Implementing Token-Based Obfuscation under (Ring) LWE

Cheng Chen
MIT CSAIL
USA
chengch@mit.edu

Nicholas Genise
SRI International
USA
nicholas.genise@sri.com

Daniele Micciancio
University of California, San Diego
USA
daniele@cs.ucsd.edu

Yuriy Polyakov
New Jersey Institute of Technology
and Duality Technologies
USA
polyakov@njit.edu

Kurt Rohloff
New Jersey Institute of Technology
and Duality Technologies
USA
rohloff@njit.edu

ABSTRACT

Token-based obfuscation (TBO) is an interactive approach to cryptographic program obfuscation that was proposed by Goldwasser et al. (STOC 2013) as a potentially more practical alternative to conventional non-interactive security models, such as Virtual Black Box (VBB) and Indistinguishability Obfuscation. We introduce a query-revealing variant of TBO, and implement in PALISADE several optimized query-revealing TBO constructions based on (Ring) LWE for conjunctions and branching programs.

Our main focus is the obfuscation of general branching programs, which are asymptotically more efficient and expressive than permutation branching programs traditionally considered in program obfuscation studies. Our work implements read-once branching programs that are significantly more advanced than those implemented by Halevi et al. (ACM CCS 2017), and achieves program evaluation runtimes that are two orders of magnitude smaller. Our implementation introduces many algorithmic and code-level optimizations, as compared to the original theoretical construction proposed by Chen et al. (CRYPTO 2018). These include new trapdoor sampling algorithms for matrices of ring elements, extension of the original LWE construction to Ring LWE (with a hardness proof for non-uniform Ring LWE), asymptotically and practically faster token generation procedure, Residue Number System procedures for fast large integer arithmetic, and others.

ACM Reference Format:
Cheng Chen, Nicholas Genise, Daniele Miccicancio, Yuriy Polyakov, and Kurt Rohloff. 2020. Implementing Token-Based Obfuscation under (Ring) LWE. In *8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, December 15, 2020, Virtual Corona Edition.* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Program obfuscation has long been of interest to the cyber-security community. Obfuscated programs need to be difficult (computationally hard) to reverse engineer, and have to protect intellectual property contained in software from theft. For many years, practical program obfuscation techniques have been heuristic and have not provided secure approaches to obfuscation based on the computational hardness of mathematical problems. In this regard, there have been multiple recent attempts to develop cryptographically secure approaches to program obfuscation based on the computational hardness of mathematical problems (see [3] for a survey of these approaches). There are multiple definitions used for cryptographically secure program obfuscation. Two prominent definitions are Virtual Black Box and Indistinguishability Obfuscation.

Virtual Black Box (VBB) obfuscation is an intuitive definition of secure program obfuscation where the obfuscated program reveals nothing more than black-box access to the program via an oracle [23]. VBB is known to have strong limitations [4]. The most significant limitation is that *general-purpose* VBB obfuscation is unachievable [4].

To address limitations of VBB, Barak *et al.* [4] 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, 27]. There has also been recent work to implement multilinear map constructions [15, 24]. Recent results show that these constructions might not be secure [12, 14]. 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 [2]. Unfortunately, point functions have limited applicability.

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

An alternative approach to program obfuscation involves interactions with a trusted party, which allows one to build program

WAHC'20, December 15, 2020, Virtual Corona Edition

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 [5, 22]. In the TBO model, the user obtains a special token before executing the obfuscated program and then finds the decrypted result by herself [21]. 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 a scenario where a vendor publishes an intensive obfuscated program and provides tokens representing the rights to run the program on specific inputs. When a user wants to input a query x to the program, she also gets a token for x from the vendor, and then executes the obfuscated program. This allows the vendor to offload the computation to the user without fear of leaking nontrivial information about the program. Another key feature for TBO is the ability to obfuscate learnable functions which cannot be obfuscated under the VBB model. These learnable functions are why general-purpose VBB obfuscation does not exist in-general [4]. TBO can be used on learnable functions since the vendor controls what input-output data is revealed.

We emphasize the difference between TBO and garbled circuits (as commonly used in secure multi-party computation) in the vendor scenario above. In the case of garbled circuits, the size of the obfuscated program grows linearly with the number of function queries since each query requires an independently garbled circuit. However, TBO, which is based on the concept of a reusable garbled circuit, only requires one function obfuscation for many input queries (as described in the introduction of [21]).

Although TBO is interactive by definition, it can also be used in a non-interactive setting, like VBB or IO. Tokens can be generated in advance and then sent to the public evaluator along with the obfuscated program. In our constructions, each token requires only 128 bits or so (for a 128-bit security setting), while the obfuscated programs requires many megabytes. The public evaluator can then run the obfuscated program non-interactively only for allowed inputs. In other words, TBO can be used for non-interactive obfuscation of learnable functions, where both VBB and IO are inadequate.

Another interesting scenario is when token generation/output is done by an embedded system with tight hardware (memory/processor) constraints. This system is accessible to the public evaluator (for example, in the same physical location). The embedded system can either store pre-generated tokens or run light-weight token generation (if secure hardware, e.g., SGX, is used).

