Implementing Token-Based Obfuscation under (Ring) LWE

Cheng Chen*, Nicholas Genise†, Daniele Micciancio†, Yuriy Polyakov‡§, Kurt Rohloff‡

* MIT CSAIL
† UCSD
‡ NJIT Cybersecurity Research Center
§ Duality Technologies

DECEMBER 20, 2018

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 implement in PALISADE several optimized TBO constructions based on (Ring) LWE covering a relatively broad spectrum of capabilities, ranging from special-purpose linear functions to general branching programs. To the best of our knowledge, these are first implementations of TBO constructions based on lattices.

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. 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. We also present an optimized TBO implementation for conjunctions, 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.

Contents

I. 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, similar to how cryptography has provided data security based on the computational hardness assumptions. Some of these prior techniques are discussed in [2], [3], [4], [5], [6], [7]. Although often usable in practice, these approaches do not provide strong security guarantees, and can often be defeated without large computational effort. For example, [8], [9], [10], [11], [12] all provide methods to defeat heuristic software obfuscation.

There have been multiple recent attempts to develop cryptographically secure approaches to program obfuscation based on the computational hardness of mathematical problems. See [13] for a survey of these recent approaches. There are multiple definitions used for obfuscation in these recent approaches. Two prominent definitions are Virtual Black Box (VBB) and Indistinguishability Obfuscation (IO).

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 [14]. VBB is known to have strong limitations [15], [16], [17]. The most significant limitation is that *general-purpose* VBB obfuscation is unachievable [15].

To address limitations of VBB, Barak *et al.* [15] 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 [18], [19], [20], [21], [22], [23], [24]. There has also been recent work to implement multi-linear map constructions [25], [26], [27], [28], [1]. Recent results show that these constructions might not be secure [29], [30], [31], [32], [33], [34], [35], [36], [37]. The only IO constructions supporting general functions that are not subject to any attack to date are the works by Garg *et al.* [38] and Chen *et al.* [39]. 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 [40], [41], [42], [43], [44]. 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 reverse-engineered 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 [45], [46]. In the TBO model, the user obtains a special token before executing the obfuscated program and then finds the decrypted result by herself [47]. 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.

Our work presents first implementation results for the TBO of several types of programs, including linear functions (binary classifiers), conjunctions [48], permutation branching programs [48], and general branching programs [39]. All of the constructions presented in our work are secure under standard assumptions, namely Learning With Errors (LWE) or Ring LWE. To the best of our knowledge, these are first implementation results for token-based obfuscation based on lattices. The performance results for binary classifiers and conjunctions suggest that they are already practical.

A. *Our Contributions*

We present a new scheme for the TBO of 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.

We develop an efficient variant of the TBO of conjunctions based on the construction presented in [48], 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 (using lower values of main parameters and Central Limit Theorem/subgaussian analysis), 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 [1].

We present an efficient (full RNS) variant of the TBO for permutation branching programs based on the construction presented in [48], 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 [39], and implement it in PALISADE. The optimizations compared to the original construction include the use of Ring LWE instead of LWE, significantly improved key generation and evaluation algorithms for the token generator (both runtime and storage requirements are reduced by about two orders of magnitude), much tighter correctness constraints (using lower values of main parameters and Central Limit Theorem/subgaussian analysis), and a larger alphabet for encoding bits. Note that we use the same general framework for the TBO of both permutation and general branching programs.

The development of the ring variant of the TBO for general branching programs also required new algorithms for lattice trapdoor sampling and a security proof for the non-uniform Ring LWE problem. These contributions are also presented in our paper.

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.

B. *Organization*

The rest of the paper is organized as follows: Section II provides the preliminaries. Section III presents our TBO scheme for linear functions and describes how it can be used to obfuscate binary classifiers. Section IV discusses our variant of TBO for conjunctions. Section V presents a general treatment of TBO for branching programs. Section VI describes lattice trapdoor sampling algorithms (needed for TBO of branching programs) and RNS algorithms. The details of implementation and performance evaluation results are presented in Section VII. The paper concludes in Section VIII. Appendices discuss in more detail the algorithms for trapdoor sampling, derivation of correctness constraints for the TBO of conjunctions and branching programs, and the security proof for the non-uniform Ring LWE (needed for the TBO of general branching programs).

II. Preliminaries

A. *Cyclotomic Rings*

Our implementation utilizes cyclotomic polynomial rings $\mathcal{R} = \mathbb{Z}[x]/\langle x^n+1\rangle$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n+1\rangle$, where n is a power of 2 and q is an integer modulus. The order of cyclotomic polynomial $\Phi_{\hat{m}}(x) = x^n+1$ is $\hat{m} = 2n$. The modulus q is chosen to satisfy $q \equiv 1 \bmod \hat{m}$. The elements in these rings can be expressed in coefficient or evaluation representation. The coefficient representation of polynomial $a(x) = \sum_{i<n} a_i x^i$ treats the polynomial as a list of all coefficients $\mathbf{a} = \langle a_0, a_1, \ldots, a_{n-1}\rangle \in (\mathbb{Z}/q\mathbb{Z})^n$. The evaluation representation, also referred to as

polynomial Chinese Remainder Transform (CRT) representation [49], computes the values of polynomial $a(x)$ at all primitive $\hat{m}$-th roots of unity modulo q, i.e., $b_i = a(\zeta^i) \bmod q$ for $i \in (\mathbb{Z}/\hat{m}\mathbb{Z})^*$. These cyclotomic rings support fast polynomial multiplication by transforming the polynomials from coefficient to evaluation representation in $O(n \log n)$ time [50] (also called negacyclic convolution NTT) and component-wise multiplication.

