5Gen: A Framework for Prototyping Applications Using Multilinear Maps and Matrix Branching Programs

Kevin Lewi* Alex J. Malozemoff† Daniel Apon‡
Brent Carmer§ Adam Foltzer† Daniel Wagner† David W. Archer†
Dan Boneh* Jonathan Katz‡ Mariana Raykova¶

Abstract

Secure multilinear maps (mmaps) have been shown to have remarkable applications in cryptography, such as multi-input functional encryption (MIFE) and program obfuscation. To date, there has been little evaluation of the performance of these applications. In this paper we initiate a systematic study of mmap-based constructions. We build a general framework, called `5Gen`, to experiment with these applications. At the top layer we develop a compiler that takes in a high-level program and produces an optimized matrix branching program needed for the applications we consider. Next, we optimize and experiment with several MIFE and obfuscation constructions and evaluate their performance. The `5Gen` framework is modular and can easily accommodate new mmap constructions as well as new MIFE and obfuscation constructions, as well as being an open-source tool that can be used by other research groups to experiment with a variety of mmap-based constructions.

1 Introduction

A multilinear map (mmap) [BS02] is an extremely powerful tool for constructing advanced cryptographic systems including program obfuscation [GGH+13b], n-party non-interactive key exchange [BS02], multi-input functional encryption [GGG+14, BLR+15], optimal broadcast encryption [BWZ14a], witness encryption [GGSW13], and many others. The recent emergence of candidate mmaps [GGH13a, CLT13, CLT15, GGH15] bring these proposals closer to reality, although several of the current candidates have been shown to be too weak for some of these applications, as discussed in §5.

Despite the remarkable power of mmaps, few published works study the efficiency of the resulting applications, primarily due to the rapid pace of development in the field and the high resource requirements needed for carrying out experiments. In this paper we develop a generic framework

*Stanford University, {klewi,dabo}@cs.stanford.edu
†Galois, {amaloz,acfoltzer,dmwit,dwa}@galois.com. Portion of work of Alex J. Malozemoff done while at University of Maryland.
‡University of Maryland, {dapon,jkatz}@cs.umd.edu
§Oregon State University, carmerb@eecs.oregonstate.edu. Portion of work done while at Yale University.
¶Yale University, mariana.raykova@yale.edu

called `5Gen`[1] (available at `https://github.com/5GenCrypto`) that lets us experiment with powerful applications of current and future mmaps. We focus on two applications in particular: multi-input functional encryption (MIFE) and program obfuscation, both of which can be instantiated with some of the existing mmap candidates (see §5). Our framework is built as a multi-layer software stack where different layers can be implemented with any of the current candidates or replaced altogether as new constructions emerge.

The top layer of our framework is a system to compile a high-level program written in the Cryptol language [Cry] into a matrix branching program (MBP), as needed for the most efficient MIFE and obfuscation constructions. We introduce several novel optimizations for obtaining efficient MBPs and show that our optimizations reduce both the dimension and the total number of matrices needed.

The next layer implements several variants of MIFE and obfuscation using a provided MBP. This lets us experiment with several constructions and to compare their performance.

The lowest layer is the multilinear map library, `libmmap`. We demonstrate our framework by experimenting with two leading candidate mmaps: GGHLite [GGH13a, LSS14, ACLL15] and CLT [CLT13, CLT15]. Our experiments show that for the same level of security, the CLT mmap performs considerably better than GGHLite in all the applications we tried, as explained in §8. (Although the GGH15 multilinear map [GGH15] was not included in our implementation or experiments, we hope that future work might integrate this multilinear map into our framework.)

Our framework makes it possible to quickly plug in new mmaps as new proposals emerge, and easily measure their performance in applications like MIFE and obfuscation.

MIFE experiments. Recall that functional encryption [BSW11] is an encryption scheme where the decryption key sk_f is associated with a function f. If c is the encryption of message m then decrypting c with the key sk_f gives the decryptor the value $f(m)$ and nothing else. An n-input MIFE scheme is the same, except that the function f now takes n inputs. Given independently created ciphertexts $c_1, \ldots, c_n$, with each c_i an encryption of a message m_i and associated with "slot" i, decrypting these ciphertexts using sk_f reveals $f(m_1, \ldots, m_n)$ and nothing else.

One important application of 2-input MIFE is *order-revealing encryption* (ORE) [GGG$^+$14, BLR$^+$15]. Here the function $f(x, y)$ outputs 1 if $x < y$ and 0 otherwise. Thus, the key sk_f applied to ciphertexts c_1 and c_2 reveals the relative order of the corresponding plaintexts. ORE is useful for responding to range queries on an encrypted database. For large domains, the only known constructions for secure ORE are based on mmaps. We conduct experiments on ORE using real-world security parameters where mmaps give the best known secure construction.

