Implementing BP-Obfuscation Using Graph-Induced Encoding

Shai Halevi
IBM Research

Tzipora Halevi*
CUNY Brooklyn College

Victor Shoup
IBM Research and NYU

Noah Stephens-Davidowitz†
NYU

September 23, 2017

Abstract

We implemented (a simplified version of) the branching-program obfuscator due to Gentry et al. (GGH15), which is itself a variation of the first obfuscation candidate by Garg et al. (GGHRSW13). To keep within the realm of feasibility, we had to give up on some aspects of the construction, specifically the "multiplicative bundling" factors that protect against mixed-input attacks. Hence our implementation can only support read-once branching programs.

To be able to handle anything more than just toy problems, we developed a host of algorithmic and code-level optimizations. These include new variants of discrete Gaussian sampler and lattice trapdoor sampler, efficient matrix-manipulation routines, and many tradeoffs. We expect that these optimizations will find other uses in lattice-based cryptography beyond just obfuscation.

Our implementation is the first obfuscation attempt using the GGH15 graded encoding scheme, offering performance advantages over other graded encoding methods when obfuscating finite-state machines with many states. In out most demanding setting, we were able to obfuscate programs with input length of 20 nibbles (80 bits) and over 100 states, which seems out of reach for prior implementations. Although further optimizations are surely possible, we do not expect any implementation of current schemes to be able to handle much larger parameters.

Keywords. Implementation, Multilinear Maps, Obfuscation, Trapdoor Lattice Sampling

Supported by the Defense Advanced Research Projects Agency (DARPA) and Army Research Office(ARO) under Contract No. W911NF-15-C-0236.

*Work done while at IBM Research
†Work done while visiting IBM Research

Contents

1 Introduction

General-purpose code obfuscation is an amazingly powerful technique, letting one hide secrets in arbitrary running software. The emergence of plausible constructions for cryptographic general-purpose obfuscation has transformed our thinking about what can and cannot be done in cryptography. Following the first construction by Garg et al. [10] (herein GGHRSW), most contemporary candidates include a "core component" that obfuscates simple functions (usually expressed as branching programs), and a transformation that bootstraps this core component to deal with arbitrary circuits. The core branching-program obfuscator consists of two steps:

- We first randomize the branching program to obscure intermediate states without changing the final outcome.
- Then we encode the randomized programs using a graded-encoding scheme [9], roughly "encrypting" the randomized program while still supporting the operations needed to evaluate it on any given input.

Unfortunately, we essentially have only three candidate graded-encoding schemes to use for the second step, namely GGH13 [9], CLT13 [7], and GGH15 [11], and they are all very inefficient.[1] As a result, so are all existing branching-program obfuscators, to the point that it is not clear if they can be used for any non-trivial purpose.

In this report, we describe our implementation of (a simplified version of) the branching-program obfuscator due to Gentry et al. [11], which is itself a variation of the GGHRSW construction, adjusted to use the GGH15 graph-based graded encoding scheme [11]. To keep within the realm of feasibility, we had to give up on some aspects of the construction, specifically the "multiplicative bundling" factors that protect against mixed-input attacks. Hence, our implementation can securely obfuscate only read-once branching programs. Nonetheless, when stretched to its limits, our implementation can obfuscate some non-trivial programs (beyond just point functions). Our use of the GGH15 encoding may offer performance advantages over implementations that use the encoding from GGH13 [9] or CLT13 [7], especially for obfuscating finite-state machines with many states. For example we were able to obfuscate read-once branching programs with input of 20 nibbles (80 bits) and over 100 states, which seems out of reach for all prior implemented systems that we know of.

Such branching programs can be used to implement multi-point functions or even multiple-substring match functions, checking if the input contains at least one out of a list of 100 possible substrings. They can also be used to obfuscate superstring match functions, checking if the input is contained in a longer string of length up to $100 + 20$ nibbles (or contained in one of two strings, each of length $50 + 20$, etc.).

To handle length 20 branching programs with 100 states over nibbles, our implementation uses about 400 Gigabytes of memory and runs for about 23 days.[2] Moreover, each obfuscated program takes about 9 Terabytes of disk space to specify. Evaluating this obfuscated program takes under 25 minutes per input. We note that further optimization of the current implementation is surely possible, but this scheme probably cannot extend too much beyond what we achieved.

[1] Moreover, their security properties are still poorly understood.

[2] The results in Table 4 are for a *binary alphabet*, featuring the same RAM consumptions but 1/8 of the disk space and running time as compared to size-16 alphabet.

Obfuscation [10, 11] (Section 2)
Graded encoding [11] (Section 3)
Trapdoor Sampling [18] & Matrix manipulations (Sections 4 & 6)

Figure 1: High-level structure of our implementation

Why GGH15? Our choice of using the encoding scheme from [11] was mainly due to the fact that using this scheme in an implementation was not attempted before, and our expectation that using GGH15 would have somewhat smaller complexity than the GGH13 and CLT13 encoding schemes when obfuscating programs with many states.

Specifically, GGH13 and CLT13 encode individual elements, but branching-program obfuscators need to encode matrices. As a result, the performance of obfuscators based on CLT13 or GGH13 degrades quadratically with the matrix dimension.[3] In contrast, GGH15 natively encodes matrices, and its performance is a lot less sensitive to the dimension of the encoded matrices. Very very roughly, the performance of encoding an n-by-n matrix scales like $(M+n)^2$, where M is a large factor that depends on security considerations — see more details in Section 5. From the performance numbers reported in [16] and in this work, it seems that CLT13 should be the graded encoding scheme of choice for small-dimension matrices, while GGH15 would start outperforming CLT for programs with about 50 states.

Moreover the optimizations that we developed in this work are likely to be useful beyond obfuscation. In particular our Gaussian and trapdoor sampling optimizations are likely useful in lattice cryptography (and of course our optimized matrix manipulations would be useful in many other settings).

1.1 High-Level Overview

Our implementation consists of three layers, as illustrated in Figure 1. Our implementation consists of three layers: At the top layer, we implemented a simplified variant of the GGH15 obfuscator from [11] (see Section 2). Below it, in the middle layer, we implemented the GGH15 graded-encoding scheme [11], including the "safeguards" suggested there (see Section 3). The main operations needed in the GGH15 encoding scheme are lattice trapdoor sampling and matrix manipulations, which are implemented in the bottom layer of our system (see Sections 4 and 6, respectively). The most notable aspects of our implementations are:

New Gaussian sampler. We implemented a sampling procedure for the ellipsoidal discrete Gaussian distribution that can directly work with the covariance matrix (rather than its square root). Recall that sampling an ellipsoidal Gaussian over $\mathbb{Z}^n$ with covariance matrix Σ is equivalent to sampling a spherical Gaussian over the lattice whose basis is $B = \sqrt{\Sigma}$. Hence, one way to implement it would be computing the basis B and then using the procedure of GPV [13] or Peikert [19]. However, computing $\sqrt{\Sigma}$ is somewhat inefficient, so we instead devised a different method which is somewhat similar to the GPV sampler but works directly with Σ, without having to find its square root. This method is described in Section 4.4.

[3] Some obfuscation constructions can handle non-square matrices, but even then it is likely that both dimensions would grow together.

Trapdoor sampling in the CRT representation. We implemented the Micciancio-Peikert trapdoor sampling algorithm [18], using a procedure that keeps all the large integers in the Chinese-Remainder representation (CRT), without having to convert back and forth between the CRT and standard integer representations. See Section 4.3.

Efficient matrix manipulation. Our implementation routinely handles matrices of dimension above 10,000, so efficient matrix multiplication and inversion is critical. We implemented highly optimized routines, taking advantage of the available hardware (cache friendly, SIMD enabled, multi-threaded, etc.). These routines, which were incorporated into the NTL library [20], are described in Section 6.

Threading/memory tradeoffs. We explored multiple parallelization strategies, trading off the level of parallelism against the need to severely conserve memory. Details are available in Section 7.2.

Some Design Choices. Most of our design choices were taken for the sake of speed. For example, this is why we chose to work with integers in CRT representation (with the CRT basis being either 23-bit numbers or 60-bit numbers, depending on the hardware platform[4]). That choice dictated that our "gadget matrix" G would be based on mixed-radix representation relative to our CRT base, rather than binary representation (see Section 4.3). Another choice made in the name of speed was to use 1-dimensional rounded continuous Gaussian distribution instead of the discrete Gaussian distribution (see Section 4.4).

Other choices were made for more prosaic reasons such as to simplify the software structure of the implementation. For example, since the "safeguards" of GGH15 encoding from [11] already include Kilian-like randomization, we chose to implement that randomization techniques at the encoding level and leave it out of the higher-level obfuscation routine. We made many such software-engineering choices during the implementation, but only report here on very few of them.

The source code of our implementation is available as open-source, under the Apache license v2.0, from `https://github.com/shaih/BPobfus`.

1.2 Prior Work

Graded encoding implementations. An implementation of the CLT13 graded encoding scheme was provided already by Coron et al. [7], and GGHlite (which is a simplified variant of GGH13 due to Langlois et al. [15]) was implemented by Albrecht et al. [2]. To the best of our knowledge, ours is the first implementation of the GGH15 graded encoding scheme.

Obfuscation implementation. The first attempt at implementing the GGHRSW obfuscator was due to Apon et al. [4], who used the CLT13 implementation for the underlying graded encoding scheme, and demonstrated a 14-bit point function obfuscation as a proof of concept. That implementation work was greatly enhanced in the 5Gen work of Lewi et al. [16]: they built a flexible framework that can use either CLT13 or GGH13 encoding (but not GGH15), and implemented obfuscation (as well as other primitives) on top of this framework, demonstrating an obfuscation of an 80-bit point function using CLT13 encoding.

[4] We used 23-bit factors when utilizing Intel AVX, and 60 bits when AVX was not available. See Section 6.

We note that point functions have branching programs of very low dimension, making the CLT13-based approach from [16] attractive. However, our implementation should out-perform the CLT13-based approach once the number of states (and hence the dimension) grows above a few dozens.

Attacks. The security properties of current-day graded encoding schemes is poorly understood, and new attacks are discovered all the time. But no attacks so far seem to apply to the construction that we implemented. In particular, the attack due to Coron et al. [8] on GGH15-based key exchange relies on having many encodings of the same matrices, which are not available in obfuscation. Also, the quantum attack due to Chen et al. [6] on GGH15-based obfuscation is not applicable at all to read-once branching programs, since it specifically targets the "bundling factors" that are used as a defense against inconsistent evaluation. (Also that attack requires that the branching-program matrices have full rank, which is not the case in our setting.)

1.3 Organization

The three layers of our implementation (obfuscation, graded encoding, and trapdoor sampling) are described in Sections 2, 3, and 4, respectively. Paremeter selection is discussed in Section 5. The matrix-manipulation optimizations are in Section 6, and more implementation details and performance results are given in Section 7.

2 Obfuscating Branching Programs

We implemented a simplified variant of the obfuscator of Gentry et al. [11], without the "multiplicative bundling" mechanism that protects against mixed-input attacks. Hence, in its current form our implementation is only secure when used to obfuscate oblivious read-once branching programs (equivalently, nondeterministic finite automata, NFA).

Recall that a read-once branching program for n-bit inputs is specified by a length-n list of pairs of $d \times d$ matrices $\mathcal{B} = \{(M_{1,0}, M_{1,1}), (M_{2,0}, M_{2,1}), \ldots, (M_{n,0}, M_{n,1})\}$, and the function computed by this program is

$$f_{\mathcal{B}}(x) = \begin{cases} 0 & \text{if } \prod_{i=1}^{n} M_{i,x_i} = 0; \\ 1 & \text{otherwise.} \end{cases}$$

We remark that in other settings it is common to define the function $f_{\mathcal{B}}(x)$ by comparing the product $\prod_{i=1}^{n} M_{i,x_i}$ to the identity rather than the zero matrix. But comparing to the identity requires that all the matrices be full rank, which for read-once programs will severely limit the power of this model. Instead, comparing to the zero matrix allow us to represent arbitrary oblivious NFAs (where the product is zero if and only if there are no paths leading to the accept state).