1.1 Our Contributions

Our work introduces a *query-revealing variant of TBO* (where input queries are in the clear), which is more efficient than the query-hiding TBO model proposed in [21] based on functional encryption/reusable garbled circuits. Alternativaly, our variant can be referred to as *reusable garbled circuits without input privacy*. This variant is adequate for most obfuscation scenarios as program inputs are typically not hidden. Query-revealing TBO (QR-TBO) thus provides an efficient method to obfuscate the classes of functions that can be learned.

We develop optimized constructions, implement them in PALISADE, and report experimental results for the TBO of conjunctions [8], permutation branching programs [8], and general branching programs [13].

Our most significant contribution is the optimized implementation for the TBO of general branching programs based on the theoretical construction for constrained-hiding constrained PRFs proposed in [13]. We evaluate the performance of our implementation for a program that finds a Hamming distance between two strings of equal length. The evaluation runtime of our implementation, which supports more than 500 accepting states, is two orders of magnitude smaller than the implementation [24] for a simpler program (with about 100 states; note that the construction from [24] was subsequently broken in [13]). The main optimizations introduced in our implementation include:

(1) Development of an efficient Residue Number System (RNS) ring variant of construction [13], requiring a hardness proof for non-uniform Ring LWE, and RNS scaling and lattice trapdoor sampling procedures.
(2) New algorithms for lattice trapdoor sampling of matrices of ring elements.
(3) Improved key generation and token generation algorithms (both runtime and storage requirements are reduced by about two orders of magnitude as compared to the original construciton.)
(4) A larger alphabet for encoding bits in the input, which reduces the multlinearity degree of the construction.
(5) Significantly tighter correctness constraints, which reduce the main functional parameter.
(6) Many code-level and system optimizations, which are of independent interest for other lattice primitives.

We also present efficient implementations of the TBO for permutation branching programs and conjunction programs. Our performance results for conjunction obfuscation 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 [15].

All our implementations of TBO constructions and lower-level lattice algorithms are added as modules to PALISADE, thus effectively providing a TBO toolkit that is made publicly available.

1.2 Related Work

The TBO construction in [21] is formulated for the case of hidden queries using reusable garbled circuits, which in their turn can be built on top of a functional encryption (FE) scheme. This implies that a TBO scheme can be derived from an FE scheme by treating a secret key for evaluating a specific function on encrypted data as a token.

General FE constructions are currently impractical. One approach is based on a combination of key-policy attribute-based encryption and fully homomorphic encryption [21]. The state-of-the-art results in key-policy attribute encryption [17] suggest these schemes are still inefficient, and hence their use in FE where each attribute bit needs to be encrypted with FHE is currently not practical. Initial experimental results for multi-input FE are presented in [9] but they are far from practical.

The main difference between QR-TBO used in our work and FE (TBO model in [21]) is that the input queries in our model are in the clear, just like in the non-interactive program obfuscation models. This enables more efficient constructions for TBO.

2 PRELIMINARIES

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

The discrete Gaussian distribution over a lattice $\Lambda \subset \mathbb{R}^n$ is defined with probability mass proportional to $\rho_{\mathbf{c},\sigma}(\mathbf{x}) = e^{-\pi\|\mathbf{x}-\mathbf{c}\|^2/\sigma^2}$ and is denoted as $\mathcal{D}_{\Lambda,\mathbf{c},\sigma}$, where $\mathbf{c} \in \mathbb{R}^n$ is the center and σ is the distribution parameter. If the center $\mathbf{c}$ 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 (representing a polynomial by its coefficient vector)[1].

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

2.1 Query-Revealing TBO

Here we define TBO with restricted queries. Our definition is similar to [21], though weaker since the input query x is in the clear. Let λ be a security parameter throughout the following two definitions.

Definition 2.1 (Query-Revealing TBO). Let $n = n(\lambda) \in \mathbb{N}$. A query-revealing TBO scheme for a class of circuits $\{C_n\}_{n\in\mathbb{N}}$, where each C_n is a set of n-bit-input circuits, is a tuple of probabilistic polynomial-time algorithms (Setup,Obfuscate, TokenGen) with the following properties:

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

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

Next, we define the security game in Figure 1. We abbreviate (Obfuscate, TokenGen) as (Obf, TG). In Figure 1, $\text{OS}(\cdot, C)[[\text{st}_S]]$ is an oracle that on input x from A_2, runs S_2 with inputs $C(x)$, x (note that it was $1^{|x|}$ in the query-hiding TBO of [21]), and the current state of S, st_S. S_2 responds with a fake tk_x^* and a new state st'_S which OS will feed to S_2 on the next call. OS returns tk_x^* to A_2.

Definition 2.2 (Security). The TBO scheme is secure if there exists a pair of PPT simulation algorithms (S_1, S_2) such that for all PPT adversaries (A_1, A_2), the two probabilistic experiments defined in Figure 1 are computationally indistinguishable $\{\text{Exp}^{\text{real}}_{\text{tOB,A}}(1^\lambda)\} \approx_c \{\text{Exp}^{\text{ideal}}_{\text{tOB,A,S}}(1^\lambda)\}$.

