 0$ with high probability since $|R_\lambda| = \lambda^{\omega(1)}$ along with the PRF property[9]. The rest of the proof follows from the definition of the security games for qr-TBO and (one-key) CHPRF.

Real games. First, we show for all adversaries the real games have the same distribution. Consider a fixed adversary $(\mathrm{A}_1, \mathrm{A}_2)$, then the distribution

$$\{(\mathrm{osk}, \mathrm{C}, \mathrm{st}_A, \mathrm{CK}_C, \alpha) : \mathrm{Real}_{\mathrm{CHPRF,A}}(1^\lambda)\}$$

is exactly the distribution generated in the real qr-TBO game (Definition 3) with adversary $(\mathrm{A}_1, \mathrm{A}_2)$,

$$\{(\mathrm{osk}, \mathrm{C}, \mathrm{st}_A, \mathrm{O}, \alpha) : \mathrm{Exp}^{\mathrm{real}}_{\mathrm{tOB,A}}(1^\lambda)\}.$$

Ideal games. Next, we consider the ideal CHPRF game and show for all simulators and adversaries, there exists a simulator pair so the ideal games have the same distribution (with the adversaries unchanged). Let $(\mathrm{S}_1, \mathrm{S}_2)$ be PPT simulators and $(\mathrm{A}_1, \mathrm{A}_2)$ be PPT adversaries again. Let $\mathrm{S}_1^{\mathrm{TBO}} := \mathrm{S}_1$. Then, let $\mathrm{S}_2^{\mathrm{TBO}}(x, C(x), \mathrm{st}_S)$ consist of the following steps:

1. First, it runs S_2, $(y, \mathrm{st}'_S) \leftarrow \mathrm{S}_2(x, C(x), \mathrm{st}_S)$.

[9] Here we remark that the PRF property is *stronger* than what is needed for the proof to go through. Specifically, a min-entropy argument here would suffice.

2. Next, it will return (y, st'_S) if $C(x) = 1$, or it will overwrite y with a uniformly sampled element in the range, $u \leftarrow \mathcal{U}(R_\lambda)$, and return (u, st_S) if $C(x) = 0$.

Now, the distribution of $\{(\mathrm{C}, \mathrm{st}_A, \mathrm{CK}^*_C, \alpha)\}$ in the ideal-CHPRF is the same as the distribution of $\{(\mathrm{C}, \mathrm{st}_A, \mathrm{O}^*, \alpha)\}$ in the ideal game of the qr-TBO game. (The description of $\mathrm{S}_2^{\mathrm{TBO}}$ merely accounts for the differing behaviors of the query/evaluation oracles in the two games.)

Bridging the games. Finally, we let $(\mathrm{S}_1, \mathrm{S}_2)$ be the PPT simulators such that the real and ideal CHPRF games are computationally indistinguishable. The equivalences given above show the ideal qr-TBO game with simulators $(\mathrm{S}_1^{\mathrm{TBO}}, \mathrm{S}_2^{\mathrm{TBO}})$ is computationally indistinguishable from the real qr-TBO game for any adversary $(\mathrm{A}_1, \mathrm{A}_2)$. □

Remark 1. Note, the correctness of the qr-TBO scheme does not need the PRF property (a security property) when $C(x) = 0$. Instead, all we need is $\textsc{Eval}(\mathrm{osk}, x) \neq \textsc{Constrain.Eval}(\mathrm{CK}_C, x)$ with high probability whenever $C(x) = 0$, a much weaker property than being a PRF. This freedom from the PRF requirement allows us to base the security of our conjunctions and permutation branching programs on regular RLWE and *not* "non-uniform" RLWE (Gaussian $\mathbf{A}$ instead of uniformly random as used in [15]). This is given explicitly by Theorems 5.4 and 5.8 in [18][10]. Correctness for our schemes based on [18] was confirmed experimentally.

B Complexity of TBO for Binary Classifiers

We examine the complexity for the more general case of the conjunction of multiple attributes. Let P be the number of features.