Our goal is to get "meaningful obfuscation," which is usually defined as achieving indistinguishability obfuscation (iO). Namely, we want it to be hard for an efficient distinguisher to tell apart the obfuscation of two equivalent programs, $\mathcal{B}, \mathcal{B}'$ such that $f_{\mathcal{B}} = f_{\mathcal{B}'}$. For very simple functions, other notions may also be possible such as one-wayness or virtual black box (VBB) obfuscation. The security analysis from [11] (sans the "bundling factors") implies that it may be reasonable to hope that our implementation satisfies iO for NFAs, and perhaps also the other notions for limited classes.

The GGH15 construction proceeds in two steps: the input branching program is first randomized, and the resulting program is encoded using the underlying graded encoding scheme. Roughly speaking, the goal of graded encoding is to ensure that "the only thing leaking" from the obfuscated program is whether or not certain matrices are equal to zero, and the goal of randomization is to ensure that "the only way to get a zero matrix" is to faithfully evaluate the branching program on some input x. That is, randomization needs to ensure that every expression not derived from a faithful evaluation yields a non-zero matrix with high probability.

2.1 Randomizing the Transition Matrices

The GGH15 construction has three randomization steps: it embeds the branching program in higher-dimensional matrices, then applies Kilian-style randomization [14], and finally multiplies by "bundling scalars". In our implementation, we forgo the "bundling scalars" for performance reasons (see below), and we chose to delegate the Kilian-style randomization to the graded encoding itself (see Section 3.1), rather than viewing it as part of the obfuscation.

Hence the only randomization that we implemented in the obfuscation layer is embedding in high-dimensional random matrices. The transition matrix that we want to encode is embedded in the upper-left quadrant of a higher-dimension block-diagonal matrix, setting the lower-right quadrant to be a random (small) matrix. Specifically, we add $d' = \lceil \sqrt{\lambda/2} \rceil$ dimensions in the lower-right $d' \times d'$ quadrant (where λ is the security parameter), with random entries of magnitude roughly 2^7. These random matrices therefore have significantly more than 2λ bits of min-entropy, so they are not susceptible to guessing (or birthday-type) attacks.

This randomization impedes functionality, however, since after multiplying we can no longer compare the result to zero. To recover functionality, we use the "dummy program" method of Garg et al. [10], where we encode not only the randomized "real" program but also a second "dummy program" in which all the matrices are always multiplied to the all-zero matrix. Specifically, for the first step we use transition matrices that have the identity at the top rows and zero at the bottom; for the last step we use matrices where the top rows are zero and the bottom have the identity; and in between we use the identity. We randomize the "real" and "dummy" programs using the same random low-norm matrices in the lower-right quadrants, as illustrated in Figure 2.

After this randomization step, we use the GGH15 graph-induced encoding scheme to encode the resulting randomized matrices. As described in Section 3, the GGH15 scheme encodes matrices relative to edges of a public directed graph, and in our case we use a construction with two separate chains (with the same source and sink), one for encoding the "real" branch and the other for encoding the "dummy" branch. The encoded matrices form our obfuscated branching program. To evaluate this obfuscated branching program on some input, we choose the same matrix (0 or 1) from both the "real" and "dummy" programs, multiply in order, subtract the product of the "dummy" program from the product of the "real" one, and test for zero.

2.2 No "multiplicative bundling"

The obfuscation scheme as described by Gorbunov et al. in [11] has another randomization step of "multiplicative bundling" to protect against "mixed input attacks": in general branching programs, one has to ensure that when a single input bit controls many steps of the branching program, an attacker is forced to either choose the matrix corresponding to zero in all of the steps or the matrix corresponding to one (but cannot mix and match between the different steps of the same input

$$\text{First matrix: } \left(M_{1,b}\right) \Rightarrow \left\{ \quad \text{Real: } \begin{pmatrix} M_{1,b} & \\ & R_{1,b} \end{pmatrix}, \quad \text{Dummy: } \begin{pmatrix} I_{\lfloor d/2 \rfloor} & & \\ & 0_{\lceil d/2 \rceil} & \\ & & R_{1,b} \end{pmatrix} \right\} \tag{1}$$

$$\text{Other matrices: } \left(M_{i,b}, 1 < i < n\right) \Rightarrow \left\{ \quad \text{Real: } \begin{pmatrix} M_{i,b} & \\ & R_{i,b} \end{pmatrix}, \quad \text{Dummy: } \begin{pmatrix} I_d & \\ & R_{i,b} \end{pmatrix} \right\} \tag{2}$$

$$\text{Last matrix: } \left(M_{n,b}\right) \Rightarrow \left\{ \quad \text{Real: } \begin{pmatrix} M_{n,b} & \\ & R_{n,b} \end{pmatrix}, \text{Dummy: } \begin{pmatrix} 0_{\lfloor d/2 \rfloor} & & \\ & 1_{\lceil d/2 \rceil} & \\ & & R_{n,b} \end{pmatrix} \right\} \tag{3}$$

Figure 2: Randomizing transition matrices, the $R_{i,b}$'s are random low-norm matrices, and the same $R_{i,b}$ is used for the "real" and "dummy". I_k and 0_k are the $k \times k$ identity matrix and $k \times k$ zero matrix respectively.

bit). To do that, Gorbunov et al. use a variant of the encoding scheme that encodes matrices over a large extension ring (rather than integer matrices), and multiply different matrices by different scalars (called "bundling factors") from this large ring.

Our implementation does not use bundling factors, since to get sufficient entropy for the bundling scalars we would have to work with scalars from a fairly large extension ring R (rather than just the integers). This would have required that we scale up all our parameters by a factor equal to the extension degree of the large ring (at least in the hundreds), rendering the system unimplementable even for very short inputs. As a result, our implementation is vulnerable to mixed input attacks if it is used for general branching programs, so it should only be used for read-once programs.

2.3 Non-binary input

Our implementation also supports non-binary input, namely input over an alphabet Δ with more than two symbols. The only difference is that instead of having pairs of matrices in each step of the program, we have $|\Delta|$ matrices per step, and the input symbol determines which of them to use. We still have only two branches in the obfuscated program ("real" and "dummy"), encoded relative to a DAG with two chains, where on each edge in this DAG we encode $|\Delta|$ matrices. The run time and space requirements of obfuscation are linear in $|\Delta|$, while initialization and evaluation are unaffected by the alphabet size. In our tests, we used $|\Delta|$ as large as 16.

3 Graph-Induced Encoding

Graded encoding schemes are the main tool in contemporary obfuscation techniques. They allow us to "encode" values to hide them, while allowing a user to manipulate these hidden values. A graded encoding scheme has three parts: key generation, which outputs a public key and a secret

key; an encoding procedure, which uses the secret key to encode values of interest; and operations that act on the encoded values using the public key. (These encoding schemes are "graded" in that the encoded values are tagged and operations are only available on encoded values relative to "compatible" tags.)

In the GGH15 graph-induced encoding scheme of Gentry, Gorbunov, and Halevi [11], the tags correspond to edges in a transitive directed acyclic graph (DAG). The DAG has a single source node s and a single sink node t. An instance of this scheme is parameterized by the underlying graph, and also by some integers $n < m < b < q$ that can be derived from the graph and the security parameter. (Roughly, we have $m = O(n \log q)$, $q = n^{O(d)}$, where d is the diameter of the graph and $b = q^\delta$ for some $\delta < 1$.) With these parameters, the functionality of the GGH15 scheme is as follows:

- The plaintext space consists of integer matrices $M \in \mathbb{Z}^{n \times n}$. An encoding of such an integer matrix M (relative to any edge $i \to j$) is a matrix $C \in \mathbb{Z}_q^{m \times m}$ over $\mathbb{Z}_q$. There is an efficient procedure that takes the secret key, a matrix $M \in \mathbb{Z}^{n \times n}$, and two vertices i, j, and produces a matrix $C \in \mathbb{Z}_q^{m \times m}$ that encodes M relative to $i \to j$.

- If C_1, C_2 encode M_1, M_2, respectively, relative to the same edge $i \to j$, then their sum modulo q, $C = [C_1 + C_2]_q$, encodes the matrix $M_1 + M_2$ relative to the same edge. (Here and throughout the paper we use $[\cdot]_q$ to denote operations modulo q, representing elements in $\mathbb{Z}_q$ as integers in the interval $[-q/2, q/2)$.)

- If C_1, C_2 encode M_1, M_2, relative to consecutive edges $i \to j$, $j \to k$, respectively, and if in addition the entries of M_1 are all smaller than b in magnitude, then their product $C = [C_1 \times C_2]_q$ encodes the matrix $M_1 \times M_2$ relative to the edge $i \to k$ (in the transitive closure).

- There is an efficient zero-test procedure that, given the public key and an encoding of some matrix M relative to the source-to-sink edge $s \to t$, determines if $M = 0$.

In more detail, key generation in the basic GGH15 scheme chooses for every vertex i a matrix $A_i \in \mathbb{Z}_q^{n \times m}$, together with a trapdoor as in [18] (see Section 4.1). The secret key consists of all the matrices and their trapdoors, and the public key consists of the source-node matrix A_s. An encoding of a plaintext matrix M with respect to edge $i \to j$ is a "low-norm" matrix $C \in \mathbb{Z}^{m \times m}$ such that $A_i C = M A_j + E \pmod q$, for a "low-norm" noise matrix $E \in \mathbb{Z}^{n \times m}$. The encoding procedure chooses a random small-norm error matrix E, computes $B = [MA_j + E]_q$, and uses trapdoor-sampling to find a small-norm matrix C as above. Encoded matrices relative to the same edge $A_i \to A_j$ can be added, and we have $A_i(C + C') = (M + M')A_j + (E + E') \pmod q$. Furthermore, encoded matrices relative to consecutive edges $i \to j$, $j \to k$ can be multiplied, such that

$$\begin{aligned} A_i(C \times C') &= (MA_j + E) \times C' = M(M'A_k + E') + EC' \\ &= MM'A_k + (ME' + EC') \pmod q. \end{aligned}$$

More generally, if we have a sequence of C_i's representing M_i's relative to $(i-1) \to i$ for

$i = 1, 2, \ldots, k$, then we can set $C^* = \left[\prod_{i=1}^{k} C_i\right]_q$ and we have

$$\begin{aligned} A_0 C^* &= \Big(\prod_{i=1}^{k} M_i\Big) A_k \;+\; \Big(\prod_{i=1}^{k-1} M_i\Big) E_k + \Big(\prod_{i=1}^{k-2} M_i\Big) E_{k-1} C_k + \cdots \\ &+\; M_1 E_2 \Big(\prod_{i=3}^{k} C_i\Big) + E_1 \Big(\prod_{i=2}^{k} C_i\Big) \\ &= \Big(\prod_{i=1}^{k} M_i\Big) A_k \;+\; \underbrace{\sum_{j=1}^{k} \Big(\prod_{i=1}^{j-1} M_i\Big) E_j \Big(\prod_{i=j+1}^{k} C_i\Big)}_{E^* :=} \pmod q. \end{aligned}$$

If we set the parameters so as to ensure that $\|E^*\| \ll q$, then we still have the invariant that C^* is an encoding of $M^* = \prod M_i$ relative to the source-to-sink edge $0 \to k$. By publishing the matrix A_0, we make it possible to recognize the case $M^* = 0$ by checking that $\|A_0 C^*\| \ll q$.

3.1 The GGH15 "Safeguards"

Since the security properties of their construction are still unclear, Gentry et al. proposed in [11] certain "safeguards" that can plausibly make it harder to attack. With every matrix A_i, we also choose a low-norm "inner transformation matrix" $P_i \in \mathbb{Z}^{n \times n}$ and a random "outer transformation matrix" $T_i \in \mathbb{Z}_q^{m \times m}$, both invertible modulo q. For the source node A_0 and sink node A_k, we use the identity matrices $P_0 = P_k = I_{n \times n}$ and $T_0 = T_k = I_{m \times m}$.

