Implementing Conjunction Obfuscation under Entropic Ring LWE

David Bruce Cousins*, Giovanni Di Crescenzo†, Kamil Doruk Gür‡, Kevin King§, Yuriy Polyakov‡‖, Kurt Rohloff‡‖, Gerard W. Ryan‡ and Erkay Savaş†¶

* Raytheon BBN Technologies, dave.cousins@raytheon.com
† Applied Communication Sciences / Vencore Labs, gdicrescenzo@vencorelabs.com
‡ NJIT Cybersecurity Research Center, New Jersey Institute of Technology, {kg365,polyakov,rohloff,gwryan}@njit.edu
§ Massachusetts Institute of Technology, kcking@mit.edu
¶ Sabancı University, erkays@sabanciuniv.edu
‖ Corresponding Authors

November 30, 2017

Abstract

We address the practicality challenges of secure program obfuscation by implementing, optimizing, and experimentally assessing an approach to securely obfuscate conjunction programs proposed in [1]. Conjunction programs evaluate functions $f(x_1,\ldots,x_L)=\bigwedge_{i\in I} y_i$, where y_i is either x_i or $\neg x_i$ and $I\subseteq[L]$, and can be used as classifiers. Our obfuscation approach satisfies distributional Virtual Black Box (VBB) security based on reasonable hardness assumptions, namely an entropic variant of the Ring Learning with Errors (Ring-LWE) assumption. Prior implementations of secure program obfuscation techniques support either trivial programs like point functions, or support the obfuscation of more general but less efficient branching programs to satisfy Indistinguishability Obfuscation (IO), a weaker security model. Further, the more general implemented techniques, rather than relying on standard assumptions, base their security on conjectures that have been shown to be theoretically vulnerable.

Our work is the first implementation of non-trivial program obfuscation based on polynomial rings. Our contributions include multiple design and implementation advances resulting in reduced program size, obfuscation runtime, and evaluation runtime by many orders of magnitude. We implement our design in software and experimentally assess performance in a commercially available multi-core computing environment. Our implementation achieves runtimes of 6.7 hours to securely obfuscate a 64-bit conjunction program and 2.5 seconds to evaluate this program over an arbitrary input. We are also able to obfuscate a 32-bit conjunction program with 53 bits of security in 7 minutes and evaluate the obfuscated program in 43 milliseconds on a commodity desktop computer, which implies that 32-bit conjunction obfuscation is already practical. Our graph-induced (directed) encoding implementation runs up to 25 levels, which is higher than previously reported in the literature for this encoding. Our design and implementation advances are applicable to obfuscating more general compute-and-compare programs and can also be used for many cryptographic schemes based on lattice trapdoors.

CONTENTS

I Introduction 4
- I-A Our Contributions 5
- I-B Related Work 5
- I-C Organization 6

II Preliminaries 6
- II-A Conjunction Programs and Their Applications 6
- II-B Cyclotomic Rings 7
- II-C Cyclotomic Fields 7
- II-D Double-CRT Representation 7
- II-E Ring Learning with Errors Problem 7

III Conjunction Obfuscator 8
- III-A Overview 8
- III-B Ring Instantiation of Directed Encoding 9
- III-C Word Encoding Optimization 9

IV Trapdoor Sampling 9
- IV-A Overview and Motivation 9
- IV-B Trapdoor Construction and G-Lattice Representation 10
- IV-C High-Level Trapdoor Sampling Algorithm 11
- IV-D Sampling G-lattices 11
- IV-E Perturbation Sampling 12
- IV-F Integer Gaussian Sampling 12

V Setting the Parameters 12
- V-A Distribution Parameters 12
  - V-A1 Distribution Parameter for Ring-LWE Trapdoor Construction 13
  - V-A2 Short Ring Elements in Directed Encoding 13
  - V-A3 Distribution Parameters for Directed Encoding 13
  - V-A4 Distribution Parameter for G-Sampling 13
  - V-A5 Spectral norm s 13
- V-B Conjunction Obfuscator Correctness 13
- V-C Security 13
  - V-C1 Ring Dimension n 13
  - V-C2 Dimension m 14
  - V-C3 Work Factors t_{VBB} and t_{RLWE} 14
  - V-C4 Small-Secret Ring-LWE vs Error-Secret Ring-LWE for Directed Encoding 14
- V-D Word Size w 15
- V-E G-Lattice Base t 15

VI Efficient Matrix and Polynomial Arithmetic 16
- VI-A Matrix Chain Product in the Evaluation 16
- VI-B Efficient Polynomial Arithmetic 16
  - VI-B1 Double-CRT Operations 16
  - VI-B2 Number Theoretic Transform 17
  - VI-B3 Cyclotomic Fields 17
  - VI-B4 Polynomial Transposition 17
  - VI-B5 Modular Arithmetic 17

VII Implementation Details 17
VII-A Pseudocode of Obfuscation Scheme Algorithms . . . 17
VII-B Integer Sampling . . . 17
VII-C Software Implementation . . . 17
VII-D Loop Parallelization . . . 18

VIII Experimental results 18
VIII-A Testbed . . . 18
VIII-B Integer Gaussian Sampling Experiments . . . 18
VIII-C Experiments for the Word Size of One Byte . . . 19
VIII-D Experiments for the Word Size of One Bit . . . 19
VIII-E Parallelization Experiments . . . 20

IX Concluding Remarks 21

X Acknowledgements 21

References 21

Appendix A: Pseudocode for Trapdoor Sampling Algorithms 23

Appendix B: Derivation of Correctness Constraint for Conjunction Obfuscator 25

Appendix C: Comparison of Integer Gaussian Samplers 26

Appendix D: Pseudocode for Conjunction Obfuscation Algorithms 27

I. Introduction

Program obfuscation has long been of interest in the cyber-security community. Obfuscated programs should be difficult to reverse engineer, and should protect intellectual property contained in software from theft. This prevents the identification of exploits usable by adversaries.

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. Prior techniques are discussed in [2], [3], [4], [5], [6], [7]. Although often usable in practice, these prior 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 recent 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]. Recent results show that these constructions might not be secure [28], [29], [30], [31], [32], [33], [34], [35], [36]. The only IO construction supporting general functions that is not subject to any attack to date is the work by Garg *et al.* [37]. These cryptographically secure program obfuscation capabilities have also been considered impractical due to their computational inefficiency.

There have been attempts to securely obfuscate *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 [38], [39], [40], [41], [42]. Unfortunately, point functions have limited applicability.

We address the practicality challenges of secure program obfuscation by implementing, optimizing, and experimentally evaluating an approach proposed in [1] to securely obfuscate programs that execute conjunction functions, which are significantly more complex than point functions. Conjunction programs evaluate functions $f(x_1, \ldots, x_L) = \bigwedge_{i \in I} y_i$, where y_i is either x_i or $\neg x_i$ and $I \subseteq [L]$.

The obfuscation of conjunction programs is explored in [43] using the graded-encoding (multi-linear map) candidate construction from [20]. This prior approach is modified for an approach that is based on a graph-induced multi-linear map construction in [21] and secure under an entropic variant of the Ring-LWE assumption [1]. The obfuscation scheme satisfies *distributional VBB security*, meaning that the obfuscated program reveals nothing more than black-box access to the conjunction function via an oracle, as long as the conjunction is chosen from a distribution having sufficient entropy. The original work [1] focused on theoretical feasibility of conjunction obfuscation under entropic Ring-LWE and did not examine practicality issues of the construction.

To address the practicality of obfuscating conjunction programs, we introduce major design and system-level improvements compared to [1] that enable obfuscation and evaluation procedures both in server and desktop computing environments. These improvements include the use of optimized Gaussian sampling for lattice trapdoors and arbitrary-base gadget matrix, word-based encoding of programs (instead of binary encoding), optimized correctness constraint and parameter selection, efficient polynomial multiplication in double Chinese Remainder Transform (CRT) representation, optimized matrix arithmetic, and loop parallelization at multiple levels of the implementation. We implement this scheme in a C++ cryptographic library with multi-threading support.

Our implementation achieves runtimes of 6.7 hours to securely obfuscate a 64-bit conjunction program, and 2.5 sec. to evaluate this program over an arbitrary input in a server computing environment. The obfuscated program size is about 750 GB. For a 32-bit conjunction program, we report the obfuscation runtime of 7.0 min. and evaluation runtime of 43ms in a desktop computing environment, with the obfuscated program size under 6 GB.

A. Our Contributions

We implement the conjunction obfuscator on top of PALISADE[1], an open-source lattice cryptography library. We add new modules in PALISADE including the following:

1) **Gaussian lattice trapdoor sampler** for power-of-two cyclotomic rings. This implementation supports arbitrary moduli, including primes and products of primes, and performs all computations without explicit generation of a Cholesky decomposition matrix, which was a bottleneck of previous implementations based on [44]. Our implementation also supports a gadget matrix with an arbitrary base, which is computationally and spatially much more efficient than the classical binary gadget matrix.
2) **Generic integer Gaussian samplers**, including recent Karney's rejection [45] and constant-time [46] samplers. These samplers can be used for any integer Gaussian sampling operation in lattice-based cryptography.
3) **Implementation of directed encoding**, a special case of GGH15 multi-linear map construction.
4) **Extended Double-CRT** support to perform trapdoor sampling and obfuscation-related operations using native integer data types.
5) **Efficient matrix arithmetic** to support fast evaluation of inputs using the obfuscated conjunction program.
6) **Multi-threading and loop parallelization** support for all operations of conjunction obfuscator and certain lower-level matrix operations.

Our implementation includes several major original design improvements of the obfuscation scheme [1]:

1) **Word encoding** of conjunction program compared to the binary alphabet used in [1], which results in the reduction of obfuscated program size and obfuscation/evaluation runtimes by many orders of magnitude.
2) **Efficient ring-based trapdoor construction and preimage sampling**, which substantially reduces the obfuscation runtime and storage requirements.
3) **Dramatically reduced dimensions of encoding matrices** due to the use of a gadget matrix with a large base, which allow us to reduce program size and obfuscation/evaluation runtimes by multiple orders of magnitude.
4) **Improved bounds on parameters** coming from more careful analysis of the matrix/polynomial products and use of the Central Limit Theorem.

B. Related Work

Prior implementation work on secure program obfuscation beyond point obfuscation includes [25], [26], [27].

The first imlementation attempt [25] is based on the CLT13 encoding [47]. The authors build a branching program (BP) that obfuscates point functions. The obfuscation time for a 14-bit point function and 60-bit security is 9 hours, the program size is 31 GB and the evaluation of a single input takes 3.3 hours. Better results are reported in [26] which shows a framework for BP obfuscation using both CLT13 and GGH13 [48] multi-linear map encodings. The obfuscation time for an 80-bit point function using CLT13 with 80-bit security is 3.3 hours, obfuscated program size is 8.3 GB and evaluation time is 180 seconds. Note that the above two studies implement multi-linear map constructions to IO not believed to be secure [28], [29], [30], [31], [32], [33].

Halevi et al. [27] present an implementation of a simplified variant of GGH15 [21] to obfuscate oblivious read-once BPs, i.e., nondeterministic finite automata, of at most 80 bits with over 100 states. The GGH15 encoding is more efficient than CLT13 and GGH13 for larger numbers of states (over approximately 50 states), and presently appears to be immune to existing attacks in the obfuscation scenarios. Obfuscation takes 23 days, obfuscated program size is 9 TB, and evaluation takes 25 min. The maximum BP length is 20.

Our implementation is based on GGH15 [21] and entropic Ring-LWE, which is different from [25], [26]. We use a different security model for the obfuscation of a special-purpose function satisfying distributional VBB security, rather than BP obfuscation satisfying IO. Hence, results can be compared only indirectly. The results of this comparison are:

1) Our evaluation time for a 64-bit conjunction program (2.5 seconds), which is often the main runtime metric when assessing the practicability of program obfuscation, is significantly smaller than the one reported in [27] for the same bit length (949 seconds) and is smaller than the runtime for an 80-bit point function with 80-bit security in [26] (180 seconds).

[1] https://git.njit.edu/palisade/PALISADE

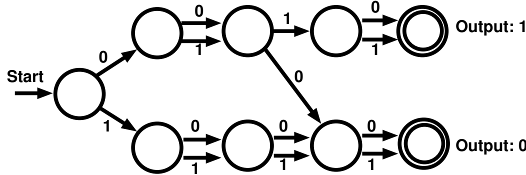

Fig. 1: Sample conjunction program that accepts [0⋆1⋆].