Storage. The size of the secret key is n integers of r bytes (in our implementation $r = 8$ as the correctness can be achieved using native 64-bit arithmetic). The size of the obfuscated program is $N + n$ integers. The size of $\mathbf{w}$ and $\mathbf{x}$ is N integers. The token size is n integers.

Computational complexity. We focus on the computational complexity for Obfuscate, TokenGen, and Eval as the key generation time is negligible and this operation is done offline. The Obfuscate procedure computes N inner products of n-sized vectors and has the complexity of $O(N \cdot n)$ integer multiplications and additions. The TokenGen procedure adds P vectors of size n (all but P components of vector $\mathbf{x}$ are 0), i.e., it has the complexity of $O(P \cdot n)$ additions. In practice, the TokenGen procedure may also compute the secret keys on the fly. Note that only P such secret keys are needed. So the total complexity may be $O(P \cdot n)$ additions + $O(P \cdot n)$ pseudorandom number generations. The Eval procedure performs an inner product and $P - 1$ additions, i.e., it has the complexity of $O(n)$ integer multiplications.

[10] This is reflected in their proofs which treat the simulators separately via distinct lemmas (Lemmas 5.5 and 5.9 in [18], respectively), reducing security directly to GLWE.

Scalability. The storage requirements for an obfuscated program scale linearly with N. For example, if $r = 8$, then a 16GB system can support N up to 2.15×10^9. The obfuscation time scales linearly with N. The two main (online) operations TokenGen and Eval do not depend on N and scale linearly with P. The above analysis suggests that this obfuscator can efficiently support relatively large dimensions (up to 2^{24}–2^{32}), hence the classification for 32-bit features can be supported on modern server systems.

C Noise Analysis for Token-Based Obfuscation of Conjunctions

The bound B on the noise introduced by error terms in the GGH15 encoding (for the case of conjunctions) can be estimated as follows:

$$\left\| \mathbf{A}_0 \prod_{i=1}^{\mathcal{L}} \mathbf{D}_{i,x_i} - \prod_{i=1}^{\mathcal{L}} s_{i,x_i} \cdot \mathbf{A}_L \right\|_\infty = \left\| \sum_{j=1}^{\mathcal{L}} \left(\prod_{i=1}^{j-1} s_{i,x_i} \cdot \mathbf{e}_{j,x_j} \cdot \prod_{k=j+1}^{\mathcal{L}} \mathbf{D}_{k,x_k}) \right) \right\|_\infty \leq 6\sigma\mathcal{L} \left(6\sigma_s\sqrt{mn}\right)^{\mathcal{L}-1}.$$

Here, we used the Central Limit Theorem (subgaussian analysis) and the following bounds:

$$\|s_{i,x_i}\|_\infty \leq 6\sigma, \left\|\mathbf{e}_{j,x_j}\right\|_\infty \leq 6\sigma, \|\mathbf{D}_{k,x_k}\|_\infty \leq 6\sigma_s.$$

Using the fact that $\|\mathbf{D}_{k,x_k}\|_\infty \gg \|s_{i,x_i}\|_\infty$, yields the bound $B := 12\sigma \left(6\sigma_s\sqrt{mn}\right)^{\mathcal{L}-1}$.

For the rounding to work correctly, we set $q \geq 2p\alpha B/P_e$, where α is the number of bits used in comparing the PRF values and P_e is the probability of a rounding error for one polynomial coefficient. We set $\alpha = 128$ and $P_e = 2^{-20}$, i.e., assume that the number of queries is bounded by 2^{20}.