[1] $\mathcal{R}$ can be viewed as a lattice in $\mathbb{R}^n$ by mapping $a(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ to $(a_{n-1}, \cdots, a_0) \in \mathbb{Z}^n$.

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

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

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

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

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

Return α

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

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

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

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

Return α

Figure 1: TBO security game.

3 QR-TBO OF CONJUNCTIONS

We first consider a construction for the token-based obfuscation of conjunctions based on Ring LWE. Our TBO construction is a significantly optimized variant of the bit-fixing construction for constraint-hiding constrained PRFs proposed in Section 5.1 of [8]. We chose the example of conjunctions to give a fair comparison with a prior recent non-interactive (distributional VBB) conjunction obfuscation construction implemented in [15] and introduce several major optimizations used in the next section for the TBO of more general programs, i.e., branching programs.

Compared to the non-interactive conjunction obfuscation construction implemented in [15] (and originally formulated in [7]), the TBO construction has several advantages w.r.t. both security and efficiency. The construction [7, 15] 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 [15] (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).

3.1 Definition of Conjunctions

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

This type of conjunctions is used in machine learning to execute or approximate classes of classifiers [26]. Conjunctions can be used to encode binary classifiers 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 [15].

3.2 Conceptual Model

The conceptual workflow is defined as follows:

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

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

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

We next describe the algorithms for each function.

3.3 Algorithms for TBO Functions

The building blocks of the TBO construction for conjuctions, such as lattice trapdoor sampling and GGH15 directed encoding, are the same as for the distributional VBB conjunction obfuscation construction implemented in [15], 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 [8].

The main difference of the TBO model as compared to the distributional VBB model [15] 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 [8] 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

Algorithm 1 Key generation

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

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 the appendix of the full version [11]) and computing the unconstrained key as $\mathcal{L} \times b$ short ring elements.

As compared to the construction in [8], we optimized the master secret key generation to only sample short ring elements $s_{i,b}$ (without calling complex lattice trapdoor sampling for these short ring elements), which reduces the storage and speed complexity for the unconstrained key by a factor of $O(m^2)$. In the original construction, the size of the master key was approximately the same as the obfuscated program. In summary, the storage requirement for the keys in our construction is $O(\mathcal{L}bn) + O(\mathcal{L}(m + 2\kappa)n)$ integers in $\mathbb{Z}_q$ versus $O(m^2\mathcal{L}bn) + O(\mathcal{L}(m + 2\kappa)n)$ integers in the original construction. Storing the secret keys rather their GGH15 encodings does not effect the security of the construction in the TBO model as the trusted party is allowed to have access to the secret keys by definition.

Algorithm 2 Obfuscation

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

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 [8]. 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 [15]. The lattice trapdoor sampling procedure GaussSamp is described in the appendix of the full version of the paper [11].

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

Algorithm 3 Directed encoding

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

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

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

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 [8]: it multiplies short ring elements followed by a single scalar product with the second ring element of the public key $\mathbf{A}_0$ (in contrast to vector-matrix products in [8]), 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($\mathbf{x} \in \{0,1\}^L, \pi_v, \mathbf{y}'$)
$\mathbf{D}_\pi := \mathbf{A}_0$
for i = 1..$\mathcal{L}$ **do**
$\mathbf{D}_\pi := \mathbf{D}_\pi \, \mathbf{D}_{i,x[1+(i-1)w:iw]} \in {\mathcal{R}_q}^{1\times m}$
$\mathbf{y} := \lfloor \frac{2}{q}\mathbf{D}_\pi[1] \rceil \in \mathbb{Z}_2^n$
return $(\mathbf{y} = \mathbf{y}')$

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

Next, we note that the probability of comparison error is linear in the number of coefficients compared under the heuristic that the coefficients are independent and uniformly distributed over $\mathbb{Z}_q$. Let B be our bound on the GGH15 noise. Then, the probability of a rounding error in a comparison of the entire output is less than $(nmd)\frac{4B}{q}$ since there are two "bad" regions of $\mathbb{Z}_q$ of size $2B$ corresponding to rounding errors and there are nmd $\mathbb{Z}_q$-coefficients being rounded to bits (nmd bits). 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 (a noise bound is explicitly derived in Appendix C).

3.4 Security

The TBO construction for conjunctions is secure under Definition 2.1 for QR-TBO under Ring LWE. The proof showing that the existence of constraint-hiding constrained PRF implies an the existence of a QR-TBO scheme is presented Appendix B. Further, we are able to base security on plain RLWE instead of small-entry $\mathbf{A}$ RLWE [6]. A proof is in Appendix B for more details.

3.5 Setting the Parameters

Ring-LWE trapdoor construction. The trapdoor secret polynomials are generated with a noise width σ, which is at least the smoothing parameter estimated as $\sqrt{\ln(2n_m/\epsilon)/\pi}$, where n_m is the maximum ring dimension and ϵ is the bound on the statistical error introduced by each randomized-rounding operation [29]. 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 $s_{i,b}, r_{i,b}$ and noise ring elements, we use error distribution with the distribution parameter σ. This implies we rely on Ring-LWE for directed encoding.