2) Our evaluation time for 20 levels of directed encoding for binary alphabet (188 seconds) is also smaller than the corresponding evaluation time in [27] (1514 seconds).
3) Our obfuscation time for a 64-bit pattern is 6.7 hours vs. 87 hours in [27].
4) The number of states supported by the conjunction obfuscator can be much higher than 100 (which is larger than in [25], [26], [27]) and is an exponential function of the number of "wildcards" in the conjunction pattern.
5) Our conjunction obfuscation does not include any randomizing as in BP obfuscation and, hence, requires the conjunction pattern to have high entropy to be VBB-secure, which is a drawback of our approach.

Although we take a software-only-based approach to program obfuscation, hardware-based approaches are also feasible. The work [49] achieves simulation-secure obfuscation for RAM programs using secure hardware to circumvent previous impossibility results.

There are related efforts to provide designs and implementations of obfuscation capabilities. Many building blocks of our implementation can be used to obfuscate compute-and-compare programs, a generalization of conjunctions, using the recently proposed construction based on LWE [50]. Another similar generalization is lockable obfuscation [51]. These more general constructions have not been implemented yet.

C. *Organization*

The rest of the paper is organized as follows: Section II provides the preliminaries of conjunction programs and lattices. Section III describes the conjunction obfuscator under entropic Ring-LWE and introduces our word encoding optimization. Section IV presents our optimizations of lattice trapdoor sampling focusing on the G-lattice generalization to arbitrary bases. Section V discusses the selection of parameters to optimize program size and runtimes. Section VI discusses our algorithms for efficient polynomial and matrix operations. Sections VII and VIII provide implementation details and performance evaluation of conjunction obfuscator. The paper concludes in Section IX. Appendices provide pseudocode for trapdoor sampling and conjunction obfuscation procedures, experimental results for integer Gaussian sampling, and derivation details.

II. Preliminaries

A. *Conjunction Programs and Their Applications*

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".

Conjunctions are used in machine learning to execute or approximate classes of classifiers [52], [53]. We can represent linear classifiers as conjunction programs, and we have used obfuscated conjunction programs to support Optical Character Recognition (OCR) applications on the standard MNIST dataset, for example, [54]. In this application, we use principal component analysis (PCA) techniques to construct a hypercube linear classifier for specific characters that we obfuscate using our conjunction obfuscation technique.

Figure 1 shows a sample conjunction program represented as the accepting language of a finite state machine where binary inputs drive state transitions. In this example the program accepts the input string [0⋆1⋆], where $\star$ represents a "wildcard input". This program accepts all 4-bit strings where the first bit is a 0 and the third bit is a 1. The second and fourth bits in the program are wildcards, meaning either 0 or 1 inputs in these locations can lead to accepting states.

We discuss below how one can group bits into larger alphabets of inputs using word encoding and we are not bound to binary inputs for conjunction programs. With these larger encodings, conjunction programs can be used to represent L^{∞}-norm and hypercube description region classifiers, among others [55], [56].

B. 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 [57], 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 using Fermat Theoretic Transform (FTT) [58] 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 $\mathcal{D}_{\mathcal{R},\sigma}$.

We use $\mathcal{U}_q$ to denote discrete uniform distribution over $\mathbb{Z}_q$ and $\mathcal{R}_q$. $\mathcal{T}$ denotes discrete ternary uniform distribution over $\{-1, 0, 1\}^n$. We define $k = \lceil \log_2 q \rceil$ as the number of bits required to represent integers in $\mathbb{Z}_q$.

C. 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 rather than the modular ones.

D. Double-CRT 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. We use a chain of same-size prime moduli $q_0, q_1, q_2, \ldots$ satisifying $q_i \equiv 1 \bmod \hat{m}$. 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 [59], we refer to this representation of polynomials as "Double-CRT".

E. Ring Learning with Errors Problem

Our scheme is based on a special case of the Ring-LWE problem [60] introduced in Definition 1. Let us define an operator $\mathbf{MakePoly}$ such that for all rings R, if $\mathbf{a} \in R^n$, then $\mathbf{MakePoly}(\mathbf{a}) \in R[x]$ is the polynomial whose coefficients are the elements of $\mathbf{a}$. If D is a distribution over R^n, then $\mathbf{MakePoly}(D)$ is the respective distribution over $R[x]$.

Definition 1 ($\mathbf{PLWE}_{n,m,q,\chi}$). *Let n be a power of two, and let $\mathcal{R} = \mathbb{Z}[x]/\langle x^n + 1\rangle$. Let $q = 2^{\omega(\log \lambda)}$, where λ is a security parameter, be such that $q \equiv 1 \pmod{2n}$ and define $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. Let $m \in \mathbb{N}$ and let χ be a distribution over the integers. The $\mathbf{PLWE}_{n,m,q,\chi}$ problem is the problem of distinguishing $\{(a_i, a_i \cdot s + e_i \pmod{x^n + 1, q})\}_{i\in[m]}$ from $\{(a_i, u_i)\}_{i\in[m]}$, where $s, e_i \leftarrow \mathbf{MakePoly}(\chi^n)$ and $a_i, u_i \leftarrow \mathbf{MakePoly}(\mathbb{Z}_q^n)$.*

In our implementation, we also use a modification of Definition 1 where $s \leftarrow \mathbf{MakePoly}\left(\mathcal{T}\right)$. This variant is often referred to as a small-secret case of Ring-LWE.

Prior to defining the entropic variant of the $\mathbf{PLWE}_{n,m,q,\chi}$ problem, we introduce $\widetilde{\mathbf{H}}_{\infty}\left(X|Z\right)$ as follows:

Definition 2 (Average Min-Entropy)**.** *Let X and Z be (possibly dependent) random variables, the average min-entropy of X conditioned on Z is*

$$\widetilde{\mathbf{H}}_{\infty}\left(X|Z\right) = -\log\left(\mathop{\mathbb{E}}_{z \leftarrow Z}\left[2^{-\mathbf{H}_{\infty}\left(X|Z=z\right)}\right]\right),$$

where $\mathbf{H}_{\infty}\left(Y\right)$ is the min-entropy of random variable Y.

Conceptually the min-entropy is the smallest of the Rényi family of entropies, which corresponds to the most conservative way of measuring the unpredictability of a set of outcomes. In this case, we deal with its averaged expression.

The entropic version of the $\mathbf{PLWE}_{n,m,q,\chi}$ problem is defined as follows:

Definition 3 (α-Entropic $\mathbf{PLWE}_{n,m,q,\chi}$)**.** *Let m, n, q, χ be parameters of λ and $\mathcal{R}_q$ as in Definition 1, and let $D = \{D_\lambda\}$ be an efficiently samplable distribution with $(x, z) \leftarrow D_\lambda$ having $x \in \{0,1\}^{\ell}$ for some $\ell = \ell(\lambda)$ and $\widetilde{\mathbf{H}}_{\infty}\left(x|z\right) \geq \alpha(\lambda)$. The α-entropic $\mathbf{PLWE}_{n,m,q,\chi}$ problem is to distinguish*

$$\left(\{s_j\}_{j\in[\ell]}, z, \{(a_i, a_i \cdot s + e_i)\}_{i\in[m]}\right)$$

from

$$\left(\{s_j\}_{j\in[\ell]}, z, \{(a_i, u_i)\}_{i\in[m]}\right),$$

where $s_j, e_i \leftarrow \mathbf{MakePoly}\left(\chi^n\right)$, $s = \prod_{j\in[\ell]} s_j^{x_j}$, and
$a_i, u_i \leftarrow \mathbf{MakePoly}\left(\mathbb{Z}_q^n\right)$.

In our implementation, we use a modification of Definition 3 where $s_j \leftarrow \mathbf{MakePoly}\left(\mathcal{T}\right)$. This variant will be referred to as a small-secret case of entropic Ring-LWE.

III. Conjunction Obfuscator

A. Overview

We first formulate the abstract conjunction obfuscator using the definition developed in [43].

To obfuscate a conjunction $F_{\mathbf{v}}$ with $\mathbf{v} \in \{0,1,\star\}^L$, we perform the following steps:

- Choose random short ring elements $\{s_{i,b}, r_{i,b} : i \in [L], b \in \{0,1\}\}$ subject to $s_{i,0} = s_{i,1}$ if $v_i = \star$.
- Create encodings $\mathbf{R}_{i,b}$ of $r_{i,b}$ and encodings $\mathbf{S}_{i,b}$ of $s_{i,b} \cdot r_{i,b}$ under $\mathbf{A}_{i-1} \rightarrow \mathbf{A}_i$ (the specific encoding technique used in our implementation is described in III-B).
- Choose a random short ring element r_{L+1}. Create an encoding $\mathbf{R}_{L+1}$ of r_{L+1} and encoding $\mathbf{S}_{L+1}$ of $r_{L+1}\prod_{i=1}^{L} s_{i,v_i}$. These encodings are under $\mathbf{A}_L \rightarrow \mathbf{A}_{L+1}$.

We set the obfuscated program to be

$$\Pi_{\mathbf{v}} = \left(\mathbf{A}_0, \{\mathbf{S}_{i,b}, \mathbf{R}_{i,b}\}_{i\in[L], b\in\{0,1\}}, \mathbf{R}_{L+1}, \mathbf{S}_{L+1}\right).$$

To evaluate $\prod_{\mathbf{v}}$ on an input $\mathbf{x} \in \{0,1\}^L$, we compute

$$\mathbf{S}^* = \left(\prod_{i=1}^{L} \mathbf{S}_{i,x_i}\right)\mathbf{R}_{L+1}, \ \mathbf{R}^* = \left(\prod_{i=1}^{L} \mathbf{R}_{i,x_i}\right)\mathbf{S}_{L+1}.$$

If $F_{\mathbf{v}} = 1$, then both $\mathbf{S}^*$ and $\mathbf{R}^*$ are encodings of the same value $r_{L+1}\prod_{i=1}^{L} s_{i,v_i}$ under $\mathbf{A}_0 \rightarrow \mathbf{A}_{L+1}$, and if $F_{\mathbf{v}} = 0$, then $\mathbf{S}^*$ and $\mathbf{R}^*$ are extremely unlikely to encode the same value, i.e., the probability of this event is $\mathrm{negl}\left(\lambda\right)$. Therefore, we can compute the output of the program by testing the equality of encoded values using $\mathbf{EqualTest}_{\mathbf{A}_0 \rightarrow \mathbf{A}_{L+1}}\left(\mathbf{S}^*, \mathbf{R}^*\right)$.

B. Ring Instantiation of Directed Encoding

We implement an instantiation of conjunction obfuscator based on a directed encoding scheme, which is a special case of GGH15 graph-induced multi-linear maps [21], specialized to a line. The ring instantiation of the directed encoding scheme for the case of cyclotomic rings $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n+1\rangle$, which was originally proposed in [1], is described below:

- **KeyGen** $\left(1^\lambda, 1^d\right)$ takes as input a security parameter λ and upper bound d on the number of levels, runs lattice trapdoor generation algorithm **TrapGen** $\left(1^\lambda\right)$ (defined in Algorithm 1) and outputs

$$(\mathbf{PK}_i, \mathbf{EK}_i) = (\mathbf{A}_i, \mathbf{T}_i) \in {\mathcal{R}_q}^{1\times m} \times \mathcal{R}^{m\times\kappa},$$

  where $i \in \{0,\ldots,d\}$ and m and κ are two trapdoor-related parameters explained in Section IV.
- **Encode**$_{\mathbf{A}_i\to\mathbf{A}_{i+1}}(\mathbf{T}_i, r)$, where $r \in \mathcal{R}$, is performed in two steps
  - Compute $\mathbf{b}_{i+1} := r\mathbf{A}_{i+1} + \mathbf{e}_{i+1} \in {\mathcal{R}_q}^{1\times m}$, where $\mathbf{e}_{i+1} \leftarrow \mathcal{D}_{\mathcal{R}^{1\times m},\sigma}$.
  - Output a matrix

$$\mathbf{R}_{i+1} \leftarrow \mathbf{GaussSamp}\left(\mathbf{A}_i, \mathbf{T}_i, \mathbf{b}_{i+1}, \sigma_t, s\right) \in \mathcal{R}^{m\times m},$$

    where **GaussSamp** is the preimage sampling algorithm discussed in Section IV and σ_t and s are distribution parameters defined in Section V-A.