D Noise Analysis for Token-Based Obfuscation of Branching Programs

The bound B on the noise introduced by error terms in the GGH15 encoding (for the case of branching programs) can be estimated as follows:

$$\left\| \mathbf{A}_0 \prod_{i=1}^{\mathcal{L}} \mathbf{D}_{i,x_i} - \prod_{i=1}^{\mathcal{L}} \gamma(\widehat{\mathbf{M}}_{i,x_i}, s_{i,x_i}) \cdot \mathbf{A}_L \right\|_\infty = \left\| \sum_{j=1}^{\mathcal{L}} \left(\prod_{i=1}^{j-1} \gamma(\widehat{\mathbf{M}}_{i,x_i}, s_{i,x_i}) \cdot \mathbf{E}_{j,x_j} \cdot \prod_{k=j+1}^{\mathcal{L}} \mathbf{D}_{k,x_k}) \right) \right\|_\infty \leq 6\sigma\mathcal{L} \left(6\sigma_s\sqrt{dmn}\right)^{\mathcal{L}-1}.$$

Here, we used the Central Limit Theorem (subgaussian analysis) and the following bounds:

$$\left\|\gamma(\widehat{\mathbf{M}}_{i,x_i}, s_{i,x_i})\right\|_\infty \leq 6\sigma, \left\|\mathbf{E}_{j,x_j}\right\|_\infty \leq 6\sigma, \left\|\mathbf{D}_{k,x_k}\right\|_\infty \leq 6\sigma_s.$$

Using the fact that $\left\|\mathbf{D}_{k,x_k}\right\|_\infty \gg \left\|\gamma(\widehat{\mathbf{M}}_{i,x_i}, s_{i,x_i})\right\|_\infty$ and adding the multiplicative term $\mathbf{J}$, yields the bound $B := 12\sigma d\left(6\sigma_s\sqrt{dmn}\right)^{\mathcal{L}-1}$ for general branching programs (for permutation branching programs, the factor d is removed).

For the rounding to work correctly, we set $q \geq 2p\alpha B/P_e$, where α is the number of bits used in comparing the PRF values and P_e is the probability of a rounding error for one polynomial coefficient. We set $\alpha = 128$ and $P_e = 2^{-20}$, i.e., assume that the number of queries is bounded by 2^{20}.

E Trapdoor Algorithms for Matrices of Ring Elements

E.1 Preliminaries

Let $\mathcal{K}$ be the corresponding power of two cyclotomic field to the cyclotomic ring $\mathcal{R}$, $\mathcal{K} = \mathbb{Q}[x]/\langle x^{2n}+1\rangle$. Here we describe the trapdoor preimage sampling technique used in our implementation, an instantiation of [49]. These are mainly the algorithms of [27], but here we replace the algorithm SAMPLE2Z in [27], which samples two-dimensional ring perturbations with covariances in $\mathcal{K}^{2\times 2}$, with a more generic algorithm for a larger dimension. We remark that the proof of correctness regarding the statistical properties of the sample is the same as [27, Section 4]. This broadens the efficient trapdoor sampling methods of [27] to GLWE/MLWE.

G-Lattice Sampling We will be sampling discrete Gaussians over lattices and lattice cosets, whose width is larger than the smoothing parameter (defined below). Informally, the smoothing parameter of a lattice is the smallest width for which a discrete Gaussian over the lattice behaves like a continuous Gaussian. Efficiently sampling discrete Gaussians over lattices above the smoothing parameter was first rigorously analyzed by Gentry et al. [32].

Definition 8. *For an $\varepsilon > 0$ and a lattice L, the* ε-smoothing parameter *is the smallest $s > 0$ such that $\rho(s \cdot L^*) \leq 1 + \varepsilon$.*

Let $\kappa = \lceil \log_t q \rceil$, and let $\mathbf{G} = \mathbf{I}_l \otimes \mathbf{g}^T \in {\mathcal{R}_q}^{l\times l\kappa}$ be the "power-of-t" G-matrix, a block diagonal matrix with $\mathbf{g}^T = (1, t, \cdots, t^{\kappa-1})$ as the non-zero blocks. Then, the G-lattice is $\Lambda_q^\perp(\mathbf{G}) = \{\mathbf{x} \in \mathcal{R}^\kappa : \mathbf{G}\mathbf{x} = \mathbf{0} \in {\mathcal{R}_q}^l\}$. For any $\mathbf{u} \in {\mathcal{R}_q}^l$, we have the coset $\Lambda_{\mathbf{u}}^\perp(\mathbf{G}) = \{\mathbf{x} \in \mathcal{R}^{l\kappa} : \mathbf{G}\mathbf{x} = \mathbf{u} \in {\mathcal{R}_q}^l\}$. We will need the following, G-lattice sampling lemma.