***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 [29]. Lemma 2.9 of [29] states that $s_1(\mathbf{X}) \leq C_0\sigma\left(\sqrt{n\kappa} + \sqrt{2n} + C_1\right)$, where C_0 is a constant and C_1 is at most 4.7. We can now rewrite σ_s as $\sigma_s > C_0\sigma\sigma_t\left(\sqrt{n\kappa} + \sqrt{2n} + 4.7\right)$. In our experiments we used $C_0 = 1.3$, which was found empirically.

Modulus q. The correctness constraint for a conjunction pattern with $\mathcal{L}$ words ($\mathcal{L} \geq 2$) is expressed as $q > 2^{10}P_e^{-1}B_e\left(\beta\sigma_s\sqrt{mn}\right)^{\mathcal{L}-1}$, where $B_e = 6\sigma, \beta = 6, P_e = 2^{-20}$, and all other parameters are the same as in [15]. The derivation details are presented in Appendix C.

Ring Dimension n. All of the security proofs presented in [8] 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[2] (commit a2296b8) [1] to find the lowest security levels for the uSVP, decoding, and dual attacks following the standard homomorphic encryption security recommendations [10]. 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.

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

Dimension m. The dimension m was set to $2 + \kappa$ following the logic described in [15].

Word size w. We found $w = 8$ to be the optimal value for all our experiments, using the same procedure as in [15].

3.6 Comparison with Construction in [15]

As the building blocks and many underlying parameters for the TBO construction are the same as for the distributional VBB constructon [15], 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 [15] requires computing two product chains versus just one product chain in our TBO construction. All other parameters are the same. This implies that the TBO construction should be at least twice faster in obfuscation and evaluation, and requires 2x smaller storage for the obfuscated program. We provide their experimental comparison later in the paper.

From the security perspective, the TBO model can be used to bound the number of queries and restrict the format of inputs, thus overcoming the main security limitation of the conjunction obfuscation construction discussed in [15].

4 QR-TBO 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 [8]. For general branching programs, we adapt the private constrained PRF[3] construction of [13] (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 3 and then adds an extra layer dealing with matrix branching programs. Conceptually speaking, the TBO of conjunctions may be considered as a simple special case of the TBO for branching programs. In this section we focus on the aspects specific to branching programs, implying that all other underlying building blocks and parameters are the same as for the TBO of conjunctions.

Compared to the constructions in [13] and [8], 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.

4.1 Matrix Branching Programs

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

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

Definition 4.1. (Matrix branching programs [13]) 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 4.2. (Permutation branching programs: Type II branching programs in [13])

(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 4.2.

Definition 4.3. (General branching programs: Type I branching programs in [13]). 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 [13] for two specific representations of CNFs).

The relationships between these two types of branching programs are discussed in [13].

4.2 TBO Construction

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

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

Algorithm 6 Key generation for branching programs

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

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.

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

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 $\mathbf{M}$.

Algorithm 8 Directed encoding for matrices

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

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 [13], 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 5.1 and the appendix of the full version of the paper [11].

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

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

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 [13] and [8].

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 [13].

Algorithm 10 Evaluation using the obfuscated program

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

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

4.3 Security

The TBO construction for permutation branching programs is secure under Definition 2.1 for QR-TBO under Ring LWE. The proof showing that the existence of constraint-hiding constrained PRF (also referred to as private constrained PRF) implies the existence of a QR-TBO scheme is presented in Appendix B, and we can rely on plain RLWE since the construction from [8] only needs small-entry $\mathbf{A}$ RLWE for pseudorandomness, which is not a requirement in Definition 2.1. See Appendix B for more details.

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

4.4 Parameter Selection

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

4.5 Efficiency of Branching Programs

The general branching program represention is typically significantly more efficient than the permutation representation [13]. 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 [13], 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.

4.6 Application: Hamming Distance

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

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

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

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

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

5 EFFICIENT LATTICE TRAPDOOR ALGORITHMS

In this section, we describe the underlying lattice trapdoor mechanism used in our construction and its efficient algorithms. The trapdoor technique is an optimized instantiation of the MP12 framework [29] (which in turn is an optimized instantiation of [20]). In addition, we introduce an algorithm, SampleMat, used to sample a perturbation of arbitrary dimension over $\mathcal{R}$ efficiently. This algorithm may be of independent interest and is described in the appendix of the full version of the paper [11].

The pseudocode for trapdoor generation and sampling is given in the appendix of the full version of the paper [11]. In short, TrapGen takes as input a security parameter and outputs a pseudorandom matrix $\mathbf{A}$ over $\mathcal{R}_q$ along with a trapdoor matrix $\mathbf{T}$ with small entries over $\mathcal{R}$. This trapdoor $\mathbf{T}$ allows us to sample discrete Gaussian preimage vectors $\mathbf{x}$ over $\mathcal{R}$ such that $\mathbf{Ax} \mod q = \mathbf{u}$ for $\mathbf{u}$ given as an input. Sampling a discrete Gaussian matrix $\mathbf{X}$ over $\mathcal{R}$ where $\mathbf{AX} = \mathbf{U} \mod q$ is done by sampling each column of $\mathbf{X}$ independently.