To encode a matrix $M \in \mathbb{Z}^{n \times n}$ relative to the edge $i \to j$, we first apply the inner transformation to the plaintext, setting $M' = [P_i^{-1} M P_j]_q$, then compute a low-norm C as before satisfying $A_i C = M' A_j + E$, and finally apply the outer transformation to the encoding, outputting $\widehat{C} = [T_i C T_j^{-1}]_q$. For the invariant of this encoding scheme, $\widehat{C}$ encodes M relative to $i \to j$ if

$$A_i (T_i^{-1} \widehat{C} T_j) \;=\; (P_i^{-1} M P_j) A_j + E \pmod q, \tag{4}$$

where E and $C = [T_i^{-1} \widehat{C} T_j]_q$ have low norm. Since we get telescopic cancellation on multiplication (and the 0 and k transformation matrices are the identity), then the non-small matrices all cancel out on a source-to-sink product. Setting $C^* = \left[\prod_{i=1}^{k} \widehat{C}_i\right]_q$, we get

$$A_0 C^* = \Big(\prod_{i=1}^{k} M_i\Big) A_k + \sum_{j=1}^{k} \Big(\prod_{i=1}^{j-1} M_i\Big) P_{j-1} E_j \Big(\prod_{i=j+1}^{k} C_i\Big) \pmod q. \tag{5}$$

Remark. Gentry et al. described in [11] a slightly more restricted form of the inner transformation, in which both P and its inverse had low norm. But in fact there is no need for P^{-1} to be small. Indeed, Eqn. (5) depends only on the P_i's and not their inverses.

3.2 A Few Optimizations

Special case for source-based encoding. Encoding relative to an edge $0 \to i$ can be substantially optimized. This is easier to see without the safeguards, where instead of publishing the vertex-matrix A_0 and the encoding-matrix C, we can directly publish their product $A_0 C = [M A_j + E]_q$.

Trapdoor Generation & Sampling (§4.1)	
G-sampling (§4.3)	Ellipsoidal Gaussians (§4.4)
Stash (§4.5)	1-dimensional Gaussians (§4.4)

Figure 3: Components of our trapdoor-sampling.

Not only would this save some space (since A_0C has dimension $n \times m$ rather than $m \times m$), we could also do away with the need to use the trapdoor to compute a low-norm C. Instead, we just choose the low-norm E and compute $B = [MA_j + E]_q$.

When using the safeguards, we recall that P_0 and T_0 are both set as the identity, and so to encode M we would compute $M' = [MP_j]_q$, then set $B = [M'A_j + E]_q$, and output $\widehat{B} = [BT_j^{-1}]_q$.

Special case for sink-bound encoding. We can also optimize encodings relative to an edge $j \to k$ by choosing a much lower-dimensional matrix A_k. Specifically, we make $A \in \mathbb{Z}_q^{n\times 1}$, i.e. a single column vector. Note that we cannot choose such a matrix with a trapdoor, but we never need to use a trapdoor for the sink A_k.

Encoding M relative to $j \to k$ is done by choosing a low-norm column vector E, setting $B = [P_j^{-1}MA_k + E]_q \in \mathbb{Z}_q^{n\times 1}$, using the A_j trapdoor to sample a small $C \in \mathbb{Z}_q^{m\times 1}$ such that $A_jC = B \pmod q$, and finally outputting $\widetilde{C} = [T_j^{-1}C]_q$.

4 Trapdoor Sampling

We implemented the trapdoor-sampling procedure of Micciancio and Peikert [18]. The structure of this implementation is depicted in Figure 3. At the bottom level, we have an implementation of one-dimensional discrete Gaussian sampling, with a stash for keeping unused samples. At the next level, we have procedures for sampling high-dimension ellipsoidal discrete Gaussians and solutions to equations of the form $G\vec{z} = \vec{v} \pmod q$, where G is the "easy gadget matrix." At the top level, we have procedures for choosing a matrix A with a trapdoor and then using the trapdoor to sample solutions to equalities of the form $A\vec{x} = \vec{u} \pmod q$.

4.1 Background: The Micciancio-Peikert Trapdoor Sampling Procedure

Recall that the Micciancio-Peikert approach [18] is based on a "gadget matrix" $G \in \mathbb{Z}^{n\times w}$ for which it is easy to sample small solutions $\vec{z}$ to equations of the form $G\vec{z} = \vec{v} \pmod q$. The trapdoor-generation procedure outputs a (pseudo)random matrix $A \in \mathbb{Z}_q^{n\times m}$ together with a low-norm trapdoor matrix $R \in \mathbb{Z}^{\bar{m}\times w}$ such that $A \times \binom{R}{I} = G \pmod q$, where $\binom{R}{I}$ has the top rows taken from R and the bottom rows taken from the identity matrix I. In our implementation, the entries of R are drawn independently from a Gaussian distribution over the integers with parameter $r = 4$.

Given the matrix A, the trapdoor R, and a target syndrome vector $\vec{u} \in \mathbb{Z}_q^n$, Micciancio and Peikert described the following procedure for sampling small solutions to $A\vec{x} = \vec{u} \pmod q$:

1. Sample a small perturbation vector $\vec{p} \in \mathbb{Z}^m$ according to an ellipsoidal discrete Gaussian distribution, with covariance matrix that depends on R.

2. Set $\vec{v} = \vec{u} - A\vec{p} \bmod q$.

3. Sample a small solution $\vec{z}$ to the equation $G\vec{z} = \vec{v} \pmod q$, according to a spherical discrete Gaussian.

4. Output $\vec{x} = \vec{p} + \left(\frac{R}{I}\right)\vec{z} \bmod q$.

Note that this indeed yields a solution to $A\vec{x} = \vec{u} \pmod q$, since

$$A\vec{x} = A\left(\vec{p} + \left(\tfrac{R}{I}\right)\vec{z}\right) = A\vec{p} + G\vec{z} = A\vec{p} + \vec{v} = \vec{u} \pmod q.$$

Also, if $\vec{p}$ is chosen relative to covariance matrix Σ_p and $\vec{z}$ is chosen from a spherical discrete Gaussian with parameter σ_z, then the covariance matrix of the resulting $\vec{x}$ will be $\Sigma_x = \Sigma_p + \sigma_z^2 \cdot \left(\frac{R}{I}\right) \times \left(R^T | I\right)$. Thus, to get a solution $\vec{x}$ sampled from a spherical Gaussian with (large enough) parameter $S_x = \sigma_x I$, we need to set the covariance matrix for $\vec{p}$ to $\Sigma_p = \sigma_x^2 I - \sigma_z^2 \left(\frac{R}{I}\right)(R|I)$. (This means that σ_x must be sufficiently large relative to σ_z and the singular values of R, so that Σ_p is positive definite.)

4.2 The Gadget Matrix G

The "Gadget matrix" in [18] is based on binary representation, but for our implementation we use a mixed radix representation instead (which makes CRT-based processing easier, see below). Specifically, we use a list of small co-prime factors (in the range from 71 to 181), which we denote here by $p_1, p_2, \ldots, p_k$.

We also have a parameter e that specifies how many times we repeat each factor (by default $e = 3$). Thus our big modulus is $q = \prod_{i=1}^{k} p_i^e$, where we take k large enough to give us as many bits in q as we need.

Given all these co-prime factors, we define the vector $\vec{g}$ as the mixed-radix vector with the p_i's appearing e times each. That is, the first entry is $\vec{g}[0] = 1$, and for each following entry we have $\vec{g}[i+1] = \vec{g}[i] \cdot p_{\lfloor i/e \rfloor}$:

$$\vec{g}^T = \big(1,\ p_1,\ \ldots,\ p_1^{e-1},\ \ p_1^e,\ p_2 p_1^e,\ \ldots,\ p_2^{e-1} p_1^e,\ \ \ldots\ \ P^*,\ p_k P^*,\ \ldots,\ p_k^{e-1} P^*\big).$$

where $P^* = \prod_{i<k} p_i^e$. In other words, the vector $\vec{g}$ has dimension $e \cdot k$, with indexing $i = 0, \ldots, ek-1$, and the ith entry in $\vec{g}$ is obtained by setting $t = \lfloor i/e \rfloor$ and $s = i \bmod e$, and then $\vec{g}[i] = p_{t+1}^s \cdot \prod_{j=1}^{t} p_j^e$. Once $\vec{g}$ is defined, our gadget matrix is just the tensor $G_n = \vec{g}^T \otimes I_n$, whose dimensions are n-by-w for $w = e \cdot k \cdot n$.

4.3 The G-Sampling Routine

Our G-sampling routine is an extension of the power-of-two procedure from [18, Sec. 4.2]. Specifically, we are given as input a syndrome vector $\vec{u} \in \mathbb{Z}_q^n$, in CRT representation, and we need to sample a random small vector $\vec{z} \in \mathbb{Z}^{nek}$ such that $G_n \times \vec{z} = \vec{u} \pmod q$.

Since $G_n = \vec{g}^T \otimes I_n$, each block of ek entries of $\vec{z}$ can be sampled separately. In particular, for each entry u in $\vec{u}$, we sample $\vec{z}' \in \mathbb{Z}^{ek}$ such that $\langle \vec{z}', \vec{g} \rangle = u \pmod q$. It is straightforward to generalize the power-of-two procedure from [18, Sec. 4.2] for choosing $\vec{z}'$ to our mixed-radix representation. The generalized procedure would work as depicted in Figure 4. For Step 2 in this procedure, we sample from the distribution $\mathcal{D}_{p_i\mathbb{Z}+u,\sigma}$ with Gaussian parameter $\sigma = 4 \cdot 181 = 724$, where $p_1 = 181$ is the largest of our small factors.

Parameters: σ, e, k and the p_i's
Input: $u \in \mathbb{Z}$
1. for $i = 0$ to $k-1$, for $j = 0$ to $e-1$
2. $\quad t \leftarrow \mathcal{D}_{p_i\mathbb{Z}+u,\sigma}$ // Sample a σ-short $t \in \mathbb{Z}$, $t = u \pmod{p_i}$
3. $\quad$ Set $\vec{z}'[j + i \cdot e] := t$
4. $\quad$ Set $u := (u - t)/p_i$ // Update u after each iteration
5. Output $\vec{z}'$

Figure 4: The G-sampling procedure

4.3.1 Working with $\vec{u}$ in CRT Representation

One of the main reasons for our use of a modulus of the form $q = \prod_{i=1}^{k} p_i^e$ is to be able to use Chinese-Remainder (CRT) representation for elements in $\mathbb{Z}_q$. Namely, each element $x \in \mathbb{Z}_q$ is represented by a vector $(x_i = x \bmod p_i^e)_{i\in[1,k]}$. Using a CRT representation allows for significantly faster processing.

In our implementation, we are given as input the syndrome vector $\vec{u}$ in the CRT representation, so an entry u in $\vec{u}$ is itself represented by a vector $\vec{v} = (u \bmod p_j^e)_{j\in[1,k]}$. We would like to implement the procedure from Figure 4 without having to convert the entries of $\vec{u}$ from the CRT representation to the standard representation. We stress that while we use CRT encoding for the *input*, the output $\vec{z}'$ consists of only short integers and is returned in binary representation.

Considering the code in Figure 4, most of it can be applied as-is to each coefficient of the CRT representation of u. The only parts that require attention are Step 2, when we need to compute $u \bmod p_i$, and Step 4, when we update $u := (u - t)/p_i$. In Step 2, we see that knowing $v_i = u \pmod{p_i^e}$ is all we need in order to compute $u \bmod p_i$. All of the other components can be ignored for the purpose of that step.

In Step 4, we have two cases: one is how to update the components $v_j = u \pmod{p_j^e}$ for $j \neq i$, and the other is how to update the component $v_i = u \pmod{p_i^e}$. Updating v_j for $j \neq i$ is easy: since p_i and p_j are co-prime we can simply set $v_j := [(v_j - t) \cdot p_i^{-1}]_{p_j^e}$. In our implementation, we pre-compute all the values $p_i^{-1} \bmod p_j^e$ for $i \neq j$, so that updating each v_j takes only one modular subtraction and one modular multiplication.