Lemma 1. *([27, 49]) For any $\sigma > (t+1)\omega(\sqrt{\log nl})$, there is a probabilistic $O(\kappa)$-time algorithm whose output is distributed statistically close to $\mathcal{D}_{\Lambda_{\mathbf{u}}^\perp(\mathbf{G}),\sigma}$.*

E.2 Main Procedures

Here we describe the trapdoor generation and discrete Gaussian sampling procedures. The latter contains a new algorithm for sampling perturbations with covariances described as $d \times d$ matrices over $\mathcal{R}$ (compared to only 2×2 matrices as in [27]).

Our trapdoor construction is identical to the original MP12 construction [49] for RLWE. Specifically, we are sampling the "computational instantiation" described in Section 5 of [49].

TrapGen simply takes as input a security parameter λ and performs the following:

1. Sample a uniformly random matrix $\bar{\mathbf{A}} \leftarrow \mathcal{U}(\mathcal{R}_q^{d \times d})$.
2. Sample RLWE secrets $\mathbf{R} \in \mathcal{R}^{d \times d\kappa}$ and RLWE errors $\mathbf{E} \in \mathcal{R}^{d \times d\kappa}$, both having discrete Gaussian entries in $\mathcal{R}$.
3. Return the trapdoor, $\mathbf{T} := (\mathbf{R}, \mathbf{E})$, and the public matrix $\mathbf{A} = [\mathbf{A}' | \mathbf{G} - \mathbf{A}'\mathbf{T}]$ where $\mathbf{G}$ is the common "gadget" matrix and $\mathbf{A}' = (\bar{\mathbf{A}}, \mathbf{I})$.

Algorithm 11 Trapdoor generation using MLWE for G lattice; $\kappa = \log_t q$

function TrapGen(1^λ)
  $\bar{\mathbf{A}} \leftarrow \mathcal{U}_q \in \mathcal{R}_q^{d \times d}$
  $\mathbf{R} := [\mathbf{r}_1, \ldots, \mathbf{r}_\kappa] \leftarrow \mathcal{D}_{\mathcal{R}^{d \times d}, \sigma} \in \mathcal{R}_q^{d \times d\kappa}$
  $\mathbf{E} := [\mathbf{e}_1, \ldots, \mathbf{e}_\kappa] \leftarrow \mathcal{D}_{\mathcal{R}^{d \times d}, \sigma} \in \mathcal{R}_q^{d \times d\kappa}$
  $\mathbf{A} := [\bar{\mathbf{A}}, \mathbf{I}_d, \mathbf{G} - (\bar{\mathbf{A}}\mathbf{R} + \mathbf{E})] \in \mathcal{R}_q^{d \times d(2+\kappa)}$
  $\mathbf{T} := (\mathbf{R}, \mathbf{E}) \in \mathcal{R}^{2d \times d\kappa}$
  return $(\mathbf{A}, \mathbf{T})$

Algorithm 12 Trapdoor Sampling

function GaussSamp($\mathbf{A}, \mathbf{T}, \mathbf{b}, \sigma_t, s$)
  Sample a perturbation $\mathbf{p} \leftarrow$SamplePert($\Sigma_d, \mathbf{T}, s, \eta$).
  Set a G-lattice coset $\mathbf{v} \leftarrow \mathbf{p} - \mathbf{A}\mathbf{p} \in {\mathcal{R}_q}^d$.
  Sample the G-lattice $\mathbf{z} \leftarrow$SampleG($\mathbf{v}$).
  return $\mathbf{p} + \begin{bmatrix} \mathbf{T} \\ \mathbf{I} \end{bmatrix} \mathbf{z}$.

We use a standard, t-ary definition of the gadget matrix $\mathbf{G} = \mathbf{I}_d \otimes \mathbf{g}^T$, where $\mathbf{g}^T = \left\{1, t, \ldots, t^{\kappa-1}\right\}$. This generalizes to the RNS form in a straightforward manner, presented in [28] and here in Appendix F.