5.1 Perturbation Sampling for the General Covariance Matrices of Ring Elements

We now discuss SampleMat, needed to extend the efficient perturbation sampling methods of [16] to the broad, module-LWE setting. Specifically, SampleMat replaces [16]'s algorithm Sample2z. This new algorithm may be of independent interest since it samples a discrete Gaussian perturbation with a covariance described as a matrix of any dimension over the ring $\mathcal{R}$ via the Schur complement method of [16]. We remark the proof of SampleMat's statistical correctness follows from [16, Theorem 4.1], whose proof only depends on the lattice dimension and is oblivious to the underlying algebraic structure or module dimension.

5.2 RNS Algorithms

We implemented all procedures for the TBO constructions of conjunctions and branching programs in the Double-CRT (RNS) representation, which supports parallel operations over vectors of fast native integers (64-bit for x86-64 architectures). The two procedures requiring special handling are the lattice trapdoor sampling in Encode and scale-and-round operation in TokenGen and Eval.

Lattice trapdoor sampling calls digit decomposition for each polynomial coefficient in the G-sampling step. The conventional digit decomposition is not compatible with RNS, and requires expensive conversion to the positional (multi-precision) format to extract the digits. Instead, we use a CRT representation of the gadget matrix that was recently proposed in [17], 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 [15], in the appendix of the full version of the paper [11].

For the scale-and-round operation, we utilize the RNS scaling procedure proposed in [25] 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.

6 IMPLEMENTATION AND RESULTS

6.1 Software Implementation

We implemented the TBO constructions in PALISADE v1.3.1 [30], 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 (1) cryptographic, (2) encoding, (3) lattice, and (4) arithmetic (primitive math). Our TBO toolkit is a new PALISADE module called "tbo", which includes the following new features broken down by layer: (1) TBO of conjunctions and branching programs in the cryptographic layer;

(2) variants of GGH15 encoding in the encoding layer; and (3) trapdoor sampling for matrices of ring elements in the lattice layer. Several lattice- and arithmetic-layer optimizations are also applied for runtime improvements. OpenMP loop parallelization is used to achieve speedup in the multi-threaded mode.

6.2 Experimental Testbed

The experiments for conjunction and branching program obfuscation 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.

6.3 TBO of Conjunctions

Table 1 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 [7, 15] 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 [15] 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 3.

We also examined the effect of OpenMP loop parallelization optimizations by comparing the results for single- and multi-threaded scenarios (Table 1). Here, we chose 14 (matching the number of cores per socket) as the number of threads as the main parallelization dimension in both evaluation and obfuscation is $m = 11$, and increasing the number of threads further than that degrades the performance due to multi-threading overhead. The speed-ups in the evaluation and obfuscation runtimes are 6.6x and 4.6x, respectively, with the maximum theoretical limit for this case being 11.

Our 64-bit conjunction obfuscation results are much less practical, mainly due to a large program size. On the other hand, they are significantly better than prior 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.

6.4 TBO of Branching Programs

Table 2 shows the performance results for the TBO of general branching programs using the Hamming distance problem as an example application. Note that $d = 5$ corresponds to the classical Barrington's theorem permutation branching program case. Hence these results can be used for benchmarking the TBO of both permutation and general branching programs of length $L = 24$ bits.

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

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

7 ACKNOWLEDGEMENTS

We are grateful for 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] Albrecht, M., Scott, S., Player, R.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology **9**(3), 169âĂŞ203 (10 2015)
[2] Bahler, L., Di Crescenzo, G., Polyakov, Y., et al., K.R.: Practical implementation of lattice-based program obfuscators for point functions. In: HPCS 2017
[3] Barak, B.: Hopes, fears, and software obfuscation. Commun. ACM **59**(3) (Feb 2016)
[4] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. J. ACM **59**(2) (May 2012)
[5] Bitansky, N., Canetti, R., Goldwasser, S., Halevi, S., Kalai, Y.T., Rothblum, G.N.: Program obfuscation with leaky hardware. In: ASIACRYPT 2011
[6] Boneh, D., Lewi, K., Montgomery, H.W., Raghunathan, A.: Key homomorphic prfs and their applications. In: CRYPTO 2013. pp. 410–428 (2013)
[7] Brakerski, Z., Vaikuntanathan, V., Wee, H., Wichs, D.: Obfuscating conjunctions under entropic ring lwe. ITCS '16
[8] Canetti, R., Chen, Y.: Constraint-hiding constrained prfs for nc^1 from LWE. In: EUROCRYPT 2017
[9] Carmer, B., Malozemoff, A.J., Raykova, M.: 5gen-c: Multi-input functional encryption and program obfuscation for arithmetic circuits. In: ACM CCS'17
[10] Chase, M., Chen, H., Ding, J., et al., S.G.: Security of homomorphic encryption. Tech. rep. (2017)
[11] Chen, C., Genise, N., Micciancio, D., Polyakov, Y., Rohloff, K.: Implementing token-based obfuscation under (ring) lwe. Cryptology ePrint Archive, Report 2018/1222 (2018), https://eprint.iacr.org/2018/1222
[12] Chen, Y., Gentry, C., Halevi, S.: Cryptanalyses of candidate branching program obfuscators. In: EUROCRYPT 2017
[13] Chen, Y., Vaikuntanathan, V., Wee, H.: Ggh15 beyond permutation branching programs: Proofs, attacks, and candidates. In: CRYPTO 2018
[14] Coron, J.S., Lee, M.S., Lepoint, T., Tibouchi, M.: Zeroizing attacks on indistinguishability obfuscation over clt13. In: Fehr, S. (ed.) PKC 2017. pp. 41–58 (2017)