  Note that $\mathbf{A}_i\mathbf{R}_{i+1} = \mathbf{b}_{i+1} = r\mathbf{A}_{i+1} + \mathbf{e}_{i+1} \in {\mathcal{R}_q}^{1\times m}$.
- **REncode**$_{\mathbf{A}_i\to\mathbf{A}_{i+1}}\left(1^\lambda\right)$ is the public encoding procedure that simply samples a matrix $\mathbf{R}_{i+1} \leftarrow \mathcal{D}_{\mathcal{R}^{m\times m},\sigma}$.
- **Mult** $(\mathbf{R}_1, \mathbf{R}_2) = \mathbf{R}_1\mathbf{R}_2$, where multiplication is performed over $\mathcal{R}_q$.
- **EqualTest**$_{\mathbf{A}_0\to\mathbf{A}_i}(\mathbf{R}_1, \mathbf{R}_2)$ outputs **1** for "**accept**" if

$$\left\|\mathbf{A}_0\left(\mathbf{R}_1 - \mathbf{R}_2\right)\right\|_\infty \leq q/8$$

  and **0** for "**reject**" otherwise. Note that this procedure does not depend on any $\mathbf{A}_i$, where $i > 0$.

The correctness of the encoding scheme is shown in [1].

C. Word Encoding Optimization

The original conjunction obfuscation design of [1] uses one level for each bit in pattern $\mathbf{v} \in \{0,1,\star\}^L$. Our first design improvement is to utilize a larger input encoding alphabet to reduce the multi-linearity degree of the directed encoding scheme, i.e., use fewer levels than the length of the pattern.

A naïve approach to extend to a larger alphabet would be to convert words of w bits into base-2^w representation and then generate 2^w encoding matrices for each word. This method would work for short elements $r_{i,b}$, where $i \in [\mathcal{L}]$, $b \in \{0,\ldots,2^w-1\}$, and $\mathcal{L} = \lceil L/w \rceil$ is the new effective length of the pattern. However, short elements $s_{i,b}$, which encode the wildcard information, need to be generated and assigned in a more complex manner.

To keep track of bit-level wildcards, we introduce wildcard subpatterns for each word that share the same short element $s_{i,b}$. Specifically, we compute a binary mask for each word that has the wildcard entries set to 1 and all other entries set to 0. Then for every new index $b \in \{0,\ldots,2^w-1\}$ we perform bitwise AND between b and the mask. If the result is 0 (all wildcard bits in the word are set to 0), we generate a new short element $s_{i,b}$. Otherwise, we reuse an existing one. The pseudocode for this optimization is depicted in Algorithm 7 (Appendix D).

To illustrate the effect of this optimization, consider the case of 32-bit conjunctions. The binary alphabet encoding method requires 33 levels of directed encoding. If instead we use 8-bit words, then the number of directed encoding levels reduces to 5. At the same time, the number of encoding matrices per level grows from 4 for $w = 1$ to 512 for $w = 8$, which increases the program size. Hence, there is a tradeoff between a lower multi-linearity degree and the number of encoding matrices, which both affect the obfuscated program size.

IV. Trapdoor Sampling

A. Overview and Motivation

The main computational bottleneck of the obfuscation procedure in the conjunction obfuscation scheme is the preimage sampling **GaussSamp**. Also, the dimensions of the encoding keys and obfuscated program matrices are determined by the dimension of the lattice trapdoor used for preimage sampling. Therefore, any advances in this

area have a profound effect on the performance of conjunction obfuscation and many other program obfuscations schemes.

Our implementation uses a trapdoor sampling approach proposed by Micciancio and Peikert [44] and improved/extended trapdoor sampling algorithms recently proposed in [61]. In this approach, samples around a target point **t** in lattice Λ are generated using an intermediate gadget lattice G^n. The lattice Λ is first mapped to G^n, then a Gaussian sample is generated in G^n. The sample is then mapped back to Λ. The linear function T mapping G^n to Λ is used as the trapdoor. The main challenge of this approach is that the mapping T produces a lattice point in Λ with an ellipsoidal Gaussian distribution and covariance dependent on the transformation T. To generate spherical samples, the authors apply a perturbation technique that adds noise with complimentary covariance to the target point **t** prior to using it as the center for G^n sampling.

From an implementation perspective, this approach decomposes the lattice trapdoor sampling **GaussSamp** procedure into two phases: 1) a perturbation sampling stage (**SampleP**$_Z$), where target-independent perturbation vectors with a covariance matrix defined by the trapdoor mapping T are generated, and 2) a target-dependent stage (**SampleG**) where Gaussian samples are generated from lattice G^n. The first phase, usually referred to as *perturbation generation* [61], can be performed offline as it does not depend on the target point **t**. The second stage, referred to as *G-sampling* [61], is always performed online as it depends on the target point.

The prior Gaussian sampling algorithm introduced in [44] and improved and implemented in [62] has a high computational complexity for an arbitrary modulus (the **SampleG** operation requires $O\left(n \log^3 q\right)$ primitive operations as compared to $O\left(n \log q\right)$ for a power-of-two modulus). Moreover, both variants of the algorithm have high storage requirements for a Cholesky decomposition matrix (computed for each trapdoor pair and used in perturbation sampling) composed of a large number of multiprecision floating-point numbers. The above implies that this prior Gaussian sampling approach is not practical for our implementation of the conjunction obfuscation construction dealing with non-power-of-two moduli and m calls to **SampleG** for each encoding matrix.

We implement a much more efficient approach based on the trapdoor sampling algorithms recently proposed in [61]. The **SampleG** algorithm developed in [61] has $O\left(n \log q\right)$ complexity for arbitrary moduli (same as for power-of-two moduli in [44], [62]). The perturbation sampling method proposed in [61] works with a Cholesky decomposition matrix implicitly and does not require additional storage. Our trapdoor sampling implementation is described in the rest of this section.

B. Trapdoor Construction and G-Lattice Representation

The concrete value of dimension m is determined by the ring trapdoor construction chosen for the implementation. It is common to write $m = \bar{m} + k$, where $\bar{m}$ is a security dimension and k denotes the dimension of (binary) gadget matrix. Two ring constructions were suggested in [44] and further developed in [62]. The first one, where $\bar{m} = 2$ and, therefore, $m = 2 + k$, is generated by drawing k samples $(a, a\hat{r_i} + \hat{e_i})$, where $i \in [k]$, from Ring-LWE distribution. The second construction uses $\bar{m}$ uniformly random polynomials, where $\bar{m}$ is usually set to at least k. As the second construction requires that m be at least $2k$, the Ring-LWE construction deals with a smaller dimension and is thus preferred for our implementation.

Note that a different type of ring trapdoor construction was proposed in [63] based on a non-standard NTRU assumption. This construction cannot be applied to the conjunction obfuscator because the generated samples have a large distribution parameter, i.e., $\Theta\left(\sqrt{q}\right)$, which prevents one from using the samples for multiplying the encodings without invalidating the correctness.

As another major optimization of this work, we introduce a generalized version of the Ring-LWE construction in [62], [44]. In our implementation $m = 2 + \kappa$, where $\kappa = \lceil k / \log_2 t \rceil$ and t is the base for the gadget lattice G^n (t was set to 2 in [62]). The use of base t higher than 2 reduces the dimension of encoding matrices, which dramatically improves all main performance metrics of the conjunction obfuscator, as shown in Section V. The algorithmic idea of using an arbitrary base t was originally suggested in [44] but has not been explored in implementations based on polynomial rings.

The pseudocode for the Ring-LWE trapdoor construction is depicted in Algorithm 1. In the pseudocode, $\hat{\mathbf{r}}$ and $\hat{\mathbf{e}}$ are the row vectors of secret trapdoor polynomials generated using discrete Gaussian distribution, $\mathbf{A}$ is the public key, and $\mathbf{g}^T = \{g_1, g_2, \ldots, g_\kappa\}$ is the gadget row vector corresponding to the gadget lattice G^n. The latter is often denoted as simply G because it is an orthogonal sum of n copies of a low-dimensional lattice G.

Algorithm 1 Trapdoor generation using Ring-LWE for G lattice of base t

function TRAPGEN(1^λ)
  $a \leftarrow \mathcal{U}_q \in \mathcal{R}_q$
  $\hat{\mathbf{r}} := [\hat{r}_1, \ldots, \hat{r}_\kappa] \leftarrow \mathcal{D}_{\mathcal{R},\sigma} \in \mathcal{R}_q^{1\times\kappa}$
  $\hat{\mathbf{e}} := [\hat{e}_1, \ldots, \hat{e}_\kappa] \leftarrow \mathcal{D}_{\mathcal{R},\sigma} \in \mathcal{R}_q^{1\times\kappa}$
  $\mathbf{A} := [a, 1, g_1 - (a\hat{r}_1 + \hat{e}_1), \ldots, g_\kappa - (a\hat{r}_\kappa + \hat{e}_\kappa)] \in \mathcal{R}_q^{1\times(2+\kappa)}$
  return $(\mathbf{A}, (\hat{\mathbf{r}}, \hat{\mathbf{e}}))$
end function

Although the trapdoor $\mathbf{T}$ in the general definition in Section III-B has dimensions $m \times \kappa$, for this construction we can perform all computations with a compact trapdoor $\widetilde{\mathbf{T}} = (\hat{\mathbf{r}}, \hat{\mathbf{e}}) \in {\mathcal{R}_q}^{2\times\kappa}$, as explained in Section IV-E.

C. High-Level Trapdoor Sampling Algorithm

The high-level preimage sampling algorithm adapted for our lattice trapdoor construction is listed in Algorithm 2. It is based on the general approach proposed in [44]. Note that we use the distribution parameter σ_t, which depends on the base t of G-lattice. The vector $\mathbf{p}$ is the perturbation vector required to produce spherical samples.

Algorithm 2 Gaussian preimage sampling

function GAUSSSAMP($\mathbf{A}, \widetilde{\mathbf{T}}, \mathbf{b}, \sigma_t, s$)
  for `i = 0..`$m-1$ **do**
    $\mathbf{p} \leftarrow \mathsf{SampleP}_Z\left(n, q, s, \sigma_t, \widetilde{\mathbf{T}}\right) \in \mathcal{R}_q^m$
    $\mathbf{z} \leftarrow \mathsf{SampleG}(\sigma_t, b_i - \mathbf{A}\mathbf{p}, q)$
    `Convert` $\mathbf{z} \in \mathbb{Z}^{\kappa\times n}$ `to` $\hat{\mathbf{z}} \in \mathcal{R}_q^\kappa$
    $\mathbf{x}_i := [p_1 + \hat{\mathbf{e}}\hat{\mathbf{z}}, p_2 + \hat{\mathbf{r}}\hat{\mathbf{z}}, p_3 + \hat{z}_1, \ldots, p_m + \hat{z}_\kappa]$
  end for
  return $\mathbf{x} \in {\mathcal{R}_q}^{m\times m}$
end function

Sections IV-D and IV-E describe in more detail the procedures **SampleG** and **SampleP**$_Z$, respectively.

D. Sampling G-lattices

The G-lattice sampling problem is defined as the problem of sampling the discrete Gaussian distribution on a lattice coset $\Lambda_v^\perp\left(\mathbf{g}^T\right) = \left\{\mathbf{z} \in \mathbb{Z}^\kappa : \mathbf{g}^T\mathbf{z} = v \bmod q\right\}$, where $q \leq t^\kappa, v \in \mathbb{Z}$ and $\mathbf{g} = \left(1, t, t^2, \ldots, t^{\kappa-1}\right)$. The G-sampling problem is formulated here for a single integer v rather than a n-dimensional lattice because each of the n integers can be sampled in parallel. Our implementation of G-sampling works with a n-dimensional lattice.

We implement a variation of the G-sampling algorithm developed in [61], which supports arbitrary bases for G-lattice. Our variation (depicted in Algorithm 3 of Appendix A) relies on continuous Gaussian sampling in the internal perturbation sampling step (in contrast to discrete Gaussian sampling in Figure 2 of [61]), reduces the number of calls to polynomial CRT operations, and increases opportunities for parallel execution.