Perturbation Sampling Here we describe the perturbation algorithm, which takes as input a structured covariance matrix Σ (described as ring elements/ polynomials), and returns a discrete Gaussian vector over the integers with the input covariance. The algorithms presented here are the techniques of [27] adapted to larger matrices over $\mathcal{R}$. They use an FFT-like technique to sample smaller and smaller structured covariances. Our techniques differ in that we introduce an intermediate algorithm SampleMat for this generalization, which takes the place of Sample2z in [27].

The main algorithm is Algorithm 13, or SamplePert, which given a trapdoor matrix $\mathbf{T}$ over $\mathcal{R}$ and a bound s, returns a discrete Gaussian perturbation $(\mathbf{p}, \mathbf{q})$ with covariance

$$\Sigma = \begin{bmatrix} s^2\mathbf{I} - \eta^2\mathbf{T}\mathbf{T}^T & -\eta^2\mathbf{T} \\ -\eta^2\mathbf{T}^T & (s^2-\eta^2)\mathbf{I} \end{bmatrix}$$

where s is greater than the largest singular value of the trapdoor $\mathbf{T}$ and $\eta = \eta_\epsilon$ is the smoothing parameter of the G-lattice [49]. It calls a subroutine, Algorithm 14 or SampleMat, which given a positive definite matrix $\Sigma_d \in \mathcal{K}^{d\times d}$, returns a discrete Gaussian perturbation with covariance Σ_d. In the case of SamplePert, $\Sigma_d = s^2\mathbf{I} - \eta^2\mathbf{T}\mathbf{T}^T$.

SampleMat is a recursive algorithm which calls a function SampleF, Algorithm 15, at its base case. SampleF takes as input a field element f representing a covariance as well as a field element c and returns a discrete Gaussian sample with covariance f centered at c. SampleF is identical to [27]. SampleMat, however, breaks its input matrix into

$$\Sigma_d = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{D} \end{bmatrix}$$

and follows the Schur-complement sampling method from [27]. The matrices $\mathbf{A}, \mathbf{D}, \mathbf{B} \in \mathcal{K}^{d/2\times d/2}$ for an even dimension d. For an odd dimension, $\mathbf{A} \in \mathcal{K}^{\lceil d/2\rceil\times\lceil d/2\rceil}$, $\mathbf{D} \in \mathcal{K}^{\lfloor d/2\rfloor\times\lfloor d/2\rfloor}$, and $\mathbf{B} \in \mathcal{K}^{\lceil d/2\rceil\times\lfloor d/2\rfloor}$. The last algorithm called is a one-dimensional discrete Gaussian sampler, SampleZ.

Parameters Here we give the parameters for which these trapdoor algorithms are correct. Let $\varepsilon > 0$ be some error term and let $C_{\varepsilon,N} = \sqrt{\frac{\log(2N(1+1/\varepsilon))}{\pi}}$ be our approximation for the smoothing parameter of $\mathbb{Z}^N$. The G-lattice Gaussian width satisfies $\sigma_t \geq (t+1)C_{\varepsilon,d\kappa}$ [27]. Then, the parameter s in Algorithm 13 must satisfy $s^2 \geq \sigma_t^2(s_1(\mathbf{T})^2+1)+C^2_{\varepsilon,d(2+\kappa)}$, where $s_1(\mathbf{T})$ denotes the trapdoor's largest singular value.