Updating the component $v_i = u \pmod{p_i^e}$ is a different story. Clearly the new value $(u - t)/p_i \bmod p_i^e$ depends not only on the value of v_i itself but on the values of all the v_j's, so we cannot hope to be able to compute the updated v_i without reconstructing the entire integer u. Luckily, it turns out that in this case we do not need to fully update v_i. To see this, notice that the only reason we need the value of $v_i = u \pmod{p_i^e}$ in this computation is to be able to compute $u \pmod{p_i}$ in Step 2, and we only need to use it e times. After the first time, we can update $v_i := (v_i - t)/p_i$ (treating v_i as an integer and using the fact that $t = u = v_i \pmod{p_i}$ and therefore $v_i - t$ is divisible by p_i). The updated v_i may no longer satisfy $v_i = u \pmod{p_i^e}$, but we do have the guarantee that $v_i = u \pmod{p_i^{e-1}}$, and this is sufficient for us since all we need is to be able to use v_i for computing $u \bmod p_i$. More generally, after the kth time that we use v_i, we update it as $v_i := (v_i - t)/p_i$, and we know that the updated value satisfies $v_i = u \pmod{p_i^{e-k}}$. We also do not need to update any of the v_j's for $j < i$, since we will never need them again in this computation. The resulting code

Parameters: σ, e, k and the p_i's
Input: $\vec{v} \in \mathbb{Z}^k$ // $v[j] = u \bmod p_j^e$
1. for $i = 0$ to $k-1$, for $j = 0$ to $e-1$
2. $t \leftarrow \mathcal{D}_{p_i\mathbb{Z}+v[i],\sigma}$ // Sample a σ-short $t \in \mathbb{Z}$ s.t. $t = v[i] = u \pmod{p_i}$
3. Set $\vec{z}'[j + i \cdot e] := t$
4. for $m = i+1$ to $k-1$ // Update $v[m]$ for $m > i$
5. $v[m] := \left[(v[m] - t) \cdot p_i^{-1}\right]_{p_m^e}$
6. $v[i] := (v[i] - t)/p_i$ // Update $v[i]$ as an integer
7. Output $\vec{z}'$

Figure 5: The G-sampling procedure in CRT representation

for implementing the G-sampling routine with u in CRT representation is given in Figure 5.

Lazy update of the v_j's. Updating each of the v_j's requires modular arithmetic modulo p_j^e, which may mean using some pre-computed tables for this modulus. As written, the code in Figure 5 would need to constantly shuffle these tables into memory, since every iteration requires updating many v_j's modulo many different p_j^e's. We note, however, that we only use the value of v_j when we get to $i = j$ in Step 1, and so in our implementation we delay updating v_j until it is needed and do all the mod-p_j^e operations only then.

4.4 Sampling Gaussian Distributions Over $\mathbb{Z}$

One-dimensional Gaussians. Our current implementation uses the "shortcut" of sampling from the one-dimensional rounded continuous Gaussian distribution instead of the discrete Gaussian distribution. (I.e., instead of $\mathcal{D}_{\mathbb{Z}-x,r}$, we sample from the corresponding one-dimensional continuous Gaussian distribution and round to the nearest $z-x$ for integral z.) This "shortcut" makes our one-dimensional sampling routine faster and easier to implement, but introduces a noticeable variation. We are not aware of any security vulnerability due to this deviation. We also note that any one-dimensional sampling implementation can be plugged in to our code without changing anything else, but this is one aspect that we did not experiment with.

Multi-dimensional ellipsoidal Gaussians. Recall that the perturbation vector $\vec{p}$ in the Micciancio-Peikert procedure is drawn from an ellipsoidal discrete Gaussian distribution with covariance matrix $\Sigma_p = \sigma_x^2 I - \sigma_z^2 \left(\frac{R}{I}\right)\left(R^T | I\right)$. Prior work due to Gentry et al. [13] and Peikert [19] showed how to sample from this distribution by first computing $\sqrt{\Sigma_p}$, a rather expensive operation. Instead, we devised and implemented a sampling procedure for the ellipsoidal discrete Gaussian that is somewhat similar to the GPV sampler in [13] but can work directly with the covariance matrix rather than its square root. Specifically, we choose each entry in $\vec{p}$ from a one-dimensional discrete Gaussian distribution, conditioned on the previous entries. To approximate the conditional mean and variance, we use the corresponding values from the continuous case. Specifically, for a

vector (x, y) with covariance matrix Σ and mean μ we have:

$$\begin{aligned} \Sigma_{Y|X} &= \Sigma_{Y,Y} - \Sigma_{Y,X}\ \Sigma_{X,X}^{-1}\ \Sigma_{X,Y} \\ \mu_{Y|X=x} &= \mu_Y + \Sigma_{Y,X}\ \Sigma_{X,X}^{-1}(x - \mu_X) \end{aligned} \tag{6}$$

where $\Sigma_{X,X}, \Sigma_{X,Y}, \Sigma_{Y,X}, \Sigma_{Y,Y}$ are the four quadrants of the covariance matrix Σ, and μ_X, μ_Y are the two parts of the mean vector μ. (Note that when x is one-dimensional, as in our case, then $\Sigma_{X,X}^{-1}$ is just $1/\sigma_x^2$.) Below we show that this procedure yields the right probability distribution up to a negligible error, as long as the singular values of the matrix Σ_p are all $\omega(\log(\lambda n))$. [5]

Analysis of the ellipsoidal Gaussian sampler. Below we use $\mathcal{D}_{\mathbb{Z}^n,\sqrt{\Sigma}}$ to denote the n-dimensional discrete Gaussian distribution with covariance matrix Σ. For index sets $A, B \subset [n]$, we define $\Sigma_{A,B}$ to be the submatrix obtained by restricting Σ to the rows in A and the columns in B. (When the set only has one element, we simply write, e.g., $\Sigma_{A,i}$ instead of $\Sigma_{A,\{i\}}$.) We will be interested in the kth diagonal entry $\Sigma_{k,k}$, the top-left $(k-1) \times (k-1)$ submatrix $\Sigma_{[k-1],[k-1]}$, and the submatrices $\Sigma_{k,[k-1]}$ and $\Sigma_{[k-1],k}$. (Note that $\Sigma_{k,[k-1]} = \Sigma_{k,[k-1]}^T$, since Σ is positive definite and therefore symmetric.)

We then define

$$\begin{aligned} S_k &:= \Sigma_{k,k} - \Sigma_{k,[k-1]}\Sigma_{[k-1],[k-1]}^{-1}\Sigma_{[k-1],k}\ \text{, and} \\ \vec{v}_k &:= \Sigma_{[k-1],[k-1]}^{-1}\Sigma_{k,[k-1]}\ . \end{aligned} \tag{7}$$

(Note that $S_k \in \mathbb{R}$ is a scalar and $\vec{v}_k \in \mathbb{R}^{k-1}$ is a $(k-1)$-dimensional vector. For convenience, we also define $S_1 := \Sigma_{1,1}$ and $\vec{v}_1 := \vec{0}$.) S_k is known as the Schur complement of $\Sigma_{[k-1],[k-1]}$ in $\Sigma_{[k],[k]}$, a very well-studied object [21]. In particular, the following claim shows that the kth coordinate of a (continuous) Gaussian with covariance Σ conditioned on the first $k-1$ coordinates taking the value $\vec{x}' \in \mathbb{R}^{k-1}$ is exactly the Gaussian with variance S_k and mean $\langle \vec{v}_k, \vec{x}' \rangle$, as we discussed above.

Claim 1. *For any vector $\vec{x} \in \mathbb{R}^n$ and symmetric matrix $\Sigma \in \mathbb{R}^{n \times n}$, let S_n and $\vec{v}_n$ be defined as above, and let $\vec{x}' \in \mathbb{R}^{n-1}$ be the first $n-1$ coordinates of $\vec{x}$. Then, if S_n and $\Sigma_{n,n}$ are non-zero and $\Sigma_{[n-1],[n-1]}$ is invertible, then $\vec{x}^T\Sigma^{-1}\vec{x} = (x_n - \langle \vec{v}_n, \vec{x}' \rangle)^2/S_n + \vec{x}'^T\Sigma_{[n-1],[n-1]}\vec{x}'$.*

Proof. Note that

$$\Sigma^{-1} = \begin{pmatrix} \Sigma_{[n-1],[n-1]}^{-1} + \vec{v}_n\vec{v}_n^T/S_n & -\vec{v}_n/S_n \\ -\vec{v}_n^T/S_n & 1/S_n \end{pmatrix} .$$

(One can check this by simply multiplying by Σ. We note that the identity does not hold when Σ is not symmetric.) The result then follows by directly computing $\vec{x}^T\Sigma^{-1}\vec{x}$. □

We will also need the following well-known fact, which follows immediately from the Poisson summation formula.

[5] The proof below works when the underlying one-dimensional sampling is from a discrete Gaussian, and will incur noticeable deviation when using rounded continuous Gaussian.

Lemma 2. *For any $s \geq 1$ and any $x \in \mathbb{R}$,*

$$(1 - 2^{-s^2}) \cdot \rho_s(\mathbb{Z}) \leq \rho_s(\mathbb{Z} - x) \leq \rho_s(\mathbb{Z}) \ .$$

Here, $\rho_s(x)$ is the one-dimensional Gaussian function with parameter s, and for a set of points X, $\rho_s(X)$ is the sum of ρ_s over these points. We now prove the correctness of our sampler.

Theorem 1. *Consider the following procedure that takes as input a positive-definite matrix $\Sigma \in \mathbb{R}^{n \times n}$.*

1. *Set $\vec{z} \leftarrow \vec{0}$.*
2. *For $k = 1, \ldots, n$, compute S_k and $\vec{v}_k$ as defined above.*
3. *For $k = 1, \ldots, n$, sample z_k from $\mu_k + \mathcal{D}_{\mathbb{Z} - \mu_k, \sqrt{S_k}}$, where $\mu_k := \langle \vec{v}_k, \vec{z} \rangle$, and set $\vec{z} \leftarrow \vec{z} + z_k \cdot \vec{e}_k$.*
4. *Return $\vec{z}$.*

Then, the output vector $\vec{z}$ is within statistical distance $n2^{-S}$ of $\mathcal{D}_{\mathbb{Z}, \sqrt{\Sigma}}$, where $S := \min_k S_k$.

Proof. Note that the first coordinate of $\vec{z}$ is distributed exactly as $\mathcal{D}_{\mathbb{Z}, \sqrt{S_1}} = \mathcal{D}_{\mathbb{Z}, \sqrt{\Sigma_{1,1}}}$. Let $\Sigma' := \Sigma_{[k-1],[k-1]}$. We assume for induction that the first $k-1$ coordinates of $\vec{z}$, which we call $\vec{z}'$, are within statistical distance $(k-1) \cdot 2^{-S}$ of $\mathcal{D}_{\mathbb{Z}^{k-1}, \sqrt{\Sigma'}}$. If this were exactly the distribution of $\vec{z}'$, then for any fixed integer vector $\vec{y} \in \mathbb{Z}^k$ with first k coordinates $\vec{y}'$, we would have

$$\Pr[\vec{z}' = \vec{y}' \text{ and } z_k = y_k] = \frac{\rho_{\sqrt{\Sigma'}}(\vec{y}')}{\rho_{\sqrt{\Sigma'}}(\mathbb{Z}^{k-1})} \cdot \frac{\rho_{\sqrt{S_k}}(y_k - \mu_k)}{\rho_{\sqrt{S_k}}(\mathbb{Z} - \mu_k)} \ .$$

Here, the ρ functions are multi-dimensional Gaussians. By Claim 1, we have $\rho_{\sqrt{\Sigma'}}(\vec{y}') \cdot \rho_{\sqrt{S_k}}(y_k - \mu_k) = \rho_{\sqrt{\Sigma_{[k],[k]}}}(\vec{y})$. And, by Lemma 2, we have that

$$(1 - 2^{-S_k}) \cdot \rho_{\sqrt{S_k}}(\mathbb{Z}) \leq \rho_{\sqrt{S_k}}(\mathbb{Z} - \mu_k) \leq \rho_{\sqrt{S_k}}(\mathbb{Z}) \ .$$

Therefore, $\Pr[\vec{z}' = \vec{y}' \text{ and } z_k = y_k]$ is within a factor of $1 - 2^{-S_k}$ of $\rho_{\sqrt{\Sigma_{[k],[k]}}}(\vec{y}) / \rho_{\sqrt{\Sigma_{[k],[k]}}}(\mathbb{Z}^k)$. It follows that the real distribution of the first k coordinates of $\vec{z}$ is within statistical distance $(k-1)2^{-S} + 2^{-S_k} \leq k2^{-S}$ of $\mathcal{D}_{\mathbb{Z}^k, \sqrt{\Sigma_{[k],[k]}}}$, as needed. □

Finally, we note that we can relate the Schur complement to the eigenvalues of Σ, which makes it easier to compare the performance of our sampler with prior work, such as [13, 19].