Table 1: 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]	Obf. [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 [15]									
32	14	4096	180	15	36.8	12.4	–	–	53.0

Table 2: Execution times and program size for TBO of the branching program checking 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]	Obf. [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

[15] Cousins, D.B., Crescenzo, G.D., GÃijr, K.D., King, K., et al.: Implementing conjunction obfuscation under entropic ring lwe. In: 2018 IEEE SSP
[16] Genise, N., Micciancio, D.: Faster gaussian sampling for trapdoor lattices with arbitrary modulus. In: EUROCRYPT 2018
[17] Genise, N., Micciancio, D., Polyakov, Y.: Building an efficient lattice gadget toolkit: Subgaussian sampling and more. In: EUROCRYPT 2019
[18] Gentry, C., Gorbunov, S., Halevi, S.: Graph-induced multilinear maps from lattices. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. pp. 498–527 (2015)
[19] Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. pp. 850–867 (2012)
[20] Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: STOC '08. pp. 197–206 (2008)
[21] Goldwasser, S., Kalai, Y., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: STOC '13
[22] Goyal, V., Ishai, Y., Sahai, A., Venkatesan, R., Wadia, A.: Founding cryptography on tamper-proof hardware tokens. In: Theory of Cryptography (2010)
[23] Hada, S.: Zero-knowledge and code obfuscation. In: Okamoto, T. (ed.) ASIACRYPT 2000. pp. 443–457 (2000)
[24] Halevi, S., Halevi, T., Shoup, V., Stephens-Davidowitz, N.: Implementing bp-obfuscation using graph-induced encoding. In: ACM CCS '17. pp. 783–798 (2017)
[25] Halevi, S., Polyakov, Y., Shoup, V.: An improved rns variant of the bfv homomorphic encryption scheme. In: Matsui, M. (ed.) CT-RSA 2019. pp. 83–105 (2019)
[26] Kubat, M.: An Introduction to Machine Learning. Springer Publishing Company, Incorporated, 1st edn. (2015)
[27] Lin, H.: Indistinguishability obfuscation from sxdh on 5-linear maps and locality-5 prgs. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. pp. 599–629 (2017)
[28] Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. pp. 1–23 (2010)
[29] Micciancio, D., Peikert, C.: Trapdoors for lattices: Simpler, tighter, faster, smaller. In: Advances in Cryptology–EUROCRYPT 2012, pp. 700–718. Springer (2012)
[30] Polyakov, Y., Rohloff, K., Ryan, G.W.: PALISADE lattice cryptography library. https://git.njit.edu/palisade/PALISADE (Accessed November 2018)
[31] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6), 34:1–34:40 (2009)

A ADDITIONAL PRELIMINARIES

A.1 Double-CRT (RNS) Representation

Our implementation utilizes the Chinese Remainder Theorem (referred to as integer CRT) representation to break multi-precision integers in $\mathbb{Z}_q$ into vectors of smaller integers to perform operations efficiently using native (64-bit) integer types. The integer CRT representation is also often referred to as the Residue-Number-System (RNS) representation. We use a chain of same-size prime moduli $q_0, q_1, q_2, \ldots$ satisfying $q_i \equiv 1 \bmod 2n$. Here, the modulus q is computed as $\prod_{i=0}^{l-1} q_i$, where l is the number of prime moduli needed to represent q. All polynomial multiplications are performed on ring elements in polynomial CRT representation where all integer components are represented in the integer CRT basis. Using the notation proposed in [19], we refer to this representation of polynomials as "Double-CRT".

A.2 Ring Learning with Errors

The following distinguishing problem, originated by Regev and modified to an algebraic version [28], will be our source of cryptographic hardness.