Lattice sampling works with n-dimensional discrete Gaussian distributions over lattice $\Lambda \subset \mathbb{R}^n$ 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 lattice 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 $\mathbb{Z}_q$.

B. Cyclotomic Fields

The perturbation generation procedure in trapdoor sampling also utilizes cyclotomic fields $\mathcal{K}_{2n} = \mathbb{Q}[x]/\langle x^n + 1 \rangle$, which are similar in their properties to the cyclotomic rings except that the coefficients/values of the polynomials are rationals rather than integers. The elements of the cyclotomic fields also have coefficient and evaluation (CRT) representation, and support fast polynomial multiplication using variants of the Fast Fourier Transform (FFT). The evaluation representation of such rational polynomials in our implementation works with complex primitive roots of unity.

C. 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 $q_i \equiv 1 \bmod \hat{m}$. 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 [51], we refer to this representation of polynomials as "Double-CRT".

D. Non-uniform Generalized Learning with Errors

The following distinguishing problem, originated by Regev and modified to an algebraic version [52], will be our source of cryptographic hardness.

Definition 1. *(Generalized, 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 χ, D be distributions over $\mathcal{R}_q$. Then, the non-uniform $(\mathcal{R}^l, m, q, \chi, D)$RLWE problem is to distinguish between the following two distributions:*

$$\{(\mathbf{A}, \mathbf{s}^T\mathbf{A} + \mathbf{e}^T)\} \text{ and } \{(\mathbf{A}, \mathbf{u}^T)\},$$

where, $\mathbf{s} \leftarrow \mathcal{U}({\mathcal{R}_q}^l)$, $\mathbf{A} \leftarrow D^{l \times m}$, $e \leftarrow \chi^m$ and $\mathbf{u} \leftarrow \mathcal{U}({\mathcal{R}_q}^m)$[1].

Traditionally, D is the uniform distribution and the noise, χ, is a discrete Gaussian (defined below). We will, however, prove the hardness of RLWE when D is a discrete Gaussian. This is required for extending the security reduction of [39] to RLWE.

[1]This problem is refered to as GLWE or MLWE in literature [53], [54], though we refer to it as RLWE for succintness.

E. Discrete Gaussians and G-Lattice Sampling

Let $\rho_\sigma(\mathbf{x}) = e^{-\pi\|\mathbf{x}\|^2/\sigma^2}$ be the Gaussian function. Then for any discrete subset of euclidean space, $S \subset \mathbb{R}^n$, the discrete Gaussian distribution over S of width $\sigma > 0$ has probability mass function $\mathcal{D}_{S,\sigma}(\mathbf{x}) = \rho_\sigma(\mathbf{x})/(\sum_{\mathbf{y}\in S}\rho_\sigma(\mathbf{y})) = \rho_\sigma(\mathbf{x})/\rho_\sigma(S)$. 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. [55].

Definition 2. *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 $\Lambda_q^\perp(\mathbf{G}) = \{\mathbf{x} \in \mathcal{R}^{l\kappa} : \mathbf{G}\mathbf{x} = \mathbf{0} \in {\mathcal{R}_q}^l\}$. 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 3. *([56], [57]) For any $\sigma > (t+1)\omega(\sqrt{\log nl})$, there is a probabilistic $O(\kappa)$-time algorithm whose output is distributed statistically close to $\mathcal{D}_{\Lambda_\mathbf{u}^\perp(\mathbf{G}),\sigma}$.*

F. GGH15 Encoding

We will use the generalized GGH15 construction [21] given in [39], called γ-GGH15. For a complete description, see Section 2 of [39].

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 set ring elements. Let $\gamma(\mathbf{M}, s)$ be a function mapping $(\mathbf{M}, s)$ to another matrix satisfying $\gamma(\mathbf{M}, s)\gamma(\mathbf{M}', s') = \gamma(\mathbf{M}\mathbf{M}', ss')$. 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 $\mathbf{x} \in \{0,1\}^\ell$.

G. Token-Based Obfuscation

Definition 4. *A token-based obfuscation [58] 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 pair of probabilistic polynomial time algorithms* (tOB.Obfuscate, tOB.TokenGen) *with the following properties.*

- tOB.Obfuscate$(1^\kappa, C \in \mathcal{C}_n)$ *takes as input a security parameter κ and a circuit C, and outputs a secret key* osk *for token generation as well as an obfuscated circuit, O.*
- tOB.TokenGen(osk, x) *takes as input a bit string $x \in \{0,1\}^n$ and the secret key* osk *and returns a token* tk$_x$.

We require that $O(\text{osk}, tk_x) = C(x)$ with all but negligible probability and that the running time of tOB.TokenGen *is independent of the size of C.*

For succinctness, we sketch the security definition for TBO (Section 5 of [58]). An efficient adversary $A = (A_1, A_2)$ must decide which of the following two experiments it is in. The first (real experiment) is where A_1 outputs a circuit C and a state s_A to pass on to A_2, then A_2 gets the obfuscated circuit as well as a number of token queries tk_x of its choice (the number and form of the queries can be restricted in advance). The second (ideal) experiment is the same except we replace tOB.Obfuscate$(1^\kappa, C \in \mathcal{C}_n)$ with an efficient simulator S_1 and the token queries are replaced with another efficient simulator S_2 that only receives the output of the circuit, $C(x)$.