Lemma 3. *[17, Corollary 2.4] For any positive-definite matrix $\Sigma \in \mathbb{R}^{n \times n}$, $\sigma_n(\Sigma) \leq S_k \leq \sigma_1(\Sigma)$ for all $k = 1, \ldots, n$, where σ_i is the ith largest eigenvalue of Σ.*

Corollary 4. *The output of the procedure described in Theorem 1 is within statistical distance $n2^{-\sigma_n(\Sigma)}$ of $\mathcal{D}_{\mathbb{Z}^n, \sqrt{\Sigma}}$, where $\sigma_n(\Sigma)$ is the smallest eigenvalue of Σ.*

4.5 Using a Stash of Samples

Our G-Sampling Routine (see Section 4.3) requires us to sample from $D_{p\mathbb{Z}+u,\sigma}$ for a factor p taken from the short list in Section 4.2 and an input integer u. For the small factors p that we use (all between 71 and 181), this is done by repeatedly sampling short integers from $\mathcal{D}_{\mathbb{Z},\sigma}$ until we hit one that satisfies $t = u \pmod{p}$. A naive implementation of this procedure will need to sample $p/2 > 50$ times (on the average) before it hits a suitable t, making for a very slow implementation. In our tests, this naive implementation spent about 30% of the obfuscation time sampling 1D Gaussians.

To do better, we keep for each factor p a stash of unused samples. Whenever we need a new sample we first check the stash. Only if there are no hits in the stash do we sample new points, and all the points which are *not equal* to u modulo p are then stored in the stash for future use. (This is similar to the "bucketing" approach described in [18, Sec. 4.1], but without the online/offline distinction.)

The stash itself is implemented as a simple size-p array of integers, where $\mathsf{stash}_p[i]$ contains the latest sample that was equal to i modulo p (if that sample was not yet used). An alternative approach would be to keep for each entry i a queue of *all sample values* satisfying $x = i \pmod{p}$, but such an implementation would be considerably more complex.

It is easy to see that the use of stash does not bias the distribution from which we sample: by definition each non-empty entry j contains a random element $x \leftarrow \mathcal{D}$, constrained only by $x = j \pmod{p_i}$. (Note that we use a separate stash for each factor p_i.)

As we show now, that simple stash implementation already reduces the required number of trials per sample from $p/2$ to $\approx \sqrt{2p}$, reducing the sampling time from 30% to about 3% of the total running time. To see how the simple implementation above reduces the overhead, denote by f the fraction of full entries in the stash just prior to a sample operation. The *expected change* in the number of full entries after the sample operation (where $u \bmod p$ is uniform in $\mathbb{Z}_p$) is described by the formula

$$E[ch] = f \cdot (-1) + (1-f) \cdot ((1-f) \cdot p - 1)/2 = \frac{1}{2} \cdot ((1-f)^2 \cdot p - 1 - f) \ ,$$

where the second term follows from the fact that each *empty entry other than i* has $1/2$ probability of being filled before we sample a match. Assuming that the sampler reaches a steady state (with expected change equal to zero), the value of $f \in [0,1]$ is

$$f = \frac{1 + 2p - \sqrt{1+8p}}{2p} = 1 - \sqrt{2/p} + \Theta(1/p) \ .$$

The expected work per sampling operation is therefore $f \cdot 1 + (1-f) \cdot p \approx \sqrt{2p}$.

Thread-safety of our stash implementation. One important reason that we chose to implement the stash using the simple procedure above (rather than implementing a full queue per entry) is that it is easier to make it thread-safe. Implementing a queue per entry would require the use of semaphores to coordinate between the threads, whereas having only one integer per entry lets us use simple atomic test-and-set operations (implemented via the `C++11`'s `atomic<int>` type). Since the simple implementation already reduced the overhead to about 3%, we did not attempt the more complicated one.

5 Setting the Parameters

In setting the parameters, we try to minimize the dimension m and the bit size of the modulus q, subject to functionality and security. For security, we need the dimensions $m, \bar{m}$ to be large enough relative to q, so that the relevant computational problems are hard ($\bar{m}$ is the number of rows in the trapdoor matrix R). For functionality, we need q to be sufficiently larger than the largest noise component that we get, and we also need the large dimension m to be sufficiently large (relative to $n \log q$) for our trapdoor sampling procedure. These security and functionality considerations imply a set of constraints, described in Equations (10) through (13) below, and our implementation searches for the smallest values that satisfy all these constraints.

5.1 Functionality

Basic facts. We use the bound from [18, Lemma 2.9] on the singular values of random matrices. Namely, if the entries of an a-by-b matrix X are chosen from a Gaussian with parameter σ, then the largest singular value of X (denoted $s(X)$) is bounded by

$$\Pr\left[s(X) > \mathsf{const} \cdot (\sqrt{a} + \sqrt{b} + t)\right] < 2^{-t^2} \tag{8}$$

for some absolute constant const, and our empirical results show that we can use $\mathsf{const} = 1$. Below we use Eqn. (8) also in situations where the entries of X are not independent, such as when every column of X is chosen using the trapdoor sampling procedure from Section 4.1. Namely, if C is an m-by-m encoding matrix (before the "outer transformation") with the columns chosen from a Gaussian with parameter σ_x (over some coset in $\mathbb{Z}^m$), then we heuristically use $2\sqrt{m}$ to estimate its largest singular value.

The parameter r. The starting point of our parameter setting is the smallest Gaussian parameter that we use for sampling the trapdoor R over $\mathbb{Z}$. This choice ripples through the entire implementation and has dramatic effect on efficiency, so we would like to make it as small as we possibly could. Following Micciancio-Peikert [18], we set this parameter to $r = 4$ (which also seems large enough to defeat the Arora-Ge attacks [5]).

Output size of trapdoor sampling. Next consider the size of vectors that are output by the trapdoor sampling procedure. The trapdoor sampling procedure samples a solution $\vec{x}$ to $A\vec{x} = \vec{u} \pmod q$, according to a spherical Gaussian distribution with some parameter σ_x, and we would like to make σ_x as small as we can. The procedure works by sampling a perturbation vector $\vec{p}$ and a solution $\vec{z}$ to $G\vec{z} = \vec{u} - A\vec{p} \pmod q$, then outputting $\vec{x} = \vec{p} + \left(\frac{R}{I}\right)\vec{z} \bmod q$. The vector $\vec{z}$ is drawn from a spherical Gaussian with parameter σ_z (to be determined shortly), and the perturbation $\vec{p}$ has covariance $\Sigma_p = \sigma_x^2 I - \sigma_z^2 \left(\frac{R}{I}\right)\left(R^t | I\right)$. Hence the overriding constraint in setting the parameters for this procedure is to ensure that Σ_p is positive definite, which we can do by setting $\sigma_x^2 > \sigma_z^2(s+1)^2 \approx (\sigma_z s)^2$, where s is an upper bound (whp) on the largest singular value of R.

To determine the value of σ_z that we can get, recall that each entry in $\vec{z}$ is drawn from a Gaussian $\mathcal{D}_{v+p_i\mathbb{Z},\sigma_z}$, where p_i is one of our small co-prime factors that are listed in Section 4.2 (so in particular $p_i \leq 181$). To be able to sample efficiently from this distribution, it is sufficient to set

$$\sigma_z = r \cdot \max_i(p_i) = 4 \cdot 181 = 724 \tag{9}$$

Using Eqn. (8), the largest singular value of our $\bar{m}$-by-w matrix R (with entries chosen with Gaussian parameter r) is bounded whp by $s = r \cdot (\sqrt{\bar{m}} + \sqrt{w} + 6)$ (to get 2^{-36} error probability). Hence to ensure $\sigma_x > \sigma_z \cdot s$ it is enough to set

$$\sigma_x > (r \cdot \max_i(p_i)) \cdot (r \cdot (\sqrt{\bar{m}} + \sqrt{w} + 6)) \approx 2900 \cdot (\sqrt{\bar{m}} + \sqrt{w} + 6). \tag{10}$$

As usual when setting parameters for lattice-based system, there is some weak circularity here since $\bar{m}, w$ depend on the size of q, which in turn depends on the output size our sampling procedure, that depends on σ_x. But this circular dependence is very weak, and it is easy to find a solution that satisfies all the constraints. For example, in our largest setting $L = 20$ we have $\bar{m} \approx 6000$ and $w \approx 8000$, for which the above bound yields $\sigma_x \approx 2^{18.9}$.

The modulus q. The vectors $\vec{x}$ that are output by the trapdoor sampling procedure (which are drawn from a spherical Gaussian with parameter σ_x over some coset in $\mathbb{Z}^m$) form the columns of the GGH15 encoding matrices C before the outer transformation of the GGH15 "safeguards". As explained in Section 3.1, the noise term when we multiply L encodings is

$$\mathsf{noise} = \sum_{j=1}^{L} \Big(\prod_{i=1}^{j-1} M_i\Big) P_{j-1} E_j \Big(\prod_{i=j+1}^{L} C_i\Big)$$

where the M_i's are the "plaintext matrices" that we encode, the P_i's are the inner transformation matrices used in the GGH15 "safeguards", the E_i's are the error matrices that we choose, and the C_i's obtained using our trapdoor sampling procedure. Since the C_i's are much larger than the other matrices in this expression, the only relevant term in this sum is the first one, namely $E_1 \times \prod_{i=2}^{L} C_i$.

Below we use the largest singular value of the matrix product $E_1 \times \prod_{i=2}^{L} C_i$ to represent its "size". By Eqn. (8) the singular values of all the C_i's are bounded by $\sigma_x(2\sqrt{m}+6) \approx 2\sigma_x\sqrt{m}$, and that of E is bounded by $2^7(\sqrt{m} + \sqrt{n} + 6) \approx 2^7\sqrt{m}$. (Each entry of E is chosen from a Gaussian with parameter 2^7.) Therefore we can heuristically bound the largest singular value of the product by $2^7 \cdot 2^{L-1} \cdot m^{L/2} \cdot \sigma_x^{L-1}$. For our zero-test we check that the noise is no more than $q/2^{10}$, so we need q to be 2^{10} times larger than this bound, or in other words:

$$\log_2 q \geq 7 + \log_2 \sigma_x \cdot (L-1) + \log_2 m \cdot L/2 + (L-1) + 10. \tag{11}$$

Once we have a bound on q we choose the number k of co-prime factors so that the product $\prod_{i=1}^{k} p_i^e$ exceeds that bound. The parameter e depends on the hardware architecture: For performance reasons we always use $e = 3$ when running on a platform with Intel AVX (so all the p_i^e factors are less than 23 bits long, see Section 6), and on platforms without Intel AVX we use $e = 8$ (so the p_i^e's are just under 60 bits long).

For our largest parameters (with $L = 20$ and $\sigma_x \approx 2^{18.9}$) we need to set $m \approx 2^{13.8}$ for security breasons (see below). Hence we set $\log_2 q \geq 7 + 18.9 \cdot 19 + 13.8 \cdot 10 + 29 \approx 535$, and with $e = 3$ we need $k = 26$ co-prime factors.

The large dimension m. To be able to generate trapdoors, we must also ensure that the parameters m (number of columns in A) is large enough. Specifically, for a given lower bound $\bar{m}$ on the number of columns in $\bar{A}$ (obtained by security considerations), and given the parameters k (number of co-prime factors), e (number of times each factor repeats — either 3 or 8), and n (the

dimension of "plaintext matrices" to encode), we need to ensure that $m \geq nke + \bar{m}$. (In all cases, the bound that we get on m due to security considerations was larger than this functionality-based bound.)