Definition A.1. (Gaussian-secret, cyclotomic-RLWE). Let $\mathcal{R}$ be a power-of-two cyclotomic ring of dimension n over $\mathbb{Z}$, $q \geq 2$ be integer used as a modulus, and $m > 0$. Let $\mathcal{D}_{\mathcal{R},\sigma}$ be a discrete Gaussian distribution over $\mathcal{R}_q$ (sampled over $\mathcal{R}$, then taken modulo q). Then, the $(\mathcal{R}^l, m, q, \mathcal{D}_{\mathcal{R},\sigma}, \mathcal{D}_{\mathcal{R},\sigma})$RLWE problem is to distinguish between the following two distributions: $\{(\mathbf{A}, \mathbf{s}^T\mathbf{A} + \mathbf{e}^T)\}$ and $\{(\mathbf{A}, \mathbf{u}^T)\}$, where, $\mathbf{s} \leftarrow \mathcal{D}^l_{\mathcal{R},\sigma}$, $\mathbf{A} \leftarrow \mathcal{U}(\mathcal{R}_q)^{l\times m}$, $\mathbf{e} \leftarrow \mathcal{D}^m_{\mathcal{R},\sigma}$ and $\mathbf{u} \leftarrow \mathcal{U}({\mathcal{R}_q}^m)$[4].

A.3 GGH15 Encoding

We will use the generalized GGH15 construction [18] given in [13], called γ-GGH15. Here we give a brief description, though for a complete description, see Section 2 of [13]. The main idea is that the γ-GGH15 construction provides a way to multiply matrices homomorphically under the security of LWE.

First, we give the parameters and variables. Fix some ring $\mathcal{R}_q$. Let $\ell > 0$ be a fixed computation length, $(\mathbf{M}_{i,b} \in {\mathcal{R}_q}^{w\times w})_{i\in[\ell], b\in\{0,1\}}$ be a collection of binary, scalar matrices to be used as a form of computation, e.g. a matrix-branching program, and let $(s_{i,b} \in \mathcal{R}_q)_{i\in[\ell], b\in\{0,1\}}$ be a tuple ring elements. Let $\gamma(\mathbf{M}, s)$ be a function mapping $(\mathbf{M}, s)$ to another matrix satisfying $\gamma(\mathbf{M}, s)\gamma(\mathbf{M}', s') = \gamma(\mathbf{M}\mathbf{M}', ss')$. 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 $\mathbf{x} \in \{0,1\}^\ell$.

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

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

$\text{msk} \leftarrow \textsc{Gen}(1^\lambda)$

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

$\text{CK}_C \leftarrow \textsc{Constrain}(\text{msk}, \text{C})$

$\alpha \leftarrow \text{A}_2^{\textsc{Eval}(\text{msk},\cdot)}(\text{C}, \text{CK}_C, \text{st}_A)$

Return α

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

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

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

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

Return α

Figure 2: The one-key CHCPRF security games.

B QUERY-REVEALING TBO FROM CONSTRAINT-HIDING CONSTRAINED PRF

Here we prove that the existence of a constraint-hiding constrained pseudorandom function (CHCPRF) implies the existence of a query-revealing TBO scheme. First, we recall the definition of a (one-key) CHCPRF [8, 13].

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

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

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

(1) For all inputs $x \in D_\lambda$ s.t. $C(x) = 1$, we have $\Pr\{\textsc{Eval}(\text{msk}, x) = \textsc{Constrain.Eval}(\text{CK}_C, x)\} \geq 1 - \text{negl}(\lambda)$ where the probability is taken over Gen and Constrain's random coins.
(2) There exists a PPT simulator pair (S_1, S_2) such that for all PPT adversaries (A_1, A_2), the outputs of $\text{Real}_{\text{CHCPRF,A}}(1^\lambda)$ and $\text{Ideal}_{\text{CHCPRF,A,S}}(1^\lambda)$ are computationally indistinguishable (Figure 2).

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

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

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

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

as follows:

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

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

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

By the correctness of the CHCPRF scheme, we have $O(tk_x) = 1 = C(x)$ with $1 - \text{negl}(\lambda)$ probability whenever $C(x) = 1$. Further, we have $F_{\text{osk}}(x) \neq F_{\text{CK}_C}(x)$ when $C(x) = 0$ with high probability since $|R_\lambda| = \lambda^{\omega(1)}$ along with the PRF property[5]. The rest of the proof follows from the definition of the security games for qr-TBO and (one-key) CHCPRF.

Real games. First, we show for all adversaries the real games have the same distribution. Consider a fixed adversary (A_1, A_2), then the distribution

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

is exactly the distribution generated in the real qr-TBO game (Definition 2.2) with adversary (A_1, A_2),

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

Ideal games. Next, we consider the ideal CHCPRF game and show for all simulators and adversaries, there exists a simulator pair so the ideal games have the same distribution (with the adversaries unchanged). Let (S_1, S_2) be PPT simulators and (A_1, A_2) be PPT adversaries again. Let $\text{S}_1^{\text{TBO}} := \text{S}_1$. Then, let $\text{S}_2^{\text{TBO}}(x, C(x), \text{st}_S)$ consist of the following steps:

(1) First, it runs S_2, $(y, \text{st}'_S) \leftarrow \text{S}_2(x, C(x), \text{st}_S)$.
(2) Next, it will return (y, st'_S) if $C(x) = 1$, or it will overwrite y with a uniformly sampled element in the range, $u \leftarrow \mathcal{U}(R_\lambda)$, and return (u, st_S) if $C(x) = 0$.