III. 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 show that this technique can efficiently support the obfuscation of binary classifiers for 24-bit features in modern commodity computing environments. 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.

A. 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:

- 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 σ.
- KeyGen$(N) \to$ 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.
- Obfuscate(SK, $\mathbf{w}$) $\to \mathbf{C}$. 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
$$\mathbf{C} := \Big[\mathbf{a}, \mathbf{c} := \{\langle \mathbf{a}, \mathbf{s_i} \rangle + pe_i + w_i\}_{i=1}^{N}\Big].$$
- TokenGen(SK, $\mathbf{x}$) $\to \mathbf{t}$. 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.
- Eval$(\mathbf{C}, \mathbf{t}, \mathbf{x}) \to \bar{\mu}$. Compute
$$\bar{\mu} := \sum_{i=1}^{N} c_i x_i - \langle \mathbf{a}, \mathbf{t} \rangle \mod p.$$

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. $\bmod q$. Indeed,

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

where the first term in the result is eliminated after applying $\bmod p$. 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 $\mathbf{C}$ is secure under LWE. The main 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 weigts vector $\mathbf{w}$. As our main motivating application is binary classifiers, we defer a more detailed discussion of security to the next section.

B. Token-Based Obfuscation of Binary Classifiers

The TBO of linear functions can be used to build obfuscated binary classifiers. Consider two possible scenarios: (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.

1) 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 $111\dots111$ 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.

2) 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.

3) Security: Our security definition is the same as Definition 5.2 in [58]. First, we note that the simulator S_1 given in the game is simply a program which obfuscates the $\mathbf{w} = \mathbf{0} \in \mathbb{Z}_q^N$ vector. This is indistinguishable from the real obfuscated program due to LWE's pseudorandomness[2]. Next, we replace the real token queries ($\mathbf{t}$) with $\mathbf{t}' = \mathbf{S}\mathbf{x} - \mathbf{v}$ where $\langle \mathbf{a}, \mathbf{v} \rangle = \langle \mathbf{w}, \mathbf{x} \rangle \mod p$ (say $\mathbf{v} = (a[1]^{-1} \cdot \langle \mathbf{w}, \mathbf{x} \rangle, 0, \cdots, 0) \mod p$). This ensures the evaluations are the same in both the real experiment and the ideal experiment.

In the real setting, information is leaked through the adversary's token queries, $\mathbf{t}_\mathbf{x}$. Let $\mathbf{S} \in \mathbb{Z}_q^{n \times N}$ be the concatenated $\mathbf{s}_i \in \mathbb{Z}_q^n$ forming SK. These queries reveal a system of equations to the adversary: $\mathbf{T} = \mathbf{S}\mathbf{X} \mod q$

[2]There is a routine pseudorandomness reduction from LWE to LWE of the form $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + p \cdot e)$. The reduction is to map the input $(\mathbf{a}, b)$ to $(\mathbf{a}, pb + \langle \mathbf{a}, \mathbf{s}' \rangle)$ for a random $\mathbf{s}'$.

and $\mathbf{v} = \mathbf{w}^t\mathbf{X} \mod p$ for $\mathbf{X} \in \mathbb{Z}_q^{N\times y}$ where y is the number of token queries and each column of $\mathbf{T}$ is a token query. Therefore, we limit the number of tokens administered to less than N to ensure collusion-resistance. Most applications will limit the number of token queries to much less than $N-1$ depending on the number of classifiers used for the application (the size of the adversary's search space).

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.

4) Space and computational complexity: 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 compexity 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.

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.

IV. 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 constrained-hiding constrained PseudoRandom Functions (PRF) proposed in Section 5.1 of [48]. We chose the example of conjunctions to give a fair comparison with a prior recent non-interactive (distributional VBB) conjunction obfuscation construction implemented in [1] and introduce several major optimizations that are used in the next section for the TBO of more general programs, i.e., permutation and general branching programs.

Compared to the non-interactive conjunction obfuscation construction implemented in [1] (and originally formulated in [59]), the TBO construction has several advantages w.r.t. both security and efficiency. The construction [1], [59] 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 [1] (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).

A. 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 [60], [61]. Conjunctions can be used to encode binary classifiers (similar to the approach discussed in Section III-B) 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 [1].

B. Conceptual Model

The scheme for the TBO of conjunctions includes the same tuple of functions as the TBO for linear functions (Section III-A) 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 $\mathbf{y}' \in R_p^{1\times m}$, 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 $\mathbf{y} \in R_p^{1\times m}$, where $\mathcal{R}_p = \mathbb{Z}_p[x]/\left\langle x^n+1\right\rangle$. 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.

C. 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 [1], 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 [48].

The main difference of the TBO model as compared to the distributional VBB model [1] 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 [48] 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.

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 D) and computing the unconstrained key as $\mathcal{L} \times b$ short ring elements.