5.2 Security

The trapdoor dimension $\bar{m}$. Recall that trapdoor generation chooses a uniform $\bar{A}$ and small R, then sets $A = [\bar{A}|G - \bar{A}R] \bmod q$ (so that $A \times \left(\frac{R}{I}\right) = G \pmod q$). We would like A to be random, so we need to argue that $G - \bar{A}R$ is nearly uniform, even conditioned on $\bar{A}$. This is typically done by appealing to the leftover hash lemma, but doing so requires that each column of R has more than $n \log_2 q$ bits of min-entropy. In our implementation the entries of R have constant magnitude, so to use the leftover hash lemma we need R to have at least $\Omega(n \log_2 q)$ rows (and of course $\bar{A}$ must have the same number of columns).

Micciancio and Peikert observed in [18] that we can get by with lower-dimension R if we are willing to have A *pseudorandom* (under LWE) rather than random. Splitting $\bar{A}$ into two parts $\bar{A} = [\bar{A}_1|\bar{A}_2]$, and denoting the corresponding two parts of R by $R = \left(\frac{R_1}{R_2}\right)$ (and assuming that $\bar{A}_2$ is square and invertible), we have that

$$AR = A_1R_1 + A_2R_2 = \bar{A}_2 \underbrace{\left((\bar{A}_2^{-1}\bar{A}_1)R_1 + R_2\right)}_{=A'R_1+R_2},$$

which is pseudorandom under LWE. Using this argument requires that R be chosen from the LWE error distribution. Our implementation therefore relies on the assumption that LWE is hard even when the error distribution uses Gaussian parameter $r = 4$.

Moreover, this LWE-based assumption also requires that $\bar{m}$ is large enough so we can use $\bar{m} - n$ as the "security parameter" for LWE. (Recall that $\bar{m}$ is the number of columns in $\bar{A}$.) In our setting of parameters, we in particular must set $\bar{m}$ large enough so given $\bar{A}$ and $A' = \bar{A}R$ it will be hard to find R. Finding each column of R is an instance of the small-integer-solution (SIS) problem in dimensions n-by-$\bar{m}$ over $\mathbb{Z}_q$, so we must set $\bar{m}$ large enough to get the level of security that we want. Setting the dimension to get SIS-security is not entirely straightforward and typically involves consulting some tables [1], but in the range that we consider (with $\lambda \in [80, 256]$) it behaves very roughly as[6]

$$\bar{m} \geq (\sqrt{\lambda} + 2) \cdot \sqrt{n \log_2 q}. \tag{12}$$

With $n \approx 100$ (for NFAs with 100 states) and $\log_2 q \approx 535$, this means that we must ensure $\bar{m} \geq (\sqrt{\lambda} + 2) \cdot \sqrt{100 \cdot 535} \approx 2545$.

The large dimension m. We use the formula from [12, Appendix C] to relate the lattice dimension m in the LWE instances to the desired security level.[7] Namely to get security level λ we need the dimension m to be at least $m \geq \log_2(q/\text{fresh-noise-magnitude}) \cdot (\lambda + 110)/7.2$. The fresh noise magnitude does not have a crucial impact on our parameters, so we can choose it rather large (e.g., Gaussian of parameter $\approx 2^7$). Thus we get the constraint

$$m \geq (\log_2 q - 7)(\lambda + 110)/7.2\ . \tag{13}$$

[6] This yields root Hermite factors of $\delta \approx 1.006$ for $\lambda = 80$, $\delta \approx 1.0044$ for $\lambda = 128$, and $\delta \approx 1.0023$ for $\lambda = 256$.

[7] That formula is somewhat out of vogue these days, and should really be replaced by more refined analyses such as [3], but it still gives reasonable values.

With $\log_2 q \approx 535$ and $\lambda = 80$, this yields $m \geq (535 - 7) \cdot (80 + 110)/7.2 \approx 2^{13.8}$.

5.3 Putting it together

Given the desired program length L, the dimension of plaintext matrices n (which depends on the number of states in the branching program), the parameter e (which depends on the hardware platform), and the security parameter λ, our implementation tries to find the smallest number k of co-prime factors that satisfies all the constraints above. Trying $k = 1, 2, 3, \ldots$, we set $q = \prod_{i<k} p_i^e$, then set m using Eqn. (13), compute $w = nek$ and $\bar{m} = m - w$, and verify that this value of $\bar{m}$ satisfies Eqn. (12). Next we compute σ_x using Eqn. (10), and finally check if the value of q satisfies the functionality constraint from Eqn. (11). Some example parameter values that we used in our tests can be found in Table 1.

L :	5	8	10	12	14	17	20
σ_x :	$2^{18.0}$	$2^{18.3}$	$2^{18.4}$	$2^{18.6}$	$2^{18.7}$	$2^{18.8}$	$2^{18.9}$
k :	6	10	12	15	18	22	26
$\log_2 q$:	133	219	261	322	382	458	542
$\bar{m}$:	1462	2471	2950	3614	4253	4998	5955
m :	3352	5621	6730	8339	9923	11928	14145

Table 1: Parameters in our tests, security $\lambda = 80$, plaintext dimension $n = 105$

6 Efficient matrix arithmetic

The majority of the obfuscation time is spent doing matrix arithmetic, either modulo small integers or over single-precision integers (or floating point numbers). We first discuss the modular matrix arithmetic.

6.1 Matrix multiplication in $\mathbb{Z}_t$

As discussed in Section 4.3, we use a CRT representation of $\mathbb{Z}_q$, where q is the product of small co-prime factors. The parameters are typically chosen so that each factor has a bit-length at most 23 (or at most 60), the reason for this will be explained shortly.

So assume that we are working modulo a small number t of bit-length at most 23 bits. The two main operations of concern are large matrix multiplication and inversion over $\mathbb{Z}_t$, where the dimensions of the matrices are measured in the thousands. For matrix inversion, we assume that t is a prime power. Consider computing the product $C = AB$, where A and B are large matrices over $\mathbb{Z}_t$.

Cache friendly memory access. To obtain cache friendly code, all the matrices are organized into *panels*, which are matrices with many rows but only 32 columns. We compute the ith panel of C by computing AB_i, where B_i is ith panel of B. If multiple cores are available, we use them to parallelize the computation, as the panels of C can be computed independently.

Next consider the computation of AP, where P is a single panel. We can write $AP = \sum_j A_j P_j$, where each A_j is a panel of A and each P_j is a 32×32 square sub-matrix of P. We thus reduced the problem to that of computing

$$Q \leftarrow Q + RS, \tag{14}$$

where Q and R are panels, and S is a 32×32 square matrix. The matrix S is small and fits into the first-level cache on most machines — that is why we chose a panel size of 32. While the panels Q and R typically do not fit into the first-level cache, the data in each panel is laid out in contiguous memory in row-major order. In the implementation of Eqn. (14), we process the panels a few rows at a time, so the data in each panel gets processed sequentially, and we rely on hardware prefetch (which is common on modern high-performance systems) to speed up the memory access. Indeed, the computation of Eqn. (14) can be reduced to operations of the form

$$u \leftarrow u + vS, \tag{15}$$

where u consists of a few rows of Q and v consists of a few rows of R. While we may need to fetch u and v from a slower cache, the hardware prefetcher should help a bit, and, more significantly, these slower fetches are paired with a CPU-intensive computation (involving S, which is in the first-level cache), so the resulting code is fairly cache friendly.

Fast modular arithmetic. The basic arithmetic operation in any matrix multiplication algorithm is the computation of the form $x \leftarrow x + yz$, where x and y are scalars. In our case, the scalars lie in $\mathbb{Z}_t$. Unfortunately, modern CPUs do not provide very good direct hardware support for arithmetic in $\mathbb{Z}_t$. However, by restricting t to 23 bits, we can use the underlying floating point hardware that is commonly available and typically very fast. Indeed, if we have 23-bit numbers w and x_i and y_i, for $i = 1, \ldots, k$, then we can compute $w + \sum_i x_i y_i$ exactly in floating point, provided k is not too big: since standard (double precision) floating point can exactly represent 53-bit integers, we can take k up to $2^{53-23 \cdot 2} = 2^7$. If k is larger than this, we can still use the fast floating point hardware, interspersed with occasional "clean up" operations which convert the accumulated floating point sum to an integer, reduce it mod t, and then convert it back to floating point.

Using AVX instructions. By using this floating point implementation, we can also exploit the fact that modern x86 CPUs come equipped with very fast SIMD instructions for quickly performing several floating point operations concurrently. Our code is geared to Intel's AVX (and AVX2) instruction set, which allows us to process floating points operations 4 at a time (the next-generation AVX512 instruction set will allow us to process 8 at a time).

The core of our matrix arithmetic code is a routine that computes Eqn. (15) in floating point using Intel's AVX (and AVX2) instructions. Suppose u and v are single rows. If the ith entry of v is v_i and the ith row of S is S_i (for $i = 1, \ldots, 32$), then the computation of Eqn. (15) can be organized as $u \leftarrow u + \sum_i v_i S_i$.

To carry this out, we load the vector u from memory into eight AVX registers, into which we will accumulate the result, and then store it back to memory. To accumulate the result, for $i = 1, \ldots, 32$, we do the following:

- Use the AVX "broadcast" instruction to initialize an AVX register r with 4 copies of v_i.

- Load the values of S_i four at a time into an AVX register, multiply by the register r, and add the result into the corresponding accumulator register. (For AVX2, we use a fused multiply-add instruction, which saves the use of an instruction and a temporary register.)

That is the simplest implementation, but not necessarily the fastest. We experimented with a number of different implementations. In our actual implementation, we process 2 or 3 rows of Q and R at a time in Eqn. (14), so that u and v in Eqn. (15) consist of either 2 or 3 rows. (The choice depends on whether we have a fused multiply-add instruction, as without it, we run out of AVX registers more quickly.) With this strategy, values loaded from S into AVX registers can participate in several arithmetic operations instead of just one — while loads from S are fast (it is in first-level cache), they are still not as fast as direct register access.

6.2 Matrix inversion in $\mathbb{Z}_t$

Let A be a high-dimension square matrix over $\mathbb{Z}_t$. We perform an "in place" Gaussian elimination, performing elementary row operations on A until we get the identity matrix, but we store the entries of the inverse in A itself. Our algorithm works when t is a prime or a prime power.[8] This is easy to do: when selecting a pivot, instead of choosing any non-zero pivot, we always choose an invertible pivot modulo t.

Just as for multiplication, we organize A into panels. In carrying out Gaussian elimination, we start with the first panel, and we carry out the algorithm just on this panel, ignoring all the rest. After this, we perform a series of panel/square operations, as in Eqn. (14) to perform the same elementary row operations on the remaining panels of A that were performed on the first panel (if any rows in the first panel were swapped, we first perform those same swaps of the remaining panels before performing the panel/square operation). If multiple cores are available, these panel/square operations can be done in parallel. After we finish with the first panel, we move on to the second panel, and so on, until we are done with the whole matrix. We use the same floating point strategy for arithmetic in $\mathbb{Z}_t$ as we did above, exploiting AVX instructions, if available.

6.3 Integration into NTL

Our new matrix arithemtic has been integrated into NTL (see `http://www.shoup.net/ntl/`), which has an interface that supports matrix arithmetic modulo small numbers p that "fit" into a machine word. On 64-bit machines, the bit length of the modulus p may go up to 60 (or 62, with a special compilation flag).

For p up to 23 bits, the strategy outlined above is used. For larger p, the code reverts to using scalar integer instructions (rather than the AVX floating-point instructions), but still uses the same "cache friendly" panel/square memory organization, and utilizing multiple cores, if available.

Besides matrix multiplication and inverse, the same strategies are used for general Gaussian elimination, and image and kernel calculations.

6.4 Multi-dimensional Gaussian sampling

In Section 4.4, we sketched our basic strategy for sampling from Gaussian distributions. As discussed in that section, to sample from a multi-dimensional Gaussian distribution, we need to

[8] For a composite t, it can fail even if A is invertible modulo t.

compute the conditional mean and covariance as in Eqn. (6). It turns out that this computation is very similar in structure to Gaussian elimination (over floating point numbers). As such, we were able to easily re-purpose our AVX-enabled floating-point code for Gaussian elimination, discussed above, to significantly improve the performance of this computation. This resulted in a roughly $10\times$ speedup over a straightforward implementation (of the same algorithm) of the mean and covariance computation.