Algorithm 13 Perturbation Sampling

function SamplePert($\Sigma_d, \mathbf{T}, s, \eta$)
  for $i = 0, \cdots, d^2\kappa n - 1$ **do**
    $q_i \leftarrow$SampleZ($s^2 - \eta^2$)
  $\mathbf{c} := \frac{-\eta^2}{s^2-\eta^2}\mathbf{Tq}$
  $\mathbf{p} \leftarrow$SampleMat($\Sigma_d, \mathbf{c}$)
  return $(\mathbf{p}, \mathbf{q})$

Algorithm 14 Perturbation Sampling, Matrix

function SampleMat($\Sigma, \mathbf{c}$)
  if $d = 1$ **then**
    return SampleF($\Sigma, \mathbf{c}$)
  $\mathbf{c} = (\mathbf{c}_0, \mathbf{c}_1) \in \mathcal{K}^d$
  $\mathbf{q}_1 \leftarrow$ SampleMat($\mathbf{D}, \mathbf{c}_1$).
  $\Sigma' := \mathbf{A} - \mathbf{BD}^{-1}\mathbf{B}^T$.
  $\mathbf{q}_0 \leftarrow$SampleMat($\Sigma', \mathbf{c}_0 + \mathbf{BD}^{-1}(\mathbf{q}_1 - \mathbf{c}_1)$).
  return $(\mathbf{q}_0, \mathbf{q}_1)$

Algorithm 15 Perturbation Sampling, Field Element

function SampleF(f, c)
  if $n = 1$ **then**
    return SampleZ(f, c)
  Let $f(x) = f_0(x^2) + xf_1(x^2)$.
  Let $c(x) = c_0(x^2) + xc_1(x^2)$.
  $q_1 \leftarrow$ SampleF(f_0, c_1).
  $c_0 := c_0 + f_1 f_0^{-1}(q_1 - c_1)$.
  $q_0 \leftarrow$ SampleF($f_0 - xf_1^2 f_0^{-1}, c_0$).
  return (q_0, q_1)

F Trapdoor Algorithms in CRT

F.1 Trapdoor Generation

The TrapGen procedure is the same as described in Algorithm 1 of [24] but w.r.t. the CRT gadget vector $\mathbf{g}_{CRT}^T$ rather than the regular gadget vector $\mathbf{g}^T = \{1, t, t^2, \ldots, t^\kappa\}$.

The CRT gadget vector $\mathbf{g}_{CRT}^T$ used in our implementation is described as follows. For each coprime factor q_i, fix the *base-t* gadget vector as $\mathbf{g}_i^T := (1, t, \cdots, t^{\kappa_i - 1})$ where $\kappa_i = \lceil \log_t(q_i) \rceil$. Let $\kappa = \sum_i \kappa_i$, $q_i^* = q/q_i$, and $\hat{q}_i = (q_i^*)^{-1} \mod q_i$. We then define the CRT gadget vector $\mathbf{g}_{CRT}^T = (q_1^* \hat{q}_1 \cdot \mathbf{g}_1^T, \cdots, q_l^* \hat{q}_l \cdot \mathbf{g}_l^T) \mod q \in \mathbb{Z}_q^{1 \times \kappa}$ [28].

Note that in the implementation the $q_i^* \hat{q}_i$ factors are dropped because $(q_i^* \hat{q}_i) \equiv 1 \bmod q_i$. Hence, no precomputations are needed.

F.2 Trapdoor Sampling

The GaussSamp algorithm is the same as Algorithm 2 in [24] but the SampleG operation is called independently for each native-integer polynomial in the Double-CRT representation. The perturbation sampling is not affected by the use of CRT gadget vectors.

G Non-uniform Ring LWE

Extending the security proof [22, Thm 7.5] of private constrained PRFs to cyclotomic rings assumes the hardness of RLWE (or GLWE) with discrete Gaussian public samples, $a \in \mathcal{R}_q$ in the equation $a \cdot s + e$. We prove the security of this GLWE variant since its pseudorandomness is, at first glance, not obvious and it is the main step needed to extend [22] to the GLWE setting. The proof of the following theorem is adapted from [15], Section 4, with slightly better parameters. We remark that this is only needed for extending the security proof and our construction for general branching programs use a pseudorandom GLWE matrix $\mathbf{A}$ over $\mathcal{R}_q$ (with large entries).

The proof outline is straightforward: given $(\mathbf{A}, \mathbf{u}^t = \mathbf{s}^t\mathbf{A} + \mathbf{e}^t)$, simply view the GLWE sample as $\mathbf{s}^t\mathbf{A} + \mathbf{e}^t = \mathbf{s}^t\mathbf{G}\mathbf{G}^{-1}(\mathbf{A}) + \mathbf{e}^t$, and re-randomize the the secret $\mathbf{s}^t\mathbf{G}$ to uniformly random.