As compared to the construction in [48], 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

Algorithm 1 Key generation

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

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
                $r_{i,b} := s_{i,b}$
            end if
            $\mathbf{D}_{i,b} := \mathsf{Encode}_{\mathbf{A}_{i-1}\to\mathbf{A}_i}(\widetilde{\mathbf{T}}_{i-1}, r_{i,b}, \sigma)$
        end for
    end for
    $\Pi_v := \left(\mathbf{A}_0, \{\mathbf{D}_{i,b}\}_{i\in[\mathcal{L}],b\in\{0,...,2^w-1\}}\right)$
    return $\Pi_v$
end function
```

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 [48]. 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 [1]. The lattice trapdoor sampling procedure GAUSSSAMP is described in Appendix D.

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

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 [48]: it multiplies short ring elements followed

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 $\mathbf{R}_{i+1}$
end function
```

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 $\mathbf{y}'$
end function
```

by a single scalar product with the second ring element of the public key $\mathbf{A}_0$ (in contrast to vector-matrix products in [48]), 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($\mathbf{x} \in \{0,1\}^L$, $\Pi_v$, $\mathbf{y}'$)
    $\mathbf{D}_{\mathbf{\Pi}} := \mathbf{A}_0$
    for i = 1..$\mathcal{L}$ do
        $\mathbf{D}_{\mathbf{\Pi}} := \mathbf{D}_{\mathbf{\Pi}}\, \mathbf{D}_{i,x[1+(i-1)w\,:\,iw]} \in {\mathcal{R}_q}^{1\times m}$
    end for
    $\mathbf{y} := \lfloor \frac{2}{q}\mathbf{D}_{\mathbf{\Pi}}[1] \rceil \in \mathbb{Z}_2^n$
    return $(\mathbf{y} = \mathbf{y}')$
end function
```

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 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) [3]. 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 A).

D. 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 [63]. 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.

[3]This analysis is nearly identical to the original LWE to LWR reduction in [62] for $p = 2$.

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 [63].

Lemma 2.9 of [63] states that $s_1(\mathbf{X}) \le C_0\sigma\left(\sqrt{n\kappa} + \sqrt{2n} + C_1\right)$, 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 ($\mathcal{L} \ge 2$) is expressed as

$$q > 2^{10} P_e^{-1} B_e \left(\beta\sigma_s\sqrt{mn}\right)^{\mathcal{L}-1}, \tag{1}$$

where $B_e = 6\sigma, \beta = 6$, $P_e = 2^{-20}$, and all other parameters are the same as in [1]. The derivation details are presented in Appendix A.

Ring Dimension n. All of the security proofs presented in [48] 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[4] (commit a2296b8) [64] to find the lowest security levels for the uSVP, decoding, and dual attacks following the standard homomorphic encryption security recommendations [65]. 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 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 [1].

Word size w. We found $w = 8$ to be the optimal value for all our experiments, using the same procedure as described in [1].

E. Comparison with Construction in [1]

As the building blocks and many underlying parameters for the TBO construction are the same as for the distributional VBB constructon [1], 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 [1] 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 is approximately 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 [1].

V. 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 [48]. For general branching programs, we adapt the private constrained PRF[5] construction of [39] (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 IV 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

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

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

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.

Compared to the constructions in [39] and [48], 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. In addition to the above, we also present a ring variant of the general branching program construction (versus the matrix one in [39]), providing a proof for the non-uniform Ring LWE problem in Appendix E.

A. Matrix Branching Programs

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

Definition 5. *(Matrix branching programs [39]) 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 6. *(Permutation branching programs: Type II branching programs in [39])*

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 6.

Definition 7. *(General branching programs: Type I branching programs in [39])*
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 [39] for two specific representations of CNFs).

The relationships between these two types of branching programs are discussed in [39].

B. 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.

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 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 [39], 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 VI-A and Appendix C.

Algorithm 6 Key generation for branching programs

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

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 [39] and [48].

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 [39].

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.

C. *Parameter Selection for Matrix Branching Programs*

The correctness constraint for branching programs with $\mathcal{L}$ words ($\mathcal{L} \geq 2$) is expressed as

$$q > 2^{10} P_e^{-1} B_J B_e \left(6\sigma_s\sqrt{dmn}\right)^{\mathcal{L}-1}, \tag{2}$$

Algorithm 7 Obfuscation for branching programs

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

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 $\mathbf{R}_{i+1}$
end function
```

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 B.

D. *Efficiency of Permutation and General Branching Programs*

The general branching program represention is typically significantly more efficient than the permutation representation [39]. 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 [39], 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.

Algorithm 9 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 $\mathbf{y}'$
end function
```

Algorithm 10 Evaluation using the obfuscated program

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

E. 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 $T + 1$.

VI. Efficient Algorithms for the Ring Constructions

A. Trapdoor Sampling for the Matrices of Ring Elements

Here we describe the trapdoor generation and sampling procedures, TrapGen and GaussSamp, respectively. For branching programs, we needed a new algorithm, SampleMat, which may be of independent interest. This algorithm samples a discrete Gaussian perturbation with a covariance described as a general matrix over the ring $\mathcal{R}$.

The pseudocode for trapdoor generation and sampling is given in Appendix C. In short, TrapGen takes as input a security parameter and outputs a (pseudo)random 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 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.

Discrete Gaussian sampling, GaussSamp, is broken into two subroutines. The first is a perturbation sampling procedure, SamplePert, which outputs a perturbation to statistically hide the trapdoor, independent of the input $\mathbf{u}$. Second, the procedure samples a discrete Gaussian over a G-lattice represented by some gadget matrix. The output of GaussSamp is simply the sum of the perturbation and the gadget lattice sample (under a linear transformation dependent on the trapdoor).