6.5 Experimental results for matrix arithmetic

Our testing was done on a machine with Intel Xeon CPU, E5-2698 v3 @2.30GHz (which is a Haswell processor), featuring 32 cores and 250GB of main memory. The compiler was GCC version 4.8.5, and we used NTL version 10.3.0 and GMP version 6.0 (see `http://gmplib.org`).

We compared NTL's new matrix multiplication and inverse code to the current versions of FFLAS, (see `http://linbox-team.github.io/fflas-ffpack/`) and FLINT (see `http://www.flintlib.org/`).

- FFLAS refers to the current version of FFLAS (version 2.2.2, available at `http://linbox-team.github.io/fflas-ffpack/`). FFLAS stands for "Finite Field Linear Algebra Subprograms", and provides an interface analogous to the well-known BLAS interface for linear algebra over floating point numbers. Roughly speaking, FFLAS works by reducing all linear algebraic operations over $\mathbb{Z}_t$ to matrix multiplication over floating point numbers, and then uses a BLAS implementation for the latter. In our tests, we use the BLAS implementation OpenBLAS (see `http://www.openblas.net/`), which is recommended by the FFLAS authors. OpenBLAS itself is highly optimized for many different architectures. We configured and built OpenBLAS so that it was optimized for our Haswell architecture.

  For small t, this floating point strategy has some similarities to NTL's strategy, although NTL implements everything directly (in particular, NTL does not use BLAS and is not so well optimized for anything other than AVX-enabled x86 CPUs).

  For larger t, FFLAS uses a Chinese remaindering strategy, while NTL does not (for larger, but still word-sized t, NTL uses scalar integer arithmetic, as discussed above).

- FLINT refers to the current version of FLINT (version 2.5.2, available at `http://www.flintlib.org/`). FLINT stands for "Fast Library for Number Theory". FLINT only uses scalar integer arithmetic — it does not use any floating point. For matrix multiplication, it uses Strassen's recursive algorithm. This gives slightly better asymptotic complexity, and more importantly it yields much more cache-friendly code. Matrix inversion is implemented by a reduction to matrix multiplication.

We were able to compare both single-threaded and multi-threaded performance of NTL's and FFLAS's multiplication routines, as described in Table 2. FLINT's multiplication does not exploit multiple threads. Neither FFLAS's nor FLINT's inversion routine exploit multiple threads. We can see that NTL's multiplication is a bit slower than FFLAS's, while NTL's inversion is a bit faster. Both NTL and FFLAS are several times faster than FLINT.

We also mention here that for matrix inversion, both FFLAS and FLINT require that the modulus t be a prime number, but the modulus in our application is a prime power rather han a prime. NTL's inverse routine directly supports prime-power moduli, with no extra computational

	$n = 4096$			$n = 8192$		
threads	1	4	16	1	4	16
NTL	3.7	1.28	0.52	26.4	9.49	3.12
FFLAS	3.1	0.89	0.52	21.9	6.62	3.65
FLINT	15.6			111.8		

(a) Multiplication

	$n = 4096$			$n = 8192$		
threads	1	4	16	1	4	16
NTL	4.8	1.78	1.27	37.9	11.8	7.6
FFLAS	6.7			43.2		
FLINT	32.6			219.9		

(b) Inversion

Table 2: Time (in seconds) for multiplication and inversion of $n \times n$ matrices modulo a 20-bit prime

	$n = 4096$		$n = 8192$	
# bits	20	60	20	60
NTL	3.7	27.7	26.4	194.5
FLINT	15.6	30.6	111.8	214.8
NTL (old)		125.2		1000.9

(a) Multiplication

	$n = 4096$		$n = 8192$	
# bits	20	60	20	60
NTL	4.8	45.0	37.9	354.5
FLINT	32.6	57.7	219.9	402.5
NTL (old)		333.2		2776.3

(b) Inversion

Table 3: Time (in seconds) for multiplication and inversion of $n \times n$ matrices modulo 20- and 60-bit primes

cost. In contrast, using FFLAS or FLINT directly would require some type of Hensel lifting to go from prime to prime-power, which would significantly increase the cost of the inverse operation.

Table 3 gives some timing data for matrix multiplication and inversion over $\mathbb{Z}_t$ for different sized moduli t. Data for 20-bit and 60-bit t is presented. We compared NTL and FLINT, along with an old version of NTL. The old version of NTL uses only scalar integer operations, and is very naive and very cache-unfriendly; also, its performance is not sensitive to the size of t, so we only collected data for 60-bit t. For 20-bit t, the current version of NTL is about $35\times$ faster than the old version for matrix multiplication, and about $70\times$ faster for inversion. Looking at NTL vs FLINT, we see that for 60-bit t, the times are pretty close; as we already saw, for 20-bit t, NTL's floating point strategy gives it a huge advantage over FLINT.

7 Implementation Details

7.1 Initialization, Obfuscation, and Evaluation

Our program includes three main stages: *initialization*, which generates the public and secret keys for the encoding scheme, *obfuscation*, which uses the secret key to obfuscate a given branching program, and *evaluation*, which computes the obfuscated program value on a given input (using the public key). After each stage we write the results to disk, then read them back as needed in the next stages.

Initialization. This is where we choose the public and secret keys. As described in Section 3, this includes choosing random node matrices A_i with their trapdoors R_i, and also the "inner transformation matrix" $P_i \in \mathbb{Z}^{n \times n}$ and "outer transformation matrix" $T_i \in \mathbb{Z}_q^{m \times m}$.

Once R_i and A_i are chosen, we compute the perturbation covariance matrix $\Sigma_p = \sigma_x I - \sigma_z \left(\frac{R}{I}\right)(R|I)$ as per Section 4.1 (with σ_x and σ_z as derived in Section 5), then compute the con-

ditional covariance matrices as per Eqn. (6) in Section 4.4 (and we optimize it as in Section 6.4). We also compute the modular inverses of the transformation matrices, namely P^{-1} and T^{-1}. (See Section 6.2.)

Since keeping all these matrices in memory consumes a lot of RAM (especially T and T^{-1}), our initialization phase processes one node at a time, writing all the matrices to disk as soon as it computes them and before moving to the next node.

Obfuscation. Given a branching program to obfuscate, we first randomize it as described in Section 2, where for each matrix in the program we generate a pair of higher-dimension "real" and "dummy" matrices. We then use the trapdoors and transformation matrices that we computed to encode the resulting pairs of matrices. The most expensive parts of this stage are the trapdoor sampling and the multiplication by the transformation matrices T and T^{-1}, all of which are part of the GGH15 encoding procedure.

Here too, we need to conserve memory, and so we only keep in RAM one or two GGH15 nodes at a time. As the real and dummy matrices in each pair are encoded relative to different edges, we cannot encode them together. Hence, we first generate all the $|\Sigma|$ real/dummy pairs for the current input symbol and keep them all in memory. (These matrices only take little memory.) We then read from disk the edge on the "real" path and encode the "real" matrices from all the pairs. Finally, we read the edge on the "dummy" path and encode the "dummy"" matrices from all the pairs.

Evaluation. Once we have all the encoded matrices written on disk and we are given the input string, we simply read the corresponding matrices from disk, multiply them, subtract the "dummy" from the "real" product and check for smallness. One important consideration here is the order in which to multiply the matrices. Recall from Section 3.2 that encodings relative to the sink-bound edges consist of a single vector. So, it is much better to begin with these matrices and then multiply backwards. In this way, we only need to do matrix-vector products as we accumulate the multipliers, rather than full matrix-matrix products. This optimization is one reason why evaluation is many orders of magnitude faster than obfuscation. (Another reason is that we only need to process two matrices per step when evaluating, while during obfuscation we had to generate $2|\Sigma|$ matrices per step.)

7.2 Parallelization Strategies

We implemented and tested various multi-threading strategies, trying to parallelize the computation at different levels. Below we describe these different strategies, focusing mostly on the initialization and obfuscation stages (which are much more expensive). We briefly touch on parallelism during the evaluation stage at the end.

7.2.1 Parallelism Across Different Nodes

The easiest strategy to implement is a high-level strategy in which all of the nodes of the graph are processed in parallel. This is trivial to implement, as the computation at each node is independent of all other nodes. For small parameters, this strategy works quite well. Unfortunately, it does not scale very well, as it requires the data for many nodes to be in memory at the same time. We found that this strategy quickly consumed all available RAM, and ultimately had to be abandoned. Instead, we opted to process the nodes in the graph sequentially, and parallelized computations inside each node, as described below.

7.2.2 Trapdoor Sampling

As discussed in Section 3, when encoding a matrix M w.r.t. edge $i \to j$, we choose a low-norm E and compute $B = [MA_j + E]_q$, then use trapdoor sampling to find a small norm matrix C such that $A_i C = B \pmod q$. This trapdoor sampling routine samples each column of C separately, by invoking the trapdoor sampling procedure from Section 4.1 to solve $A_i C_k = B_k \pmod q$ (with B_k the k'th column of B and C_k the corresponding column of C). In our implementation we therefore parallelize across the different columns, sampling together as many of these columns as we have threads. As discussed in Section 4.5, we used a stash to speed up the computation, and we implemented the stash in a thread-safe manner so that it could be shared between the threads.

We note that it is also possible to parallelize trapdoor sampling at a lower level: specifically the procedure for solving $A\vec{c} = \vec{b}$ involves solving $G\vec{z} = \vec{u}$ with G the "gadget matrix". Due to the structure of G, we can sample the entries of $\vec{z}$ in batches of size ek, where all the batches can be processed independently. Although we did not test it, we expect this strategy to perform worse than parallelism across the different columns.

7.2.3 Gaussian Sampling

As discussed in Section 4.4, during initialization we compute the conditional mean and covariance matrices as in Eqn. (6). This computation essentially has the same structure as standard Gaussian elimination, and we implemented a parallel version of it as described in Section 6.4.

7.2.4 CRT-level Parallelism

A significant amount of time is spent performing matrix multiplication and inversion operations over $\mathbb{Z}_q$. Since q is chosen to be the product of small co-prime factors and the matrices represented using Chinese remaindering, these matrix operations can be performed independently modulo each small factor q_i.

7.2.5 Lower-level Parallelism

We also implemented multi-threaded versions of matrix multiplication and inversion modulo each of the small factors in our CRT base. However, we found empirically that it was more effective not to parallelize at this lowest level, but rather at the higher CRT level.

7.2.6 Disk I/O Pipelining

Each of the three stages (initialize, obfuscate, evaluate) reads its inputs from disk and writes its output to disk. The amount of data transferred between main memory and disk is huge, and we found that a significant amount of time was just spent waiting for disk I/O operations to complete. The problem was only made worse as the multi-threading strategy reduced the computation time relative to the I/O time. To mitigate this problem, we used a multi-threaded "pipelining" strategy. One thread was dedicated to reading from disk, one thread was dedicated to writing to disk, and the remaining threads are used for the actual computations. In this way, while the next block of data to be processed is being read in, and the previous block of data is being written out, the current block of data is being processed.

L	m	Initialization	Obfuscation	Evaluation
Intel Xeon CPU,E5-2698 v3:				
5	3352	66.61	249.80	5.81
6	3932	135.33	503.01	13.03
8	5621	603.06	1865.67	56.61
10	6730	1382.59	4084.14	125.39
12	8339	3207.72	8947.79	300.32
14	9923	7748.91	18469.30	621.48
16	10925	11475.60	38926.50	949.41
17	11928	16953.30	44027.80	1352.48
18	12403	20700.00	out-of-RAM	
4 x 16-core Xeon CPUs:				
17	11928	16523.7	84542.3	646.46
19	13564	36272.9	182001.4	1139.36
20	14145	46996.8	243525.6	1514.26

Table 4: Running time (seconds) as a function of the branching-program length, with security $\lambda = 80$, 100 states, and a binary alphabet (L=BPlength, m=large dimension)

7.2.7 Parallelizing the Evaluation Stage

Recall that in the evaluation stage, we have to multiply encodings along a "main path" and along a "dummy path". In our implementation, each path is processed on a different thread. Specifically, the system sets one thread for processing the dummy branch and one for the main branch (regardless of the total number of threads set during run-time). Then, when processing each branch, the programs sets the number of threads to half of the overall number of threads set during run-time. However, since only each node multiplication is parallelized, and the run-time is relatively negligible for this function, we do not see a difference in the run-time of the evaluation for different number of threads (see Figure 9).