Theorem 3. *(Discrete Gaussian Matrix GLWE) There is a probabilistic polynomial time reduction from the generalized*
$(\mathcal{R}, d, m, q, \chi, \mathcal{U}(\mathcal{R}_q))$ *GLWE problem to the* $(\mathcal{R}, d', m, q, \chi, \mathcal{D}_{\mathbb{Z}^m, s})$ *GLWE problem for any* $d' \geq d \log_t q$, q, m *and* $s \geq \sqrt{t^2 + 1}\omega(\sqrt{\log(nd)})$ *for any* $t \geq 2$.

Proof. (of Theorem 3) We will simply map the uniformly random matrix $\mathbf{A} \in \mathcal{R}_q^{d \times m}$ to a discrete Gaussian $\mathbf{B} \in \mathcal{R}_q^{d' \times m}$, along with mapping a GLWE sample $\mathbf{u}$ with public matrix $\mathbf{A}$ to a GLWE sample $\mathbf{v}$ defined by $\mathbf{B}$. Further, uniform $\mathbf{u}$

will map to a new uniform vector under our mapping. The proof makes crucial use of discrete Gaussian G-lattice sampling algorithms, Lemma 1.

We can pad $\mathbf{G} = \mathbf{I}_d \otimes \mathbf{g}^T$ with columns of all 0s in ${\mathcal{R}_q}^d$ so Lemma 1 easily extends to an $d' \geq d \log_t q$.

Given an input $(\mathbf{A}, \mathbf{u}) \in {\mathcal{R}_q}^{d \times m} \times {\mathcal{R}_q}^m$, we perform the following steps:

1. For each column $\mathbf{a}_i \in {\mathcal{R}_q}^d$ of $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_m] \in {\mathcal{R}_q}^{d \times m}$, sample an independent discrete Gaussian $\mathbf{b}_i \leftarrow \mathbf{G}^{-1}(\mathbf{a}_i)$. Assemble these vectors into a matrix $\mathbf{B} = [\mathbf{b}_1, \cdots, \mathbf{b}_m] \in {\mathcal{R}_q}^{d' \times m}$. Notice $\mathbf{A} = \mathbf{GB}$.
2. Sample a uniformly random vector $\mathbf{r} \sim \mathcal{U}({\mathcal{R}_q}^{d'})$.
3. Return the tuple $(\mathbf{B}, \mathbf{v}^T = \mathbf{u}^T + \mathbf{r}^T\mathbf{B}) \in {\mathcal{R}_q}^{d' \times m} \times {\mathcal{R}_q}^m$.

Since we are sampling above the smoothing parameter of $\Lambda_q^{\perp}(\mathbf{G})$, a consequence of Claim 3.8 in [51] is the columns of $\mathbf{B}$ are i.i.d. vectors distributed as $\mathcal{D}_{\mathcal{R}^{d'}, s}$. Next, we see when $\mathbf{u}$ is uniformly random over ${\mathcal{R}_q}^m$ $\mathbf{v}^T$ is as well. On the other hand, we have $\mathbf{u}^T + \mathbf{r}^T\mathbf{B} = \mathbf{e}^T + \mathbf{s}^T\mathbf{A} + \mathbf{r}^T\mathbf{B} = \mathbf{e}^T + (\mathbf{s}^T\mathbf{G} + \mathbf{r}^T)\mathbf{B}$ when $\mathbf{u}$ is a $(\mathcal{R}, d, m, q, \chi)$LWE sample.

Remark 2. Since the base t can be chosen as a large parameter, the dimension-increase from RLWE to non-uniform GLWE can be small in-practice. Therefore, an increase in the dimension and the Gaussian width of the secrets in Section 5 leads to a TBO scheme for general branching programs *provably* secure from RLWE using the reductions in [22] along with Theorem 3.