We implemented the trapdoor generation algorithm, TrapGen used in Algorithm 1 and Algorithm 6, from [56] in the computational instantiation (meaning pseudorandomness comes from RLWE).

In the case of branching programs, an extra layer is needed in the perturbation portion of our implementation in order to account for the extra ring dimensions d. This extra layer is given in the Appendix C as Algorithm 14, SampleMat. This algorithm follows the same Schur-complement decomposition used in [57], but is applied to general matrices over the ring. It breaks the covariance into four blocks, $\Sigma = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{D} \end{bmatrix}$, and performs a discrete Gaussian convolution with samples over $\mathcal{R}$ with covariances $\mathbf{D}$ and the Schur-complement $\mathbf{A} - \mathbf{BD}^{-1}\mathbf{B}^T$. Notice that we can break the covariance matrix Σ up into any blocks we choose, though we implemented halving the dimension each time since it is asymptotically the most efficient approach.

B. 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 (64-bit for x86-64 architectures). There are many benefits of using the Double-CRT representation, and new algorithms have recently been proposed [66], [67], [68], [69]. 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

TABLE I: 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

(multi-precision) format to extract the digits. Instead, we use a CRT representation of the gadget matrix that was recently proposed in [69], 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 [1], in Appendix D.

For the scale-and-round operation, we utilize the RNS scaling procedure proposed in [67] 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.

VII. Implementation and Results

A. Software Implementation

We implemented the TBO constructions in PALISADE v1.3.1 [70], 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.
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.

B. 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.

C. Token-Based Obfuscation of Linear Functions (Binary Classifiers)

Table I shows both single- and multi-threaded experimental 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

TABLE II: 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 [1]									
32	14	4096	180	15	36.8	12.4	–	–	53.0

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.

D. Token-Based Obfuscation of Conjunctions

Table II 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 [1], [59] 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 [1] 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 IV.

We also examined the effect of OpenMP loop parallelization optimizations by comparing the results for single- and multi-threaded scenarios (Table II). Here, we chose 14 (which matches the number of cores per socket) as the number of threads because 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.

E. Token-Based Obfuscation of Branching Programs

Table III 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.

TABLE III: 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

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.

Although our TBO construction for branching programs includes many major optimizations, our efficiency results for general branching programs are still far from being practical. However, the functionality supported by our construction is more advanced than for all prior implementations of non-trivial program obfuscation [26], [27], [1], [71] under the VBB or IO models.

VIII. Conclusion

We have presented the 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 (branching program) constructions are still far from being practical. The important benefit of the TBO model is the ability to support the obfuscation of certain programs, such as some binary classifiers, that are not secure 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.

The obfuscation of general branching programs is still not practical under both TBO and IO. Note that the IO candidate based on the non-uniform LWE presented in [39], which is one of very few unbroken IO candidates, would further degrade the performance, as compared to the TBO of general branching programs, due to the addition of "bundling" matrices and other extra objects. It appears that a fundamental breakthrough comparable to Gentry's fully homomorphic encryption idea in 2009 is needed to make the general program obfuscation efficient and secure. To make the TBO of general branching programs practical, either the GGH15 or multilinear approach has to be replaced with a totally different approach, or a fundamentally new compact trapdoor construction has to be proposed.

IX. 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] D. B. Cousins, G. D. Crescenzo, K. D. Gür, K. King, Y. Polyakov, K. Rohloff, G. W. Ryan, and E. Savaş, "Implementing conjunction obfuscation under entropic ring lwe," in *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 68–85, cryptology ePrint Archive, Report 2017/844, https://eprint.iacr.org/2017/844. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2018.00007

[2] D. Low, "Protecting java code via code obfuscation," *Crossroads*, vol. 4, no. 3, pp. 21–23, Apr. 1998.

[3] G. Wroblewski, "General method of program code obfuscation," Ph.D. dissertation, Citeseer, 2002.

[4] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, 2003, pp. 290–299.

[5] S. Schrittwieser, S. Katzenbeisser, P. Kieseberg, M. Huber, M. Leithner, M. Mulazzani, and E. Weippl, "Covert computation: Hiding code in code for obfuscation purposes," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13, 2013, pp. 529–534.