Algorithm 3 has complexity $O\left(n \log q\right)$ for an arbitrary modulus. The main idea of the algorithm is not to sample $\Lambda_v^\perp\left(\mathbf{g}^T\right)$ directly, but to express the lattice basis $\mathbf{B}_q = \mathbf{T}\mathbf{D}$ as the image (using a transformation $\mathbf{T}$) of a matrix $\mathbf{D}$ with a sparse, triangular structure. This technique requires adding a perturbation with a complementary covariance to obtain a spherical Gaussian distribution, as in the case of the **GaussSamp** procedure described in Algorithm 2. In this prior work the authors select an appropriate instantiation of $\mathbf{D}$ that is sparse and triangular, and has a complementary covariance matrix with simple Cholesky decomposition $\Sigma_2 = \mathbf{L} \cdot \mathbf{L}^T$, where $\mathbf{L}$ is an upper triangular matrix, and find the entries of the $\mathbf{L}$ matrix in closed form.

E. Perturbation Sampling

The lattice preimage sampling algorithm developed in [44] requires the generation of nm-dimensional Gaussian perturbation vectors $\mathbf{p}$ with covariance $\Sigma_p := s^2 \cdot \mathbf{I} - \sigma_t^2 \big[\mathbf{T}\,\mathbf{I}\big]^T \cdot \big[\mathbf{T}^T\,\mathbf{I}\big]$, where $\mathbf{T} \in \mathbb{Z}^{2n \times n\kappa}$ is a matrix with small entries serving as a lattice trapdoor, s is the upper bound on the spectral norm of $\sigma_t \big[\mathbf{T}^T, \mathbf{I}\big]^T$.

When working with algebraic lattices, the trapdoor $\mathbf{T}$ can be compactly represented by a matrix $\widetilde{\mathbf{T}} \in R_n^{2\times\kappa}$, where n denotes the rank (dimension) of the ring R_n. In our case, this corresponds to the cyclotomic ring of order $\hat{m} = 2n$. For the Ring-LWE trapdoor construction used in our implementation (Algorithm 1), the trapdoor $\widetilde{\mathbf{T}}$ is computed as $(\hat{\mathbf{r}}, \hat{\mathbf{e}})$. The main challenge with the perturbation sampling techniques developed in [62], [44] is the direct computation of a Cholesky decomposition of Σ_p that destroys the ring structure of the compact trapdoor and operates on matrices over $\mathbb{R}$.

Genise and Micciancio [61] propose an algorithm that leverages the ring structure of R_n and performs all computations either in cyclotomic rings or fields over $\Phi_{2n}(x) = x^n + 1$. The algorithm does not require any preprocessing/storage and runs with time and space complexity quasi-linear in n. The perturbation sampling algorithm can be summarized in a modular way as a combination of three steps [61]:

1) The problem of sampling a $n(2+\kappa)$-dimensional Gaussian perturbation vector with covariance Σ_p is reduced to the problem of sampling a $2n$-dimensional integer vector with covariance expressed by a 2×2 matrix over R_n.
2) The problem of sampling with covariance in $R_n^{2\times 2}$ is reduced to sampling two n-dimensional vectors with covariance in R_n.
3) The sampling problem with covariance in R_n is reduced to sampling n-dimensional perturbation with covariance expressed by a 2×2 matrix over the smaller ring $R_{n/2}$ using an FFT-like approach.

We implement a variation of the perturbation generation algorithm developed in [61]. Our variation (depicted in Algorithm 4 of Appendix A) reduces the number of calls to CRT operations and increases opportunities for parallel execution.

F. Integer Gaussian Sampling

Our implementations of G-sampling and perturbation sampling procedures require generating integers with Gaussian distribution for large distribution parameters and varying centers. For instance, the optimal values of base t lead to distribution parameters up to 2^{20} for G-sampling and even larger values for perturbation generation. This implies that conventional Gaussian sampling techniques such as the inversion sampling developed in [64] and rejection sampling proposed in section 4.1 of [65] are not practical for trapdoor sampling, as described in detail in [46].

To this end, we implement two recently proposed generic samplers: Karney's rejection sampler [45] and constant-time sampler [46].

The rejection sampler [45] provides a relatively low rejection rate (roughly 0.5) vs. a much higher rate in the case of rejection sampling [65], and has no additional storage requirements, at least when it is not separated into offline and online stages. However, it has a relatively significant variability in sampling time making it prone to timing attacks.

The generic sampler [46], on the other hand, uses a constant-time algorithm that breaks down sampling for large distribution parameters into multiple runs for much smaller distribution parameters. It also utilizes multiple cosets to support varying-center requirements, with the number of cosets being an adjustable parameter. At the lowest level, this generic sampler depends on the implementation of a base sampler for a small distribution parameter and fixed center, which can be realized using efficient Cumulative Distribution Function (CDF) inversion [64] or Knuth-Yao [66] methods. This algorithm has significant memory requirements to store precomputed lookup tables/trees for the base sampler but the storage requirements can be adjusted at the expense of increased sampling runtime.

The choice of a specific generic sampler in our experiments is determined by minimizing the obfuscation runtime.

V. Setting the Parameters

A. Distribution Parameters

1) Distribution Parameter for Ring-LWE Trapdoor Construction: The trapdoor secret polynomials are generated using the smoothing parameter σ estimated as $\sigma \approx \sqrt{\ln(2n_m/\epsilon)/\pi}$, where n_m is the maximum ring dimension and ϵ is the bound on the statistical error introduced by each randomized-rounding operation [44]. For $n_m \leq 2^{14}$ and $\epsilon \leq 2^{-80}$, we choose a value of $\sigma \approx 4.578$.

2) Short Ring Elements in Directed Encoding: For short ring elements $s_{i,b}, r_{i,b}$, we use ternary uniformly random ring elements, which are sampled over $\{-1, 0, 1\}^n$. This implies that we rely on small-secret Ring-LWE for directed encoding.

3) Distribution Parameters for Directed Encoding: To encode ternary random elements, we use the smoothing parameter σ (for the noise polynomials) defined in Section V-A1.

To encode a product of ternary random ring elements under the Ring-LWE assumption, we need to sample noise polynomials using $\sigma' = \omega(\log \lambda)\sqrt{n}\sigma$ (Section 4.3 of [1]). The term $\omega(\log \lambda)$ guarantees that $\mathcal{D}_{\mathcal{R},\sqrt{n}\sigma+\sigma'}$ is "smudged" by Lemma 2.4 of [1] to $\mathcal{D}_{\mathcal{R},\sigma'}$. In our implementation, we use a concrete estimate $\sigma' = k\sqrt{n}\sigma$.

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

5) Spectral norm s: Parameter s is the spectral norm used in computing the Cholesky decomposition matrix (it guarantees that the perturbation covariance matrix is well-defined). To bound s, we use inequality $s > s_1(\mathbf{X})\sigma_t$, where $\mathbf{X}$ is a sub-Gaussian random matrix with parameter σ [44].

Lemma 2.9 of [44] 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 s as $s > C_0\sigma\sigma_t\left(\sqrt{n\kappa} + \sqrt{2n} + 4.7\right)$. In our experiments we used $C_0 = 1.3$, which was found empirically.

B. Conjunction Obfuscator Correctness

The correctness constraint for a conjunction pattern with $\mathcal{L}$ words ($\mathcal{L} \geq 2$) is expressed as

$$q > 192\sigma'\left(2s\sqrt{mn}\right)^{\mathcal{L}}. \quad (1)$$

The correctness constraint (1), which is derived using the Central Limit Theorem, significantly reduces bitwidth requirements for modulus q (as compared to the analysis in [1] for a multi-level directed encoding scheme; note also that no correctness constraint for conjunction obfuscator was derived in [1]). Hence, our correctness estimate is another major improvement in this work. The details of deriving the correctness constraint are provided in Appendix B.

C. Security

1) Ring Dimension n: We utilize Ring-LWE for the trapdoor construction and a combination of small-secret Ring-LWE and an entropic variant of small-secret Ring-LWE for directed encoding. Since entropic small-secret Ring-LWE is the strongest assumption, it should determine the value of the ring dimension n. However, no experimental results for entropic Ring-LWE are available and hence we assume that we can use the same lower bounds for λ as for regular (non-entropic) Ring-LWE. As the directed encoding Ring-LWE instance uses the ternary distribution $\mathcal{T}$ to generate secret polynomials, our lower-bound estimates of the number of security bits λ are computed for this variant of Ring-LWE.

We run the LWE security estimator[2] (commit 9302d42) [67] to find the lowest security levels for the uSVP, decoding, and dual attacks following the standard homomorphic encryption security recommendations [68]. 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.

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

2) Dimension m: The dimension m can be written as $\bar{m} + \kappa$, where $\bar{m}$ is a security dimension determined by the Ring-LWE trapdoor construction and κ is a functional parameter.

Consider the Ring-LWE construction constraint. Let us write the public key $\mathbf{A}$ in Algorithm 1 as $\mathbf{A} = \left[\bar{\mathbf{A}}|\mathbf{g}^T - \bar{\mathbf{A}}\mathbf{R}\right]$, where $\mathbf{A} \times \left(\frac{\mathbf{R}}{\mathbf{I}}\right) = \mathbf{g}^T$. Here, $\bar{\mathbf{A}}$ is uniformly random and $\mathbf{R}$ is small. The pseudorandomness of $\mathbf{A} = \left[\bar{\mathbf{A}}|\mathbf{g}^T - \bar{\mathbf{A}}\mathbf{R}\right]$ (required by our application) immediately follows from the pseudorandomness of $\left[\bar{\mathbf{A}}|\bar{\mathbf{A}}\mathbf{R}\right]$, which is implied by the Ring-LWE assumption.

More specifically, we use the Ring-LWE construction from [44], [62], implying that $\bar{\mathbf{A}}$ is represented as $[a, 1]$, i.e., a 1×2 matrix over the Ring-LWE ring. Then each column of $\bar{\mathbf{A}}\mathbf{R}$ is of the form $c_i = a\hat{r}_i + \hat{e}_i$. The pseudorandomness of (a, c_i) follows from Ring-LWE. Since each c_i uses a different "secret" r_i, the public value of a can be reused, and joint pseudorandomness follows by a standard hybrid argument. This means that the security dimension $\bar{m} = 2$, i.e., $m = 2 + \kappa$, can be used regardless of dimension κ.

3) Work Factors t_{VBB} and t_{RLWE}: We consider two attack models to learn the full conjunction pattern. The first one is based on VBB (input-output) analysis, independently of the underlying cryptographic obfuscation construction. The second model is based on lattice attacks on the obfuscated program, i.e., requires solving multiple Ring-LWE problems. We present here the attacks that result in lowest work factors t_{VBB} and t_{RLWE} for the VBB and lattice models, respectively (our analysis showed these attacks correspond to optimal lower bounds for both models but the formal proofs are beyond the scope of this paper). The work factors are expressed in terms of the number of clock cycles to abstract from specific hardware architectures.

VBB model. The work factor for a specific conjunction pattern depends on the number of wildcard bits, denoted as X. If we assume that the conjunction pattern has high entropy, then the adversary can (on average) run 2^{L-X} random-input evaluation queries to find a first match, which is equivalent to a point function with $L - X$ bits. Once the first match is found, the adversary can run additional L evaluation queries to find the wildcard positions. Therefore, the VBB work factor can be estimated as $t_{VBB} = \left(2^{L-X} + L\right) t_{eval} \approx 2^{L-X} t_{eval}$, where t_{eval} is the number of clock cycles for a single evaluation of the obfuscated program.

Lattice model. Our most efficient lattice-based attack can be described as follows:

1) Find all wildcard positions. For each word of w bits, solve $w \times 2 \times 2 = 4w$ Ring-LWE problems, where the first factor of 2 corresponds to bits 0 and 1, and the second factor of 2 corresponds to each encoding matrix for $s_{i,b} \cdot r_{i,b}$ and $r_{i,b}$, respectively (both ring elements are needed to find $s_{i,b}$). This procedure is repeated for each word, i.e., L/w times (for simplicity, we assume $w|L$), which implies this step requires a work factor of $2^{\lambda+2} L$.
2) For each word, solve additional $2 \times 2^{w-X_i}$ problems to find all $s_{i,b}$, where X_i is the number of wildcards in the i-th word. Some solutions may have already been found in Step 1 but we can expect their contribution to be small (to simplify the analysis). The work factor of this step is $2^{\lambda+1} \sum_{i=1}^{L/w} 2^{w-X_i}$.
3) Solve Ring-LWE problems for r_{L+1} and $r_{L+1} \prod_{i=1}^{L} s_{i,v_i}$.
4) Steps 1–2 yield all values of $s_{i,b}$. Now compare 2^{L-X} products $\prod_{i=1}^{L} s_{i,b}$ with $\prod_{i=1}^{L} s_{i,v_i}$ found in step 3. The computation of one product $\prod_{i=1}^{L} s_{i,b}$ requires approximately the work factor of $\left(2m^2\right)^{-1} t_{eval}$, as can be seen from Algorithm 8.