Now, the distribution of $\{(\text{C}, \text{st}_A, \text{CK}_C^*, \alpha)\}$ in the ideal-CHCPRF is the same as the distribution of $\{(\text{C}, \text{st}_A, \text{O}^*, \alpha)\}$ in the ideal game of the qr-TBO game. (The description of S_2^{TBO} merely accounts for the differing behaviors of the query/evaluation oracles in the two games.)

Bridging the games. Finally, we let (S_1, S_2) be the PPT simulators such that the real and ideal CHCPRF games are computationally indistinguishable. The equivalences given above show the ideal qr-TBO game with simulators $(\text{S}_1^{\text{TBO}}, \text{S}_2^{\text{TBO}})$ is computationally indistinguishable from the real qr-TBO game for any adversary (A_1, A_2)[6]. □

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

[6] Note, the correctness of the qr-TBO scheme does not need the PRF property (a security property) when $C(x) = 0$. Instead, all we need is $\textsc{Eval}(\text{osk}, x) \neq \textsc{Constrain.Eval}(\text{CK}_C, x)$ with high probability whenever $C(x) = 0$, a much weaker property than being a PRF. This freedom from the PRF requirement allows us to base the security of our conjunctions and permutation branching programs on regular

C NOISE ANALYSIS FOR TOKEN-BASED OBFUSCATION OF CONJUNCTIONS

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

$$\left\|\mathbf{A}_0\prod_{i=1}^{\mathcal{L}}\mathbf{D}_{i,x_i}-\prod_{i=1}^{\mathcal{L}}s_{i,x_i}\cdot\mathbf{A}_L\right\|_\infty=$$

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

Here, we used the Central Limit Theorem (subgaussian analysis) and the following bounds: $\|s_{i,x_i}\|_\infty\leq 6\sigma, \|\mathbf{e}_{j,x_j}\|_\infty\leq 6\sigma, \|\mathbf{D}_{k,x_k}\|_\infty\leq 6\sigma_s$.

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

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

D NOISE ANALYSIS FOR TOKEN-BASED OBFUSCATION OF BRANCHING PROGRAMS

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

$$\left\|\mathbf{A}_0\prod_{i=1}^{\mathcal{L}}\mathbf{D}_{i,x_i}-\prod_{i=1}^{\mathcal{L}}\gamma(\widehat{\mathbf{M}}_{i,x_i},s_{i,x_i})\cdot\mathbf{A}_L\right\|_\infty=$$

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

Here, we used the Central Limit Theorem (subgaussian analysis) and the following bounds: $\left\|\gamma(\widehat{\mathbf{M}}_{i,x_i},s_{i,x_i})\right\|_\infty\leq 6\sigma, \|\mathbf{E}_{j,x_j}\|_\infty\leq 6\sigma, \|\mathbf{D}_{k,x_k}\|_\infty\leq 6\sigma_s$.

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}.

RLWE and *not* "non-uniform" RLWE (Gaussian $\mathbf{A}$ instead of uniformly random as used in [6]). This is given explicitly by Theorems 5.4 and 5.8 in [8][7]. Correctness for our schemes based on [8] was confirmed experimentally.

E NON-UNIFORM RING LWE

Extending the security proof [13, Thm 7.5] of private constrained PRFs to cyclotomic rings assumes the hardness of RLWE (or GLWE) with discrete Gaussian public samples, $a\in\mathcal{R}_q$ in the equation $a\cdot s+e$. We prove the security of this GLWE variant since its pseudorandomness is, at first glance, not obvious and it is the main step needed to extend [13] to the GLWE setting. The proof of the following theorem is adapted from [6], Section 4, with slightly better parameters. We remark that this is only needed for extending the security proof and our construction for general branching programs use a pseudorandom GLWE matrix $\mathbf{A}$ over $\mathcal{R}_q$ (with large entries).

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

Theorem E.1. *(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 E.1) 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 use of discrete Gaussian G-lattice sampling algorithms, or that we can efficiently sample a discrete Gaussian just above the G-lattice's smoothing parameter.

We can pad $\mathbf{G}=\mathbf{I}_d\otimes\mathbf{g}^T$ with columns of all 0s in ${\mathcal{R}_q}^d$ so that we can efficiently sample a discrete Gaussian just above the G-lattice's smoothing parameter easily extends to any $d'\geq d\log_t q$.

Given an input $(\mathbf{A},\mathbf{u})\in{\mathcal{R}_q}^{d\times m}\times{\mathcal{R}_q}^m$, we perform these 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 $\mathbf{A}=\mathbf{G}\mathbf{B}$.
(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 [31] 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[8]. □

[8] Since the base t can be chosen as a large parameter, the dimension-increase from RLWE to non-uniform GLWE can be small in-practice. Therefore, an increase in the dimension and the Gaussian width of the secrets in Section 4 leads to a TBO scheme for general branching programs *provably* secure from RLWE using the reductions in [13] along with Theorem E.1.