[6] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *NDSS*, 2008.
[7] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *Proceedings of the 8th International Conference on Information Security Applications*, ser. WISA'07, 2007, pp. 61–75.
[8] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proffing, and obfuscation: Tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735–746, Aug. 2002.
[9] N. Eyrolles, L. Goubin, and M. Videau, "Defeating mba-based obfuscation," in *Proceedings of the 2016 ACM Workshop on Software PROtection*, ser. SPRO '16, 2016, pp. 27–38.
[10] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX Security Symposium*, 2004.
[11] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static analyzer of vicious executables (save)," in *20th Annual Computer Security Applications Conference*, Dec 2004, pp. 326–334.
[12] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: reverse engineering obfuscated code," in *12th Working Conference on Reverse Engineering (WCRE'05)*, Nov 2005, p. 10.
[13] B. Barak, "Hopes, fears, and software obfuscation," *Commun. ACM*, vol. 59, no. 3, pp. 88–96, Feb. 2016.
[14] S. Hada, *Zero-Knowledge and Code Obfuscation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 443–457.
[15] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *J. ACM*, vol. 59, no. 2, pp. 6:1–6:48, May 2012.
[16] N. Bitansky, R. Canetti, H. Cohn, S. Goldwasser, Y. T. Kalai, O. Paneth, and A. Rosen, *The Impossibility of Obfuscation with Auxiliary Input or a Universal Simulator*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 71–89.
[17] S. Goldwasser and Y. T. Kalai, "On the impossibility of obfuscation with auxiliary input," in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, Oct 2005, pp. 553–562.
[18] B. Applebaum and Z. Brakerski, *Obfuscating Circuits via Composite-Order Graded Encoding*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 528–556.
[19] B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai, *Protecting Obfuscation against Algebraic Attacks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 221–238.
[20] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," *SIAM Journal on Computing*, vol. 45, no. 3, pp. 882–929, 2016.
[21] C. Gentry, S. Gorbunov, and S. Halevi, *Graph-Induced Multilinear Maps from Lattices*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 498–527.
[22] H. Lin, *Indistinguishability Obfuscation from SXDH on 5-Linear Maps and Locality-5 PRGs*. Cham: Springer International Publishing, 2017, pp. 599–629.
[23] H. Lin, R. Pass, K. Seth, and S. Telang, *Indistinguishability Obfuscation with Non-trivial Efficiency*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 447–462.
[24] H. Lin and S. Tessaro, *Indistinguishability Obfuscation from Trilinear Maps and Block-Wise Local PRGs*. Cham: Springer International Publishing, 2017, pp. 630–660.
[25] D. Apon, Y. Huang, J. Katz, and A. J. Malozemoff, "Implementing cryptographic program obfuscation," Cryptology ePrint Archive, Report 2014/779, 2014, http://eprint.iacr.org/2014/779.
[26] K. Lewi, A. J. Malozemoff, D. Apon, B. Carmer, A. Foltzer, D. Wagner, D. W. Archer, D. Boneh, J. Katz, and M. Raykova, "5gen: A framework for prototyping applications using multilinear maps and matrix branching programs," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016, pp. 981–992.
[27] S. Halevi, T. Halevi, V. Shoup, and N. Stephens-Davidowitz, "Implementing bp-obfuscation using graph-induced encoding," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 783–798. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133976
[28] B. Carmer, A. J. Malozemoff, and M. Raykova, "5gen-c: Multi-input functional encryption and program obfuscation for arithmetic circuits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 747–764. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133983
[29] J.-S. Coron, C. Gentry, S. Halevi, T. Lepoint, H. K. Maji, E. Miles, M. Raykova, A. Sahai, and M. Tibouchi, *Zeroizing Without Low-Level Zeroes: New MMAP Attacks and their Limitations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 247–266.
[30] J.-S. Coron, M. S. Lee, T. Lepoint, and M. Tibouchi, *Cryptanalysis of GGH15 Multilinear Maps*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 607–628.
[31] J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehlé, *Cryptanalysis of the Multilinear Map over the Integers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 3–12.
[32] J. H. Cheon, P.-A. Fouque, C. Lee, B. Minaud, and H. Ryu, *Cryptanalysis of the New CLT Multilinear Map over the Integers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 509–536.
[33] Y. Hu and H. Jia, *Cryptanalysis of GGH Map*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 537–565.
[34] E. Miles, A. Sahai, and M. Zhandry, *Annihilation Attacks for Multilinear Maps: Cryptanalysis of Indistinguishability Obfuscation over GGH13*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 629–658. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-53008-5_22
[35] J.-S. Coron, M. S. Lee, T. Lepoint, and M. Tibouchi, *Zeroizing Attacks on Indistinguishability Obfuscation over CLT13*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 41–58.
[36] Y. Chen, C. Gentry, and S. Halevi, *Cryptanalyses of Candidate Branching Program Obfuscators*. Cham: Springer International Publishing, 2017, pp. 278–307. [Online]. Available: https://doi.org/10.1007/978-3-319-56617-7_10
[37] D. Apon, N. Döttling, S. Garg, and P. Mukherjee, "Cryptanalysis of Indistinguishability Obfuscations of Circuits over GGH13," in *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, vol. 80, 2017, pp. 38:1–38:16.
[38] S. Garg, E. Miles, P. Mukherjee, A. Sahai, A. Srinivasan, and M. Zhandry, *Secure Obfuscation in a Weak Multilinear Map Model*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 241–268. [Online]. Available: https://doi.org/10.1007/978-3-662-53644-5_10

[39] Y. Chen, V. Vaikuntanathan, and H. Wee, "Ggh15 beyond permutation branching programs: Proofs, attacks, and candidates," in *Advances in Cryptology – CRYPTO 2018*, H. Shacham and A. Boldyreva, Eds. Cham: Springer International Publishing, 2018, pp. 577–607.

[40] M. Bellare and I. Stepanovs, *Point-Function Obfuscation: A Framework and Generic Constructions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 565–594. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49099-0_21

[41] S. Goldwasser and G. N. Rothblum, *On Best-Possible Obfuscation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 194–213. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70936-7_11

[42] B. Lynn, M. Prabhakaran, and A. Sahai, *Positive Results and Techniques for Obfuscation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 20–39. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24676-3_2

[43] G. D. Crescenzo, L. Bahler, B. A. Coan, Y. Polyakov, K. Rohloff, and D. B. Cousins, "Practical implementations of program obfuscators for point functions," in *International Conference on High Performance Computing & Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016*. IEEE, 2016, pp. 460–467. [Online]. Available: http://dx.doi.org/10.1109/HPCSim.2016.7568371

[44] L. Bahler, G. Di Crescenzo, Y. Polyakov, K. Rohloff, and D. B. Cousins, "Practical implementation of lattice-based program obfuscators for point functions," in *2017 International Conference on High Performance Computing & Simulation, HPCS 2017, Genoa, Italy, July 17-21, 2017*, 2017, pp. 761–768. [Online]. Available: https://doi.org/10.1109/HPCS.2017.115

[45] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia, "Founding cryptography on tamper-proof hardware tokens," in *Theory of Cryptography*, D. Micciancio, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 308–326.