The Ring-LWE work factor can then be written as

$$t_{RLWE} \approx 2^{\lambda+1} \left(\sum_{i=1}^{L/w} 2^{w-X_i} + 2L \right) + \frac{2^{L-X} t_{eval}}{2m^2}.$$

For conjunction obfuscation with at least 80 bits of security, the following constraints have to be satisfied: $t_{RLWE} \geq t_{VBB} \geq 2^{80}$. Note that for a 2.5 GHz core, 2^{80} clock cycles correspond to 1.4×10^7 core-years.

4) Small-Secret Ring-LWE vs Error-Secret Ring-LWE for Directed Encoding: Our implementation also supports integer Gaussian distribution $\mathcal{D}_{\mathcal{R},\sigma}$ for short ring elements $s_{i,b}, r_{i,b}$, i.e., the error-secret Ring-LWE (Definition 1). This variant increases the modulus q, more specifically the parameter B_e in expression (5), by a factor of $\sigma\sqrt{\gamma}$ (γ is explained in Section V-B), which is only 5 bits for our parameters.

According to our estimates using [67], error-secret and small-secret Ring-LWE require almost the same bitwidth for q to achieve the same level of security for practical ring dimensions (the modulus q is at most 4 bits larger for small-secret Ring-LWE). Hence, both small-secret and error-secret Ring-LWE variants can be used without any major difference in program size or runtimes (none of the performance metrics increase by more than 15% for

TABLE I: Program size as a function of word size for 32-bit conjunctions (with $\lambda > 80$ bits and $t = 2$)

w	$\mathcal{L}$	k	n	$\Sigma_{\text{theor}}(\Pi_{\mathbf{v}})$, Terabytes
1	32	1041	32768	617
2	16	505	16384	36
4	8	248	8192	5
8	4	127	4096	3
16	2	70	2048	42
32	1	45	2048	294,900

TABLE II: Program size as a function of word size for 64-bit conjunctions (with $\lambda > 80$ bits and $t = 2$)

w	$\mathcal{L}$	k	n	$\Sigma_{\text{theor}}(\Pi_{\mathbf{v}})$, Terabytes
1	64	2204	65536	22,200
2	32	1049	32768	1,230
4	16	505	16384	142
8	8	248	8192	77
16	4	127	4096	792
32	2	70	2048	2,730,000

the error-secret case according to our experimental analysis), achieving approximately the same level of security according to LWE estimator [67].

We choose the small-secret Ring-LWE case for our main experiments because it is slightly more efficient than the error-secret Ring-LWE scenario and is currently believed to be as secure against known attacks.

D. Word Size w

The selection of word size w is governed by the tradeoff between the decrease in multi-linearity degree $(\mathcal{L}+1)$ and increase in the number of encoding matrices.

To find the optimal value of w, we introduce a formal definition of theoretical program size Σ_{theor} (in bytes):

$$\Sigma_{\text{theor}}(\Pi_{\mathbf{v}}) = \frac{1}{4}\left(2^w \cdot \mathcal{L} + 1\right)(2+\kappa)^2\, nk. \tag{2}$$

The first multiplicand accounts for the number of encoding matrices, the second multiplicand represents the number of ring elements per encoding matrix, and the last term nk deals with the storage for each ring element. This theoretical program size is generally slightly smaller than the actual storage consumed in an implementation (due to storage overhead related to the size of underlying native integers and extra data members in C++ classes).

We consider the program size as the main practical limitation of conjunction obfuscator due to the high size estimates (in Terabytes) listed in Tables I and II, which are found for the G-lattice base t of 2 (larger bases are discussed in Section V-E) and all other parameters computed using the input parameters and constraints described in Sections V-A–V-C. These estimates imply that $w = 4$ and $w = 8$ produce the smallest program sizes.

In addition to obfuscated program size, we should consider the evaluation runtime as another optimization constraint. The evaluation runtime is proportional to $\mathcal{L}(2+\kappa)^2\, nk$, which implies that smaller $\mathcal{L}$, n, and k reduce the runtime. Therefore, the case of $w = 8$ is optimal for our experiments when the combined effect of obfuscated program size and evaluation runtime is considered.

Tables I and II suggest that the use of $w = 8$ instead of $w = 1$ reduces the program size by more than 2 and 3 orders of magnitude for 32-bit and 64-bit conjunction patterns, respectively. The proportionality of evaluation runtime to $\mathcal{L}(2+\kappa)^2\, nk$ suggests that the runtime is reduced by about 4 orders of magnitude when switching from $w = 1$ to $w = 8$ both for 32-bit and 64-bit conjunction programs.

E. G-Lattice Base t

Larger values of G-lattice base t decrease the dimension of public key $\mathbf{A}_0$, encoding secret keys $\widetilde{\mathbf{T}}_i$, and encoding matrices $\mathbf{R}_i$, where i corresponds to the level of directed encoding. More concretely, the sizes of $\mathbf{A}_0$, $\widetilde{\mathbf{T}}_i$, and $\mathbf{R}_i$

are proportional to $(2+\kappa)\,nk$, κnk, and $(2+\kappa)^2\,nk$, respectively. Here, $\kappa = \lceil k/\log_2 t \rceil$ and k is the number of bits in modulus q. The program size, obfuscation time, and evaluation time are determined by the size of $\mathbf{R}_i$.

When t is increased, the term $(2+\kappa)^2$ in the size of $\mathbf{R}_i$ becomes smaller but the modulus bitwidth k and ring dimension n grow as follows from expression (1) and security analysis for n (Section V-C1). The correctness constraint (1) suggests that q is proportional to $(t+1)^{\mathcal{L}}$, which means that k grows linearly with $\log_2 t$. This implies that the size of $\mathbf{R}_i$, and hence the obfuscation program size, is always reduced with increase in t. The maximum practical value of t is reached when one of the following conditions is met:

1) Evaluation runtime becomes inadequately slow (as it is proportional to kn);
2) Implementation limitations of integer Gaussian sampling are reached, for instance, the samples start exceeding the bitwidth of a native integer data type;
3) The value of κ reaches 2 ($m = 4$), which is the smallest value supported by our perturbation sampling procedure.

It should be pointed out that the choice of t also depends on the value of the most significant digit of modulus q with respect to base t, which affects the value of $d_{\kappa-1}$ in Algorithm 3. For the worst-case analysis, assume that $q_{\kappa-1} = 1$, then $d_{\kappa-1} \approx 1/t$. Once this value is substituted into **SampleD**, $z_{\kappa-1}$ is sampled using a distribution parameter $\approx \sigma t$. Then the term $q_0 z_{\kappa-1}$ in the expression for t_0 in **SampleG** may reach values that are proportional to σt^2, which are much higher than one would expect, i.e., comparable to σt.

To avoid this scenario, we introduce an additional constraint $q_{\kappa-1}/t > 1/\zeta$, where ζ is a constant. In our experiments, we set $\zeta = 2$, which implies $q_{\kappa-1}$ has at most one bit less than t.

We also performed a combined optimization analysis for word size w and G-lattice base t, which confirmed that $w = 8$ is still the optimal value for $t > 2$.

We use the highest value of $t = 2^{20}$ in our experiments due to the limitations of our implementation of Gaussian sampling, which operates with native C++ unsigned integers, and selected bitwidth of prime moduli in the Double-CRT representation. If these constraints are removed, higher values of G-lattice base t can be used.

VI. Efficient Matrix and Polynomial Arithmetic

A. Matrix Chain Product in the Evaluation

The matrix chain multiplication in the evaluation operation involves multiplications of encoding matrices of $m \times m$ by each other, which requires a running time of $O\left(m^3 n\right)$ for the naive implementation or $O\left(m^{\log_2 7} n\right)$ in the case of Strassen's algorithm. At the same time, the product of encoding matrices is multiplied at the end by a row vector $\mathbf{A}_0 \in \mathcal{R}_q{}^{1\times m}$. This suggests that by changing the order of multiplications, we can transform this matrix chain multiplication into a row-vector-by-matrix chain product. Each row-vector-by-matrix product has a running time of $O\left(m^2 n\right)$ and can provide a running time improvement by a factor of m, as compared to the naive implementation of matrix product. This optimization is included in Algorithm 8 listed in Appendix D. A similar idea was used in [25], [26].

B. Efficient Polynomial Arithmetic

1) Double-CRT Operations: All polynomial multiplications are performed in the Double-CRT representation. We use the bitwidth of 60 for each prime modulus (64-bit native unsigned integers are leveraged for storing the numbers). This implies a product of two polynomials with ring dimension n and modulus q (bitwidth k) requires $n\lceil k/60 \rceil$ multiplications of 64-bit native integers, i.e., scales almost linearly with increase in k. Hence, multiplications of polynomials with large k, for example, 1000 bits, can be supported without involving multiprecision arithmetic.

There are certain operations where we have to switch from Double-CRT representation to a polynomial of multiprecision integers with a large modulus q. This requires transforming all small-modulus polynomials to the coefficient representation and then performing the CRT interpolation to get large (multiprecision) coefficients of the polynomial with respect to modulus q. This procedure is computationally expensive and involves $\lceil k/60 \rceil$ NTTs followed by the CRT interpolation with modulo reductions for every coefficient with respect to q. The two operations requiring CRT Interpolation are (1) G-sampling where the digits of the large coefficients are extracted and (2) infinity norm computation at the last stage of evaluation.

2) Number Theoretic Transform: The multiplication of elements in cyclotomic rings $\mathcal{R}_{p_i}$ is performed using the Chinese Remainder Transform (CRT) [57]. We use an implementation of Fermat Theoretic Transform (FTT) described in [58]. We implement FTT with Number Theoretic Transform (NTT) as a subroutine. For NTT, we use the iterative Cooley-Tukey algorithm with optimized butterfly operations, which is implemented in PALISADE.

3) Cyclotomic Fields: For multiplications in $\mathcal{K}_{2n}$ we use the iterative Cooley-Tukey FTT algorithm over complex primitive roots of unity.

To convert elements of rings to fields, we switch the polynomials from the evaluation representation to the coefficient one as an intermediate step because the CRTs for rings operate with modular primitive roots of unity and CRTs for fields deal with complex primitive roots of unity.

4) Polynomial Transposition: Element transposition for a polynomial $f(x) = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$ over cyclotomic polynomial $x^n + 1$ is expressed as $f^t(x) = f_0 - f_{n-1}x - \cdots - f_1 x^{n-1}$. This transposition technique was used for both rings and fields. In our implementation the transposition operation is performed directly in evaluation representation by applying an automorphism from $f(\zeta_{2n})$ to $f(\zeta_{2n}^{2n-1})$.

5) Modular Arithmetic: For modular reduction of multiprecision integers (in CRT interpolation), we use a generalized Barrett modulo reduction algorithm [69]. This approach requires one pre-computation per NTT run and converts modulo reduction to roughly two multiplications.

VII. Implementation Details

A. Pseudocode of Obfuscation Scheme Algorithms

We provide pseudocode for key generation, encoding, obfuscation, and evaluation of the scheme in Appendix D. The pseudocode matches our implementation in C++.

B. Integer Sampling

Both conjunction obfuscation and trapdoor sampling algorithms call the integer sampling subroutine **SampleZ**(σ, c) that returns a sample statistically close to $\mathcal{D}_{\mathbb{Z},c,\sigma}$. When the center c does not change and distribution parameter is small (as in directed encoding or Ring-LWE trapdoor construction), our **SampleZ** implementation uses the inversion sampling method developed in [64]. In all other cases (trapdoor sampling), we use either Karney's rejection sampler [45] or constant-time sampler [46].