7.3 Results

Most of our testing was done on a machine with Intel Xeon CPU, E5-2698 v3 @2.30GHz (which is a Haswell processor), featuring 32 cores and 250GB of main memory. The compiler was GCC version 4.8.5, and we used NTL version 10.3.0 and GMP version 6.0 (see `http://gmplib.org`).

Because of memory limitations, the largest value of L we could test on that machine was $L = 17$ (though initialization was also possible for $L = 18$). These tests are described in the top part of Table 4. For even larger parameters, we run a few tests on a machine with 4×16-core Xeon CPUs and 2TB of DRAM. All these tests were run on binary alphabets and security parameter $\lambda = 80$. The results of these tests appear in the lower part of Table 4.

8 Conclusions and Future Work

In this work we implemented GGH15-based branching-program obfuscation, showing that on one hand it is feasible to use it to obfuscate non-trivial functions, and on the other hand that the class of functions that it can handle is still extremely limited. In the course of this work we developed many tools and optimizations, that we expect will be useful also elsewhere. In particular We expect that our Gaussian sampling techniques will find use in other lattice-based cryptographic constructions. In this context, it will be interesting to explore using these techniques also in the realm of ring-LWE schemes.

An interesting topic to explore is hardware acceleration (with GPUs or FPGAs), which may be able to reduce overhead to a point where some niche applications can use it.

References

[1] M. R. Albrecht. Private communications, 2016.

[2] M. R. Albrecht, C. Cocis, F. LaguillAumie, and A. Langlois. Implementing candidate graded encoding schemes from ideal lattices. In *Progress in Cryptology - AsiaCRYPT'15*, Lecture Notes in Computer Science. Springer, 2015.

[3] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015.

[4] 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/`.

[5] S. Arora and R. Ge. New algorithms for learning in presence of errors. In *ICALP (1)*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.

[6] Y. Chen, C. Gentry, and S. Halevi. Cryptanalyses of candidate branching program obfuscators. Cryptology ePrint Archive, Report 2016/998, 2016. `http://eprint.iacr.org/2016/998`.

[7] J. Coron, T. Lepoint, and M. Tibouchi. Practical multilinear maps over the integers. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 476–493. Springer, 2013.

[8] J.-S. Coron, M. S. Lee, T. Lepoint, and M. Tibouchi. Cryptanalysis of ggh15 multilinear maps. Cryptology ePrint Archive, Report 2015/1037, 2015. `http://eprint.iacr.org/2015/1037`.

[9] S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In *Advances in Cryptology - EUROCRYPT'13*, volume 7881 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.

[10] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM J. Comput.*, 45(3):882–929, 2016.

[11] C. Gentry, S. Gorbunov, and S. Halevi. Graph-induced multilinear maps from lattices. In Y. Dodis and J. B. Nielsen, editors, *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, volume 9015 of *Lecture Notes in Computer Science*, pages 498–527. Springer, 2015. `https://eprint.iacr.org/2014/645`.

[12] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *"Advances in Cryptology - CRYPTO 2012"*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012. Full version at `http://eprint.iacr.org/2012/099`.

[13] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC'08*, pages 197–206, 2008.

[14] J. Kilian. Founding cryptography on oblivious transfer. In J. Simon, editor, *STOC*, pages 20–31. ACM, 1988.

[15] A. Langlois, D. Stehlé, and R. Steinfeld. Gghlite: More efficient multilinear maps from ideal lattices. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 239–256. Springer, 2014.

[16] 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 *ACM Conference on Computer and Communications Security*, pages 981–992. ACM, 2016.

[17] J. Liu. *Eigenvalue and Singular Value Inequalities of Schur Complements*, pages 47–82. Springer US, Boston, MA, 2005.

[18] D. Micciancio and C. Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 700–718. Springer, 2012. Full version at `http://ia.cr/2011/501`.

[19] C. Peikert. An efficient and parallel gaussian sampler for lattices. In T. Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2010.

[20] V. Shoup. NTL: A Library for doing Number Theory. `http://shoup.net/ntl/`, Version 9.11.0, 2016.

[21] F. Zhang. *The Schur Complement and Its Applications*. Numerical Methods and Algorithms. Springer, 2005.

A More Performance Details

A.1 Asymptotics of Obfuscation

For a given branching-program length L and security parameter λ, our choice of parameters from before ensures that $\ell = \log(q)$ and the lattice dimension m satisfy $\ell \geq \Omega(L \log m + \lambda)$ and $m \geq \Omega(\ell\lambda)$.

It is easy to see that these constraints imply $\ell \geq \Omega(\lambda + L\log(\lambda L))$ and $m \geq \Omega(\lambda^2 + \lambda L\log(\lambda L))$. This means that each encoding matrix $C \in \mathbb{Z}_q^{m\times m}$ takes space $\ell m^2 = \Omega(\lambda^5 + \lambda^2 L^3 \log^3(\lambda L))$ to write down, and multiplying or inverting such matrices takes time $\ell m^3 = \Omega(\lambda^7 + \lambda^3 L^4 \log^4(\lambda L))$.

For a length-L branching program over an alphabet of size σ, the obfuscated program consists of $2\sigma(L-1)$ matrices, so:

- the total space that it consumes is $\Omega(\sigma\lambda^5 L + \sigma\lambda^2 L^4 \log^3(\lambda L)))$, and
- the time to compute it is $\Omega(\sigma\lambda^7 L + \sigma\lambda^3 L^5 \log^4(\lambda L))$.

In words, the obfuscation running time is linear in σ, sextic in λ, and quasi-quintic in L, and the hard-disk size needed is linear in σ, quintic in the security parameter, and quasi-quartic in L.

We note, however, that our implementation is parallelized across the different CRT components, whose number is proportional to $\ell = \log q$, so we expect one factor of ℓ from the running-time to be eaten up by this parallelism. We thus expect the wall-clock time of the obfuscation to be "only" quasi-quartic in the program length $\tilde{\Omega}(L^4)$, and sextic in the security parameter $\Omega(\lambda^6)$.

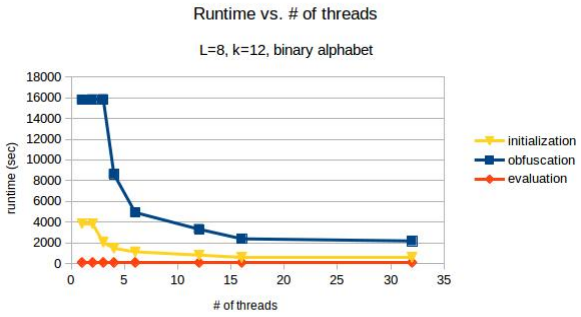
For the RAM requirements, our implementation keeps only two matrices in RAM at the same time so it uses $\Omega(\lambda^5 + \lambda^2 L^3 \log^3(\lambda L))$ memory, but this could be reduced further (by only keeping a small number of CRT components in memory, or only keeping a small number of slices of each matrix in memory).

A.2 Concrete Results

To save time, we did almost all of our experiments with binary alphabet $|\Sigma| = 2$, but for our parameter $L = 15$ we also ran it where the input is expressed in nibbles $|\Sigma| = 16$, to verify that it works also for that setting. As expected, initialization and evaluation were not affected by the alphabet size, and RAM usage during obfuscation was only marginally higher, while running-time and disk usage in obfuscation were exactly 8 times larger for $|\Sigma| = 16$ than for $|\Sigma| = 2$. The timing results for various settings can be found in Table 4, and memory and disk-space usage are described in Figures 7, 8, and 9.

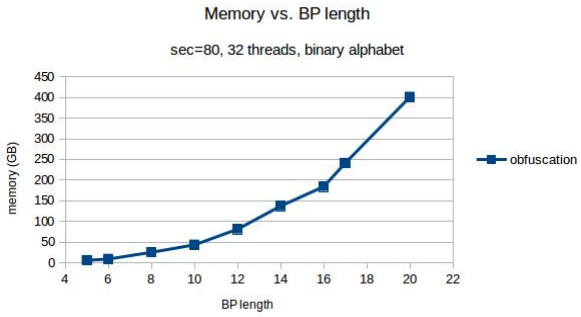
We also ran tests to examine the effectiveness of our parallelization strategies, comparing the running times for the same parameters ($L = 8$, binary alphabet, and 12 CRT factors) across different number of threads. As expected given our choice of parallelism across CRT component, increasing the number of threads upto the number of CRT factors reduces the running time, but adding more threads after that has no effect. The detailed results are described in Figure 6. [9]

[9] Our code always allocates 1-2 threads for pipelined I/O, and only increases the number of worker threads after that, hence the decrease in running-time only begins at 2-3 threads. Also the running time plateaus at 15 threads, even though there are only 12 CRT factors.



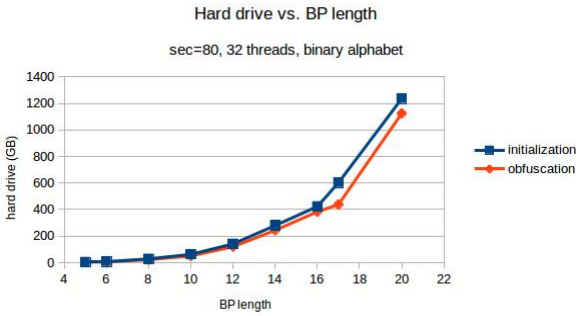
# of threads	Initialization	Obfuscation	Evaluation
1	3840.2	15809.2	90.4
2	3833.6	15837.1	91.2
3	2032.0	15844.3	89.5
4	1451.8	8605.9	87.2
6	1120.0	4917.8	89.7
12	803.0	3298.5	88.7
16	560.1	2375.8	91.8
32	568.9	2168.1	98.5

Figure 6: Running time (seconds) as a function of the number of threads.



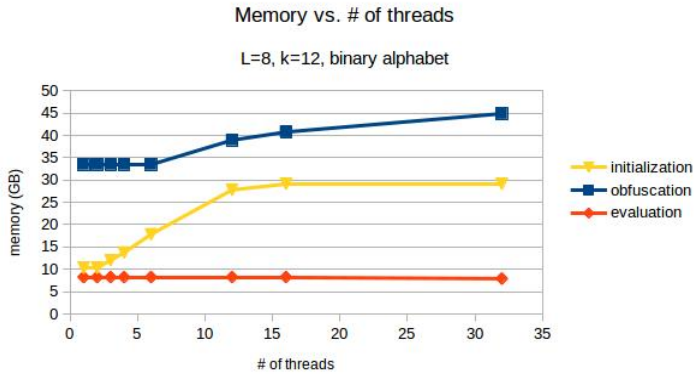
L	m	RAM (Obfuscation)
5	3352	5.5
6	3932	8.7
8	5621	25
10	6730	43
12	8339	81
14	9923	137
16	10925	184
17	11928	241
20	14145	401

Figure 7: RAM usage (Gigabytes) as a function of the BP-length. (L = input size, m = large dimension)



L	m	Initialization (GB)	Obfuscation (GB)
5	3352	3.7	2.3
6	3932	7.3	5.0
8	5621	28	13
10	6730	61	50
12	8339	141	120
14	9923	280	244
16	10925	432	383
17	11928	602	538
20	14145	1236	1124

Figure 8: Hard disk usage as a function of the input length. (L = input size, m = large dimension)



# of threads	Initialization (GB)	Obfuscation (GB)	Evaluation (GB)
1	10	33	8.2
2	10	33	8.2
3	12	33	8.2
4	14	33	8.2
6	18	33	8.2
12	28	39	8.3
16	29	41	8.3
32	29	45	7.9

Figure 9: Memory usage for different number of threads, length $L = 8$.