[46] N. Bitansky, R. Canetti, S. Goldwasser, S. Halevi, Y. T. Kalai, and G. N. Rothblum, "Program obfuscation with leaky hardware," in *Advances in Cryptology – ASIACRYPT 2011*, D. H. Lee and X. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 722–739.

[47] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable garbled circuits and succinct functional encryption," in *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '13. New York, NY, USA: ACM, 2013, pp. 555–564. [Online]. Available: http://doi.acm.org/10.1145/2488608.2488678

[48] R. Canetti and Y. Chen, "Constraint-hiding constrained prfs for nc1 from lwe," Cryptology ePrint Archive, Report 2017/143, 2017, https://eprint.iacr.org/2017/143.

[49] V. Lyubashevsky, C. Peikert, and O. Regev, "A toolkit for ring-LWE cryptography," in *EUROCRYPT*, vol. 7881. Springer, 2013, pp. 35–54.

[50] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, "SWIFFT: A modest proposal for FFT hashing," in *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, 2008, pp. 54–72. [Online]. Available: https://doi.org/10.1007/978-3-540-71039-4_4

[51] C. Gentry, S. Halevi, and N. P. Smart, *Homomorphic Evaluation of the AES Circuit*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 850–867. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_49

[52] V. Lyubashevsky, C. Peikert, and O. Regev, *On Ideal Lattices and Learning with Errors over Rings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13190-5_1

[53] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," *Des. Codes Cryptography*, vol. 75, no. 3, pp. 565–599, 2015. [Online]. Available: https://doi.org/10.1007/s10623-014-9938-4

[54] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, S. Goldwasser, Ed. ACM, 2012, pp. 309–325. [Online]. Available: http://doi.acm.org/10.1145/2090236.2090262

[55] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, ser. STOC '08. New York, NY, USA: ACM, 2008, pp. 197–206.

[56] D. Micciancio and C. Peikert, "Trapdoors for lattices: Simpler, tighter, faster, smaller," in *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, ser. Lecture Notes in Computer Science, D. Pointcheval and T. Johansson, Eds., vol. 7237. Springer, 2012, pp. 700–718. [Online]. Available: https://doi.org/10.1007/978-3-642-29011-4_41

[57] N. Genise and D. Micciancio, "Faster gaussian sampling for trapdoor lattices with arbitrary modulus," in *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. B. Nielsen and V. Rijmen, Eds., vol. 10820. Springer, 2018, pp. 174–203. [Online]. Available: https://doi.org/10.1007/978-3-319-78381-9_7

[58] S. Goldwasser, Y. T. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable garbled circuits and succinct functional encryption," in *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, D. Boneh, T. Roughgarden, and J. Feigenbaum, Eds. ACM, 2013, pp. 555–564. [Online]. Available: https://doi.org/10.1145/2488608.2488678

[59] Z. Brakerski, V. Vaikuntanathan, H. Wee, and D. Wichs, "Obfuscating conjunctions under entropic ring lwe," in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, ser. ITCS '16, 2016, pp. 147–156.

[60] M. Kubat, *An Introduction to Machine Learning*, 1st ed. Springer Publishing Company, Incorporated, 2015.

[61] Y. Xiao, K. G. Mehrotra, and C. K. Mohan, *Efficient Classification of Binary Data Stream with Concept Drifting Using Conjunction Rule Based Boolean Classifier*, 2015, pp. 457–467.

[62] A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," in *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, ser. Lecture Notes in Computer Science, D. Pointcheval and T. Johansson, Eds., vol. 7237. Springer, 2012, pp. 719–737. [Online]. Available: https://doi.org/10.1007/978-3-642-29011-4_42

[63] D. Micciancio and C. Peikert, "Trapdoors for lattices: Simpler, tighter, faster, smaller," in *Advances in Cryptology–EUROCRYPT 2012*. Springer, 2012, pp. 700–718.

[64] M. Albrecht, S. Scott, and R. Player, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, p. 169–203, 10 2015.

[65] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Security of homomorphic encryption," HomomorphicEncryption.org, Redmond WA, Tech. Rep., July 2017.

[66] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, “A full rns variant of fv like somewhat homomorphic encryption schemes,” in *Selected Areas in Cryptography – SAC 2016*, R. Avanzi and H. Heys, Eds. Cham: Springer International Publishing, 2017, pp. 423–442.

[67] S. Halevi, Y. Polyakov, and V. Shoup, “An improved rns variant of the bfv homomorphic encryption scheme,” Cryptology ePrint Archive, Report 2018/117, 2018, https://eprint.iacr.org/2018/117.

[68] A. A. Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, “Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme,” Cryptology ePrint Archive, Report 2018/589, 2018, https://eprint.iacr.org/2018/589.