A bottleneck of integer sampling operations in lattice-based cryptography, specifically those called in the subroutines of **GaussSamp**, is the use of multiprecision floating-point numbers where the number of bits in the mantissa should roughly match the number of security bits supported by the cryptographic protocol. A recent theoretical result in [46] suggests that both the G-sampling and perturbation generation algorithms used in our implementation can support at least 100 bits of security using double-precision floating point arithmetic. More specifically, Lemma 3.2 in [46] states that $\lambda/2$ significant bits in a floating-point number is sufficient for λ bits of security. This result also applies to joint (possibly dependent) distributions, as in Lemma 4.3 of [46]. Because we are not attempting to exceed 100 bits of security, the significand precision of 53 bits provided by IEEE 754 double-precision floating numbers is sufficient for our security target. Therefore, our implementation of integer Gaussian sampling performs computations on double-precision floating-point numbers.

C. Software Implementation

We implement the conjunction obfuscation scheme in PALISADE, 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), Fermat-Theoretic Transform (FTT), and Bluestein FFT. The integer distribution samplers are implemented in this layer.

Our conjunction obfuscation implementation is a new PALISADE module called "trapdoor", which includes the following new features broken down by layer:

- Conjunction obfuscation scheme in the cryptographic layer.
- Directed encoding in the encoding layer.
- Trapdoor sampling, including Ring-LWE trapdoor generation, G-sampling and perturbation generation routines in the lattice layer. Cyclotomic fields $\mathcal{K}_{2n}$ and additional polynomial/double-CRT operations, such as polynomial transposition, are also in this layer.
- Generic integer Gaussian samplers and a Cooley-Tukey transform based on complex roots of unity in the arithmetic layer.

Several lattice-layer and arithmetic-layer optimizations are also applied for runtimes improvements.

D. Loop Parallelization

Multi-threading is performed using OpenMP[3]. Loop parallelization is applied to parallelize obfuscation, lattice, and matrix operations, and we use the following loop parallelization optimizations:

1) In **KeyGen** (Algorithm 5), the loop calling **TrapGen** is parallelized, with its results combined in an ordered way into an STL vector.
2) In **GaussSamp** (Algorithm 2), the main loop is executed in parallel. The loop is called by **Encode**, which is called by **Obfuscate**. This optimization effectively achieves the overall parallel execution of the obfuscation procedure.
3) The loops in matrix and matrix-vector multiplication are parallelized. This optimization determines the parallelization of **Evaluate** (Algorithm 8).
4) Number-theoretic transforms of matrices (vectors) of ring elements are executed in parallel for each ring element. This optimization applies to key generation, obfuscation, and evaluation operations.
5) CRT Interpolation used in G-sampling (**Obfuscate**) and norm computation (**Evaluate**) is executed in parallel for each coefficient of the polynomial.

We discuss the effect of these optimizations in Sec. VIII-E.

VIII. Experimental results

A. Testbed

Experiments were performed using a server computing environment with 4 sockets of Intel Xeon CPU E7-8867 v3 rated at 2.50GHz, each with 16 cores. The total number of cores was 64 (128 logical processors). 2TB of RAM was accessible for the experiment. The executable was run using a docker image with Linux Ubuntu 16.04 LTS. The evaluation environment for parallelization experiments was a commodity desktop computer with an Intel Core i7-3770 CPU with 4 cores (8 logical processors) rated at 3.40GHz and 16GB of memory, running Linux CentOS 7. In all of our obfuscation experiments, we selected the minimum modulus bitwidth k that satisfies the correctness constraint (1) for a ring dimension n corresponding to the chosen security level.

B. Integer Gaussian Sampling Experiments

We experimentally compared the runtimes of Karney's rejection method [45] with the generic sampler [46] using the CDF inversion [64] method as the base sampler. The results are in Appendix C. Based on this analysis, we selected Karney's method for our main conjunction obfuscation experiments.

[3] http://www.openmp.org/

TABLE III: Runtimes and program size for 32-bit conjunction programs in a server computing environment for w=8

n	k	$\log_2 t$	λ_{VBB}/ λ_{RLWE}	$\Sigma_{\text{exp}}(\Pi_{\mathbf{v}})$ (GB)	**KeyGen** (ms)	**Obfuscate** (min)	**Evaluate** (ms)
1024	180	20	53/54	5.85	94	6.2	32
2048	180	15	54/56	16.4	411	17.3	60
4096	180	15	55/86	37.9	1141	36.0	117

TABLE IV: Runtimes and program size for 64-bit conjunction programs in a server computing environment for w=8

n	k	$\log_2 t$	λ_{VBB}/ λ_{RLWE}	$\Sigma_{\text{exp}}(\Pi_{\mathbf{v}})$ (GB)	**KeyGen** (s)	**Obfuscate** (hr)	**Evaluate** (s)
1024	360	20	70/60	77	0.31	0.7	0.29
2048	360	20	71/61	155	0.66	1.4	0.53
4096	360	18	72/62	374	1.58	3.3	1.06
8192	360	18	73/87	748	3.03	6.7	2.45

C. Experiments for the Word Size of One Byte

Tables III and IV show results for the word size w of 8 bits in the server computing environment for 32-bit and 64-bit conjunction programs, respectively. $\Sigma_{\text{exp}}(\Pi_{\mathbf{v}})$ is the actual program size (experimentally measured as the RAM amount used by the process after the obfuscation program is generated). These experiments were run in the multi-threaded mode with 16 and 32 threads for 32-bit and 64-bit conjunctions, respectively.

Tables III and IV also list the work factors (in bits of security) for the VBB and lattice attacks, which are computed as $\lambda_{VBB} = \log_2(t_{VBB})$ and $\lambda_{RLWE} = \log_2(t_{RLWE})$, respectively. For 32-bit and 64-bit conjunctions, the number of wildcard bits was set to 8 and 16, respectively. The wildcard bits were assumed to be uniformly distributed over the words of the pattern (2 wildcard bits per byte).

Table III suggests that lattice security parameters for 32-bit conjunctions are sufficient to match the VBB security, but the VBB work factor for the case of $n = 1024$ is only 2^{53} clock cycles, which corresponds to 39 core-days for a 2.5 GHz core. Table IV shows that the lattice attack work factor starts exceeding the VBB work factor for 64-bit conjunctions at $n = 8192$, when the VBB work factor is 2^{73} clock cycles, i.e., 1.1×10^5 core-years.

Note that our implementation is based on the entropic Ring-LWE problem with a small-secret (ternary) distribution, which is a stronger assumption than Ring-LWE. While our work factor estimates already incorporate the effect of small-secret distribution (using the LWE estimator [67]), the effect of the entropic variant of Ring-LWE on the work factor is currently unknown and is thus ignored in our estimates.

As suggested in Section V-D, program size is a major practical limitation of conjunction obfuscator. For a 64-bit conjunction program, the experimental program size reached 750 GB. However, the program size for a 32-bit program is small enough to be loaded into the RAM of a commodity desktop computer.

The experimental results in Tables III and IV also demonstrate that the key generation time is small, on the order of one second.

The obfuscation takes 6.7 hours to achieve 73-bit security for the 64-bit conjunction program, and is the main computational bottleneck of conjunction obfuscator. This operation is run offline and once per program. Thus obfuscation time is does not impact many practical settings.

Evaluation takes 32 ms to acheive 53 bits of security for a 32-bit pattern and 2.5 seconds to attain 73-bit security for a 64-bit conjunction pattern. The evaluation time is the main online operation and is expected to be run frequently. The 32-bit pattern results imply that runtime is practical. Our evaluation runtime for a 64-bit conjunction obfuscator is smaller by more than two orders of magnitude than the time (949 sec.) reported for a 64-bit read-once branching program obfuscated using GGH15 in [27].

D. Experiments for the Word Size of One Bit

To explore the effect of multilinearity degree on the runtime metrics of conjunction obfuscator, we performed a series of experiments at $w = 1$ (Table V). The multinearity degree of directed encoding corresponds to $L + 1$ as

TABLE V: Runtimes and program size for conjunction programs at w=1, $t = 2^{20}$, and $\lambda > 80$

L	n	k	$\Sigma_{\text{exp}}(\Pi_{\mathbf{v}})$ (GB)	**KeyGen** (s)	**Obfuscate** (min)	**Evaluate** (s)
5	8192	240	1.08	1.1	1.1	0.39
6	8192	300	2.36	1.7	1.8	0.72
8	16384	420	13.2	7.6	8.2	3.7
10	16384	480	28.6	11	12	5.5
12	16384	600	60.4	18	22	12
14	32768	720	227	62	103	74
16	32768	780	363	81	135	101
18	32768	900	565	115	198	158
19	32768	960	723	134	237	188
20	32768	960	825	148	252	213
21	32768	1020	994	172	310	230
22	32768	1080	1232	199	350	247
23	32768	1140	1459	212	404	286
24	32768	1200	1774	257	510	379

TABLE VI: Runtimes for 32-bit conjunction patterns at $n = 1024$ as a function of number of threads in a 4-core commodity desktop computing environment

# threads	**KeyGen** (s)	**Obfuscate** (min)	**Evaluate** (ms)
1	0.179	24.3	161
2	0.106	13.8	90
4	0.056	7.7	48
8	0.053	7.0	43

we have one more level of encoding at the end, which is specific to the test for conjunction obfuscator.

Table V shows that our implementation is able to achieve the multilinearity degree of 25 (in contrast to 20 in [27] for a comparable computing environment). For the degree of 20, our obfuscation time is 237 minutes (vs 4,060 minutes in [27]) and our evaluation is 188 seconds (vs 1,514 seconds in [27]).

Our main experimental limitation was memory in the server computing environment. Results in Table V show that our implementation would be able to support at most 24-bit conjunction programs if the word encoding optimization were not applied. Also, the runtimes for this 24-bit scenario are substantially higher than our results for 32-bit conjunction programs in Table III.

E. Parallelization Experiments

Table VI shows the runtime results for a 32-bit pattern with 53 bits of security on a 4-core desktop commodity CPU as a function of the number of threads. The total program size and all input parameters are the same as in the first row of Table III. As expected, runtimes for 4 and 8 threads are approximately the same. There is no major benefit of hyper-threading, as the number of physical cores is 4.

When increasing the number of threads from 1 to 4, the key generation time decreases by a factor of 3.2, suggesting that key generation benefits from loop parallelization. The obfuscation procedure scales well with more threads. Runtime improvement is a factor of 3.2 (and even 3.5 when 8 threads are considered). This implies that further obfuscation runtime improvements can be achieved using more CPU cores. The evaluation procedure also benefits from loop parallelization. The runtime improvement in this case is 3.4 (3.7 for 8 threads).

The evaluation runtime of 43 ms on a commodity desktop environment implies that a 32-bit conjunction obfuscator is already practical.

We also ran the evaluation of an obfuscated 64-bit conjunction program (with 73 bits of security) on the commodity desktop computer for the scenario where the obfuscation is previously performed in a high-performance computing environment (corresponds to the last row in Table IV). This fits the scenario where the obfuscated program would be stored on SSD drives (or other fast access media.) The average time of evaluation was 3.5 sec.

IX. Concluding Remarks

Our work presents an improved design and software implementation for the secure obfuscation of conjunction programs, which are significantly more complex than simple point obfuscation functions supported by prior obfuscation implementations. The obfuscation construction we implement is based on a reasonable hardness variant of a standard lattice assumption (entropic Ring-LWE) and distributional VBB, in constrast to previous implementations of non-trivial obfuscators based on IO via multilinear maps [25], [26], [27] or the heuristic techniques not derived from the computational hardness of mathematical problems [2], [3], [4], [5], [6], [7].

Through our optimizations, we are able to reduce the program size, obfuscation runtime, and evaluation runtime by multiple orders of magnitude. This allows us to execute the obfuscation and evaluation of 32-bit conjunction programs in a commodity desktop environment. Our implementation can also run secure obfuscation of 64-bit conjunction programs in a commercially available server computing environment and execute evaluation in a commodity desktop environment, achieving the evaluation runtime of 3.5 seconds.

A major challenge not addressed by this work is the encoding of real practical programs as conjunctions chosen from a distribution having sufficient entropy. A potential approach to this problem is to use the obfuscation technique for compute-and-compare programs, a recently proposed generalization of conjunction obfuscators, based on LWE [50]. Note that many design elements and optimizations presented in this study can also be applied to this more general obfuscation technique.