[69] N. Genise, D. Micciancio, and Y. Polyakov, “Building an efficient lattice gadget toolkit: Subgaussian sampling and more,” Cryptology ePrint Archive, Report 2018/946, 2018, https://eprint.iacr.org/2018/946.

[70] Y. Polyakov, K. Rohloff, and G. W. Ryan, “PALISADE lattice cryptography library,” https://git.njit.edu/palisade/PALISADE, Accessed November 2018.

[71] D. Apon, Y. Huang, J. Katz, and A. J. Malozemoff, “Implementing cryptographic program obfuscation.”

[72] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan, “Key homomorphic prfs and their applications,” in *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Canetti and J. A. Garay, Eds., vol. 8042. Springer, 2013, pp. 410–428. [Online]. Available: https://doi.org/10.1007/978-3-642-40041-4_23

[73] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” *J. ACM*, vol. 56, no. 6, pp. 34:1–34:40, 2009. [Online]. Available: http://doi.acm.org/10.1145/1568318.1568324

[74] D. Pointcheval and T. Johansson, Eds., *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7237. Springer, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-29011-4

Appendix A
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}.

Appendix B
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:

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

Using the fact that $\|\mathbf{D}_{k,x_k}\|_\infty \gg \left\|\gamma(\widehat{\mathbf{M}}_{i,x_i}, s_{i,x_i})\right\|_\infty$ 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}.

Appendix C
Trapdoor Algorithms for Matrices of Ring Elements

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 [57]).

Our trapdoor construction is identical to the original MP12 construction [56] for RLWE. Specifically, we are sampling the "computational instantiation" described in Section 5 of [56].

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

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 $\mathbf{p} + \begin{bmatrix}\mathbf{T}\\ \mathbf{I}\end{bmatrix}\mathbf{z}$.
end function
```

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 [69].

1) 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 [57] 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.

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 [56]. It calls a subroutine, Algorithm 14 or SAMPLEMAT, which given a positive definite matrix $\Sigma_d \in \mathcal{F}^{d\times d}$, 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 [57]. 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 [57]. The matrices $\mathbf{A}, \mathbf{D}, \mathbf{B} \in \mathcal{F}^{d/2\times d/2}$ for an even dimension d. For an odd dimension, $\mathbf{A} \in \mathcal{F}^{\lceil d/2\rceil\times\lceil d/2\rceil}$, $\mathbf{D} \in \mathcal{F}^{\lfloor d/2\rfloor\times\lfloor d/2\rfloor}$, and $\mathbf{B} \in \mathcal{F}^{\lceil d/2\rceil\times\lfloor d/2\rfloor}$. The last algorithm called is a one-dimensional discrete gaussian sampler, SAMPLEZ.

a) 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 $\sigma_t \geq (t+1)C_{\varepsilon,d\kappa}$ [57]. 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
        $q_i$ ←SampleZ($s^2 - \eta^2$)
    end for
    $\mathbf{c} := \frac{-\eta^2}{s^2-\eta^2}\mathbf{T}\mathbf{q}$
    $\mathbf{p}$ ←SampleMat($\Sigma_d, \mathbf{c}$)
    return $(\mathbf{p}, \mathbf{q})$
end function
```

Algorithm 14 Perturbation Sampling, Matrix

```
function SAMPLEMAT($\Sigma, \mathbf{c}$)
    if $d = 1$ then
        return SampleF($\Sigma, \mathbf{c}$)
    end if
    $\mathbf{c} = (\mathbf{c}_0, \mathbf{c}_1) \in \mathcal{F}^d$
    $\mathbf{q}_1$ ← SampleMat($\mathbf{D}, \mathbf{c}_1$).
    $\Sigma' := \mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{B}^T$.
    $\mathbf{q}_0$ ←SampleMat($\Sigma', \mathbf{c}_0 + \mathbf{B}\mathbf{D}^{-1}(\mathbf{q}_1 - \mathbf{c}_1)$).
    return $(\mathbf{q}_0, \mathbf{q}_1)$
end function
```

Algorithm 15 Perturbation Sampling, Field Element

```
function SampleF($f, c$)
    if $n = 1$ then
        return SampleZ($f, c$)
    end if
    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 $(q_0, q_1)$
end function
```

Appendix D
Trapdoor Algorithms in CRT

A. Trapdoor Generation

The TrapGen procedure is the same as described in Algorithm 1 of [1] 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 $\mathbf{g}_{CRT}^T = (q_1^* \hat{q}_1 \cdot \mathbf{g}_1^T, \cdots, q_l^* \hat{q}_l \cdot \mathbf{g}_l^T) \mod q \in \mathbb{Z}_q^{1 \times \kappa}$ [69].

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.

B. Trapdoor Sampling

The GaussSamp algorithm is the same as Algorithm 2 in [1] 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.

Appendix E
Non-uniform Ring LWE

The security proof [Thm 7.5, CVW18] of private constrained PRFs assumes the hardness of RLWE with discrete gaussian public samples, $a \in \mathcal{R}_q$ in the equation $a \cdot s + e$. In this section, we prove the security of a GLWE variant. The proof of the following theorem is adapted from [72], Section 4, with slightly better parameters.

Theorem 8. *(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 8) 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 3.

We can pad $\mathbf{G} = \mathbf{I}_d \otimes \mathbf{g}^T$ with columns of all 0s in $\mathcal{R}_q{}^d$ so Lemma 3 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 [73] 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. $\square$