X. Acknowledgements

We gratefully acknowledge the input and feedback from Sean Al-Gattas, David Archer, Lisa Bahler, Brian Coan, Nicholas Genise, Shafi Goldwasser, Michael Hsieh, Daniele Micciancio, Michael Naehrig, Rafail Ostrovsky, David Renardy, Carey Schwartz, Nigel Smart, Vinod Vaikuntanathan, and Michael Walter. We also thank the S&P'18 reviewers for helpful comments.

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] 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.

[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,” Cryptology ePrint Archive, Report 2017/104 [to appear in ACM CCS 2017], 2017, http://eprint.iacr.org/2017/104.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Y. Hu and H. Jia, *Cryptanalysis of GGH Map*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 537–565.

[33] 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

[34] 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.

[35] 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

[36] 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.

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] Z. Brakerski and G. N. Rothblum, “Obfuscating conjunctions,” *Journal of Cryptology*, vol. 30, no. 1, pp. 289–320, 2017.

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

[45] C. F. F. Karney, “Sampling exactly from the normal distribution,” *ACM Trans. Math. Softw.*, vol. 42, no. 1, pp. 3:1–3:14, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2710016

[46] D. Micciancio and M. Walter, “Gaussian sampling over the integers: Efficient, generic, constant-time,” in *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, 2017, pp. 455–485.

[47] J.-S. Coron, T. Lepoint, and M. Tibouchi, *Practical Multilinear Maps over the Integers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 476–493. [Online]. Available: https://doi.org/10.1007/978-3-642-40041-4_26

[48] S. Garg, C. Gentry, and S. Halevi, *Candidate Multilinear Maps from Ideal Lattices*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–17. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38348-9_1
[49] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal, "Hop: Hardware makes obfuscation practical," in *24th Annual Network and Distributed System Security Symposium, NDSS*. Internet Society, February 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/hop-hardware-makes-obfuscation-practical-2/
[50] D. Wichs and G. Zirdelis, "Obfuscating compute-and-compare programs under lwe," Cryptology ePrint Archive, Report 2017/276, 2017, http://eprint.iacr.org/2017/276.
[51] R. Goyal, V. Koppula, and B. Waters, "Lockable obfuscation," Cryptology ePrint Archive, Report 2017/274, 2017, http://eprint.iacr.org/2017/274.
[52] M. Kubat, *An Introduction to Machine Learning*, 1st ed. Springer Publishing Company, Incorporated, 2015.
[53] 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.
[54] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist/, accessed: 2017-11-13.
[55] A. Anand, L. Wilkinson, and D. N. Tuan, "An l-infinity norm visual classifier," in *2009 Ninth IEEE International Conference on Data Mining*, Dec 2009, pp. 687–692.
[56] L. Wilkinson, A. Anand, and D. N. Tuan, "Chirp: A new classifier based on composite hypercubes on iterated random projections," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '11. New York, NY, USA: ACM, 2011, pp. 6–14. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020418
[57] V. Lyubashevsky, C. Peikert, and O. Regev, "A toolkit for ring-LWE cryptography," in *EUROCRYPT*, vol. 7881. Springer, 2013, pp. 35–54.
[58] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient fpga implementations of lattice-based cryptography," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, June 2013, pp. 81–86.
[59] 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
[60] 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
[61] N. Genise and D. Micciancio, "Faster gaussian sampling for trapdoor lattices with arbitrary modulus," Cryptology ePrint Archive, Report 2017/308, 2017, http://eprint.iacr.org/2017/308.
[62] R. El Bansarkhani and J. Buchmann, "Improvement and efficient implementation of a lattice-based signature scheme," in *Selected Areas in Cryptography–SAC 2013*, T. Lange, K. Lauter, and P. Lisoněk, Eds. Springer, 2014, pp. 48–67.
[63] L. Ducas, V. Lyubashevsky, and T. Prest, *Efficient Identity-Based Encryption over NTRU Lattices*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 22–41.
[64] C. Peikert, "An efficient and parallel Gaussian sampler for lattices," in *CRYPTO*, 2010, pp. 80–97.
[65] 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.
[66] N. C. Dwarakanath and S. D. Galbraith, "Sampling from discrete gaussians for lattice-based cryptography on a constrained device," *Applicable Algebra in Engineering, Communication and Computing*, vol. 25, no. 3, pp. 159–180, Jun 2014.
[67] M. Albrecht, S. Scott, and R. Player, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, p. 169203, 10 2015.
[68] 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.
[69] J.-F. Dhem and J.-J. Quisquater, "Recent results on modular multiplications for smart cards," in *Smart Card Research and Applications*, ser. Lecture Notes in Computer Science, J.-J. Quisquater and B. Schneier, Eds. Springer Berlin Heidelberg, 2000, vol. 1820, pp. 336–352.

Appendix A
Pseudocode for Trapdoor Sampling Algorithms

The algorithms described in this section are variations of trapdoor sampling algorithms proposed in [61]. The modifications were made to reduce the number of calls to polynomial CRT operations, increase opportunities for parallel execution, and ease the software implementation.

A more significant modification is in the **Perturb** subroutine of Algorithm 3. Instead of using discrete Gaussian distribution, we switched to the continuous distribution. The use of discrete Gaussian distribution would require a higher value of σ_t, proportional to t^2 rather than $t+1$, due to the Σ_3 condition in Corrolary 3.1 of [61]. This would significantly increase the modulus q (for large t) determined by the correctness constraint (1). The use of the continuous distribution eliminates the Σ_3 condition. A more detailed discussion of this scenario is provided after Corrolary 3.1 in [61].

Algorithm 3 G-sampling [61]

```
function SampleG(σ_t, u, q)          ▷ q = [q]_t^κ is the vector of base-t digits in modulus q
    σ := σ_t / (t + 1)
    l_0 := sqrt(t (1 + 1/κ) + 1)
    h_0 := 0
    d_0 := q_0/t
    for i = 1..κ − 1 do
        l_i := sqrt(t (1 + 1/(κ − i)))                ▷ l_i, h_i are entries in sparse triangular matrix L
        h_i := sqrt(t (1 − 1/{κ − (i − 1)}))
        d_i := (d_{i−1} + q_i) /t                     ▷ d_i are entries in the last column of matrix D
    end for
    Define Z ∈ Z^{κ×n}                                ▷ this vector will store the result of G-sampling
    for i = 0..n − 1 do        ▷ Iterate through all coefficients of polynomial. This loop can be parallelized.
        v := u(i)                                     ▷ v = [v]_t^κ is the vector of digits in coefficient u(i) ∈ Z_q
        p ← Perturb(σ, l, h)                          ▷ p, l, h ∈ R^κ
        c_0 := (v_0 − p_0)/t
        for j = 1..κ − 1 do
            c_j = (c_{j−1} + v_j − p_j)/t
        end for
        z ← SampleD(σ, c, d)                          ▷ z ∈ Z^κ; c, d ∈ R^κ
        t_0 := t · z_0 + q_0 · z_{κ−1} + v_0
        for j = 1..κ − 2 do
            t_j := t · z_j − z_{j−1} + q_j · z_{κ−1} + v_j
        end for
        t_{κ−1} := q_{κ−1} · z_{κ−1} − z_{κ−2} + v_{κ−1}
        Z(:, i) := t                                  ▷ t = (t_0, t_1, ..., t_{κ−1}) ∈ Z^κ
    end for
    return Z
end function
```

```
function Perturb(σ, l, h)                             ▷ l, h ∈ R^κ are the entries in matrix L
    for i = 0..κ − 1 do
        z_i ← SampleR(σ, 0)                           ▷ SampleR is continuous Gaussian sampler
    end for
    for i = 0..κ − 2 do
        p_i = l_i · z_i + h_{i+1} · z_{i+1}
    end for
    p_{κ−1} = h_{κ−1} · z_{κ−1}
    return p                                          ▷ p = (p_0, p_1, ..., p_{κ−1}) ∈ R^κ
end function
```

```
function SampleD(σ, c, d)                             ▷ Sample from the lattice generated by matrix D
    z_{κ−1} ← SampleZ(σ/d_{κ−1}, −c_{κ−1}/d_{κ−1})
    c := c − z_{κ−1} d
    for i = 0..κ − 2 do
        z_i ← SampleZ(σ, −c_i)
    end for
    return z                                          ▷ z = (z_0, z_1, ..., z_{κ−1}) ∈ Z^κ
end function
```

Algorithm 4 Perturbation generation [61]

```
function SAMPLEP_Z(n, q, s, σ_t, (r̂, ê) )
    z := (σ_t^{-2} − s^{-2})^{-1}
    a := s^2 − z Σ_{i=1}^{κ} r̂_i^T r̂_i                        ▷ a ∈ K_{2n}
    b := −z Σ_{i=1}^{κ} r̂_i^T ê_i                              ▷ b ∈ K_{2n}
    d := s^2 − z Σ_{i=1}^{κ} ê_i^T ê_i                          ▷ d ∈ K_{2n}
    for i = 0..nκ − 1 do
        q_i ← SAMPLEZ(√(s^2 − σ_t^2))
    end for
    convert q ∈ Z^{κ×n} to q̂ ∈ R_q^κ       ▷ CRT operations can be executed in parallel
    c := −(σ_t^2/(s^2−σ_t^2)) [r̂; ê] q̂                          ▷ c ∈ K_{2n}^2
    p ← SAMPLE2_Z (a, b, d, c)                                  ▷ p ∈ Z^{2×n}
    convert p ∈ Z^{2×n} to p̂ ∈ R_q^2
    return (p̂, q̂)
end function
```

```
function SAMPLE2_Z(a, b, d, c )
    let c = (c_0, c_1)
    q_1 ← SAMPLEF_Z(d, c_1)                                     ▷ q_1 ∈ Z^n
    convert q_1 ∈ Z^n to q̂_1 ∈ K_{2n}
    c_0 := c_0 + b d^{-1} (q̂_1 − c_1)
    q_0 ← SAMPLEF_Z(a − b d^{-1} b^T, c_0)                      ▷ q_0 ∈ Z^n
    return (q_0, q_1)
end function
```

APPENDIX B
DERIVATION OF CORRECTNESS CONSTRAINT FOR CONJUNCTION OBFUSCATOR

Consider initially the case of a 2-word conjunction obfuscation pattern, where we use $\mathbf{R}_1$, $\mathbf{R}_2$, and $\mathbf{R}_3$ to denote the encoding matrices and $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$ to denote the public keys. The **Encode** operation for the first level can then be expressed as

$$\mathbf{A}_0\mathbf{R}_1 = r_1\mathbf{A}_1 + \mathbf{e}_1 \in \mathcal{R}_q^{1\times m},$$

where r_1 is a product of two uniform ring elements sampled over $\{-1, 0, 1\}^n$, $\mathbf{A}_0 \in \mathcal{R}_q^{1\times m}$, and $\mathbf{R}_1 \in \mathcal{R}_q^{m\times m}$.

The expression corresponding to the minuend in **Evaluate**, i.e., $\mathbf{A}_0\mathbf{S}_\Pi\mathbf{R}_3$, can be written as follows:

$$\begin{aligned}\mathbf{A}_0\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3 = (r_1\mathbf{A}_1 + \mathbf{e}_1)\,\mathbf{R}_2\mathbf{R}_3 = \\ (r_1\,(r_2\mathbf{A}_2 + \mathbf{e}_2) + \mathbf{e}_1\mathbf{R}_2)\,\mathbf{R}_3 = \\ (r_1\,(r_2\,(r_3\mathbf{A}_3 + \mathbf{e}_3) + \mathbf{e}_2\mathbf{R}_3) + e_1\mathbf{R}_2\mathbf{R}_3)\,.\end{aligned}$$

```
function SAMPLEF_Z(f, c )
    if dim(f) = 1 then return SAMPLEZ (√f, c)
    else
        let f(x) = f_0(x^2) + x · f_1(x^2)            ▷ Extract even and odd componets of f(x)
        c' := P_stride(c) ▷ P_stride permutes coefficients (a_0, a_1, ..., a_{n−1}) to (a_0, a_2, ..., a_{n−2}, a_1, a_3, ..., a_{n−1})
        (q_0, q_1) ← SAMPLE2_Z (f_0, f_1, f_0, c')
        let q(x) = q_0(x^2) + x · q_1(x^2)
        return q
    end if
end function
```

Switching to infinity norms, we get the following expression for the noise norm:

$$\|\mathbf{A}_0\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3 - r_1r_2r_3\mathbf{A}_3\|_\infty = \\ \|\mathbf{e}_1\mathbf{R}_2\mathbf{R}_3 + r_1\mathbf{e}_2\mathbf{R}_3 + r_1r_2\mathbf{e}_3\|_\infty .$$

For the subtrahend in **Evaluate**, i.e., $\mathbf{A}_0\mathbf{R}_\Pi\mathbf{S}_3$, we can use the same estimate as an upper bound for the noise. The term $r_1r_2r_3\mathbf{A}_3$ is present in both terms in **Evaluate** by the definition of conjunction obfuscator ($r_1r_2r_3\mathbf{A}_3$ gets eliminated by the subtraction in **Evaluate**). The actual norm of noise terms will be significantly smaller in this case because Gaussian polynomials are sampled using the distribution parameter σ rather than a much larger σ'.

Hence, the norm for a 2-word obfuscated conjunction pattern can be bounded as

$$\widetilde{\Delta} < 2\,\|\mathbf{e}_1\mathbf{R}_2\mathbf{R}_3 + r_1\mathbf{e}_2\mathbf{R}_3 + r_1r_2\mathbf{e}_3\|_\infty .$$

The encoding matrix $\mathbf{R}_i$ contains m rows with infinity norm $B_R = \beta s$ (for the initial encoding matrix before any multiplications of encodings), where $\beta = 2.0$ (was found empirically).

As $\|\mathbf{R}_i\|_\infty \gg r_i$, we have

$$\widetilde{\Delta} < 4\,\|\mathbf{e}_1\mathbf{R}_2\mathbf{R}_3\|_\infty . \tag{3}$$

Parameter B_e is introduced as an upper bound for the values generated using discrete Gaussian distribution and can be taken as $\sigma'\sqrt{\gamma}$, where assurance measure γ can be found empirically (usually between 36 and 144; we set $\gamma := 36$ in our experiments).

If we consider a product of $\mathbf{R}_2$ and $\mathbf{R}_3$, we obtain

$$\|\mathbf{R}_\times\|_\infty = \|\mathbf{R}_2\mathbf{R}_3\|_\infty \leq nmB_R^2.$$

Now consider the product of $\mathbf{e}_1$ and $\mathbf{R}_\times$:

$$\|\mathbf{e}_1\mathbf{R}_\times\|_\infty \leq nmB_e\,\|\mathbf{R}_\times\|_\infty \leq (nm)^2\,B_eB_R^2.$$

As $\mathbf{e}_1$, $\mathbf{R}_1$, and $\mathbf{R}_2$ are generated using zero-centered Gaussian sampling and the number of samples involved in each polynomial multiplication is relatively large, we can apply the Central Limit Theorem to replace every instance of nm with $\sqrt{nm}$, which yields

$$\|\mathbf{e}_1\mathbf{R}_\times\|_\infty \leq nmB_eB_R^2. \tag{4}$$

Applying the **EqualTest** condition ($\Delta < q/8$) and substituting (4) into (3), we obtain a correctness constraint for a 2-word conjunction obfuscator:

$$q > 32\sqrt{mn}B_e\sqrt{mn}B_R^2 = 32B_e\left(\sqrt{mn}B_R\right)^2.$$

Using the 2-word conjunction correctness constraint as the base case, we can derive by induction the following expression for an $\mathcal{L}$-word conjunction:

$$q > 32B_e\left(\sqrt{mn}B_R\right)^{\mathcal{L}} \tag{5}$$

for $\mathcal{L} \geq 2$.

APPENDIX C
COMPARISON OF INTEGER GAUSSIAN SAMPLERS

Table VII shows the comparison of sampling rates for generic integer Gaussian samplers in the desktop computing environment for the case of single-threaded execution. The distribution parameter σ was varied from 2^{17} to 2^{27} to cover the range of distribution parameters used by the subroutines of the G-sampling and perturbation generation procedures in trapdoor sampling for the conjunction obfuscator. These results were used to select the generic sampler for our main obfuscation experiments. The rejection sampling method [65] is included only for reference. Up to 20 MB of memory was allowed for the generic constant-time sampler [46]. The other two methods do not have any significant memory requirements.

Table VII suggests that Karney's method [45] has the highest sampling rate for the distribution parameter range of interest and was thus chosen for our main obfuscation experiments. The sampling rates shown in Table VII are

TABLE VII: Sampliing rates in 10^6 per second for generic integer Gaussian samplers

σ	Rejection sampling [65]	Karney [45]	Constant-time [46]
2^{17}	0.929	3.810	1.587
2^{22}	0.932	3.811	1.502
2^{27}	0.900	3.798	1.507

within 20% of the corresponding rates reported in [46], which suggests that our conclusions are not specific to our implementation but reflect the computational complexity at the algorithmtic level.

It should be noted that both constant-time sampler [46] and Karney's method [45] can be separated into offline and online subroutines. The analysis presented in [46] suggests that the constant-time sampler [46] may be faster in this case. Since the generic integer sampling method is used only in the obfuscation procedure, which is executed offline, this additional complexity is not needed for our application.

Despite a higher runtime, a constant-time sampler, such as [46], could be preferred in practice over a rejection sampler, like [45], because it reduces the opportunities for timing attacks.

APPENDIX D
PSEUDOCODE FOR CONJUNCTION OBFUSCATION ALGORITHMS

When the ring instantiation of directed encoding (described in section III-B) is applied to the conjunction obfuscator, the encodings $\mathbf{R}_{i,b}, \mathbf{S}_{i,b}, \mathbf{R}_{L+1}, \mathbf{S}_{L+1}$ get represented as matrices of $m \times m$ ring elements over $\mathcal{R}_q$.

The key generation algorithm for the ring instantiation of conjunction obfuscator is listed in Algorithm 5. Parameter $\mathcal{L}$ is the effective length of conjunction pattern.

Algorithm 5 Key generation

```
function KeyGen(1^λ)
    for i = 0..𝓛+1 do
        A_i, T̃_i := TrapGen(1^λ)
    end for
return K_{𝓛+1} := ({A_i, T̃_i}_{i∈{0,..,𝓛+1}})
end function
```

The conjunction obfuscator relies on the **Encode** algorithm of directed-encoding ring instantiation (defined in Section III-B) to encode each part of the conjunction pattern. The **Encode** algorithm is depicted in Algorithm 6.

Algorithm 6 Directed encoding

```
function Encode_{A_i→A_{i+1}}(T_i, r, σ)
    e_{i+1} ← D_{R,σ} ∈ R_q^{1×m}.
    b_{i+1} := r A_{i+1} + e_{i+1} ∈ R_q^{1×m}
    R_{i+1} := GaussSamp(A_i, T_i, b_{i+1}, σ_t, s) ∈ R_q^{m×m}
return R_{i+1}
end function
```

Algorithm 7 lists the pseudocode for the main obfuscation function. In contrast to the obfuscated program defined in Section III-A, we encode words of conjunction pattern $\mathbf{v} \in \{0, 1, \star\}^L$. 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 (multl-inearity degree). The word encoding is a major optimization proposed in this work, and is discussed in detail in Section III-C.

The $s_{i,b}, r_{i,b}$ elements are ternary uniformly random ring elements, i.e., sampled over $\{-1, 0, 1\}^n$, for $i \in [\mathcal{L}]$ and $b \in \{0, \ldots, 2^w - 1\}$. We set $s_{i,b} = \cdots = s_{i,j}$ for indices $b, \cdots, j$ corresponding to the same wildcard subpattern. To implement these wildcard subpatterns, we rely on binary masks, where the subpattern with all zeros in the

wildcard characters is used to generate a uniformly random ring element, which is then reused for all subpatterns with non-zero bits in the wildcard characters.

The obfuscated program then transforms to

$$\Pi_{\mathbf{v}} := \left(\mathbf{A}_0, \{\mathbf{S}_{i,b}, \mathbf{R}_{i,b}\}_{i\in[\mathcal{L}],b\in\{0,\ldots,2^w-1\}}, \mathbf{R}_{\mathcal{L}+1}, \mathbf{S}_{\mathcal{L}+1}\right).$$

Algorithm 7 Obfuscation

```
function OBFUSCATE(v ∈ {0,1,*}^L, K_{L+1}, σ, σ')
    {r_{i,b}}_{i∈[L],b∈{0,...,2^w−1}} ← T
    for i = 1..L do
        Build binary wildcard mask M_⋆
        for b = 0..2^w-1 do
            if (b ∧ M) = 0 then
                s_{i,b} ← T
            else
                j := b ∧ ¬M
                s_{i,b} := s_{i,j}
            end if
        end for
    end for
    for i = 1..L do
        for b = 0..2^w-1 do
            S_{i,b} := Encode_{A_{i−1}→A_i}(T̃_{i−1}, s_{i,b} · r_{i,b}, σ')
            R_{i,b} := Encode_{A_{i−1}→A_i}(T̃_{i−1}, r_{i,b}, σ)
        end for
    end for
    r_{L+1} ← T ∈ R
    s_× := r_{L+1} ∏_{i=1}^{L} s_{i,v[1+(i−1)w : iw]}
    S_{L+1} := Encode_{A_L→A_{L+1}}(T̃_L, s_×, σ)
    R_{L+1} := Encode_{A_L→A_{L+1}}(T̃_L, r_{L+1}, σ')
    Π_v := (A_0, {S_{i,b}, R_{i,b}}_{i∈[L],b∈{0,...,2^w−1}}, R_{L+1}, S_{L+1})
return Π_v
end function
```

Algorithm 8 Optimized Evaluation

```
function EVALUATE(x ∈ {0,1}^L, Π_v)
    S_Π := A_0 ∈ R_q^{1×m}
    R_Π := A_0 ∈ R_q^{1×m}
    for i = 1..L do
        S_Π := S_Π S_{i,x[1+(i−1)w : iw]} ∈ R_q^{1×m}
        R_Π := R_Π R_{i,x[1+(i−1)w : iw]} ∈ R_q^{1×m}
    end for
    Δ := ‖S_Π R_{L+1} − R_Π S_{L+1}‖_∞
return Δ ≤ q/8
end function
```

Algorithm 7 operates with two variants of **Encode** distinguished by the distribution parameter used. To encode ring elements $r_{i,b}$ and $s_\times$, we sample using σ. To encode ring elements $s_{i,b} \cdot r_{i,b}$ and $r_{\mathcal{L}+1}$, we use $\sigma' = k\sqrt{n}\sigma$. We need to use a larger value of distribution parameter in order to apply the Ring-LWE assumption to "secret" ring

elements $s_{i,b} \cdot r_{i,b}$ in the security proof for the ring variant of directed encoding specific to conjunction obfuscator, which is presented in section 4.3 of [1].

Note that the security proof presented in Section 4.3 of [1] has typos in expression (1) and **Hybrid** 1 distribution. The vectors $\mathbf{e}'_0$ and $\mathbf{e}'_1$ should be sampled from $\mathcal{D}_{\mathcal{R}^m,\sigma'}$ rather than $\mathcal{D}_{\mathcal{R}^m,\sigma}$ (here, we use the notation of [1]). This typo does not affect the rest of **Hybrid** distributions and the correctness of the proof itself.

The use of ternary distribution $\mathcal{T}$ implies that we rely on a small-secret variant of the Ring-LWE assumption to minimize the noise growth.

The pseudocode for the optimized evaluation procedure is presented in Algorithm 8 (optimization is described in VI-A). Just like in the abstract algorithm described in section III-A, if both $\mathbf{S}_\Pi$ and $\mathbf{R}_\Pi$ are the encodings of the same value, the result of $F_\mathbf{v}$ is 1. Otherwise, the result is 0. The infinity norm computation finds a coefficient with the maximum absolute value in the row vector of ring elements $\mathbf{A}_0\left(\mathbf{S}_\Pi \mathbf{R}_{\mathcal{L}+1} - \mathbf{R}_\Pi \mathbf{S}_{\mathcal{L}+1}\right) \in {\mathcal{R}_q}^{1 \times m}$. The inequality $\Delta \leq q/8$ comes directly from **EqualTest** in the ring instantiation of directed encoding (Section III-B).