We also experiment with 3-input MIFE. Here, we choose a DNF formula f that operates on triples of inputs, which is useful in the context of privacy-preserving fraud detection where a partially trusted gateway needs to flag suspicious transactions without learning anything else about the transactions (see §6.3). Again, the best known construction for such a scheme uses mmaps.

We use our framework to evaluate the implementation of these schemes using existing mmaps for which they are currently believed to be secure. Clearly these systems are too inefficient to be used in practice. Nevertheless, our experiments provide a data point for the current cost of using them. Moreover, our framework makes it possible to easily plug in better or more secure mmaps as they become available.

Obfuscation experiments. Roughly speaking, an obfuscator takes as input a program and

[1] The name `5Gen` comes from the fact that multilinear maps can be considered the "fifth generation" of cryptography, where the prior four are: symmetric key, public key, bilinear maps, and fully homomorphic encryption.

outputs a functionally equivalent program such that the only way to learn information about the program is to run it. We experiment with several obfuscators built on the obfuscator described by Barak et al. [BGK+14], including those inspired by Sahai and Zhandry [SZ14] and Ananth et al. [AGIS14]. These improvements allow for obfuscation of a point function with increased security at less than half the total obfuscation size reported by Apon et al. [AHKM14]. We also implemented the Zimmerman [Zim15] obfuscator, but we ultimately found that it was too inefficient for the settings that we consider in our experiments.

1.1 Our Contributions

Summarizing, we make the following contributions:

- An optimizing compiler from programs written in the Cryptol language to MBPs, which are used in many mmap applications including MIFE and obfuscation. Our compiler uses optimizations such as dimension reduction, matrix pre-multiplication, and condensing the input representation, and solves a constraint-satisfaction problem needed to obtain the most efficient MBP. See §4 for details.

- A library providing a clean API to various underlying mmap implementations. This allows researchers to experiment with different mmaps, as well as to easily plug future mmaps into our framework. See §5 for more details.

- A general MIFE construction based on the scheme of Boneh et al. [BLR+15] using real-world security parameters. We contribute optimized implementations of two-input MIFE (in particular, order-revealing encryption) and three-input MIFE (in particular, a functionality needed for privacy-preserving fraud detection), as well as performance results that characterize our constructions. See §6 for details and §8 for evaluation results.

- Obfuscation constructions [BGK+14, SZ14, AGIS14, Zim15] using real-world security parameters. We experiment with obfuscating point functions and evaluate their performance. See §7 for details and §8 for evaluation results.

1.2 Related Work

Several groups have previously implemented mmaps [ACLL15, CLT13, CLT15] to experiment with their performance. However, they did not go so far as experimenting with cryptographic applications of mmaps beyond direct applications such as multi-party non-interactive key exchange. The goal of our work is to explore the performance of more advanced applications such as MIFE and obfuscation. An earlier work implementing obfuscation [AHKM14] experimented with obfuscating point functions, and was only able to successfully obfuscate a 14-bit point function.

Our work builds on the vast amount of previous work showing applications of mmaps—most notably, MIFE [GGG+14, BLR+15] and obfuscation [BR14, BGK+14, PST13, AGIS14, Zim15, AB15, GLSW15, BMSZ15, Lin16, LPST16, GMS16, MSZ16b, DGG+16].

2 Preliminaries

In this section, we introduce notation and also define the various cryptographic constructions that we use in the rest of this work.

Notation. For an integer $n > 0$, we use $[n]$ to denote the set of integers $\{1, \dots, n\}$. We use λ to represent the security parameter, where "λ-bit security" means that security should hold up to 2^λ *clock cycles*. We assume that all of our procedures run *efficiently*, or more formally, in polynomial-time with respect to the size of the input to the procedure, and polynomial in the security parameter λ.

Multilinear maps. Boneh and Silverberg [BS02] first proposed the concept of *multilinear maps* (mmaps), but it was only in 2013 that Garg, Gentry, and Halevi [GGH13a] introduced the first plausible construction of an mmap. Since then, mmaps have been shown to be powerful tools in solving numerous problems in cryptography.

A *multilinear map* [BS02, GGH13a] (or *graded encoding scheme*) is a primitive for producing randomized encodings of plaintexts that may be publicly added, multiplied, and zero-tested but otherwise "do not reveal any information." Encodings are associated with a "level" that restricts the types of operations that may be performed on that encoding. More formally, a degree-κ multilinear map is a tuple of algorithms $(\mathsf{Setup}, \mathsf{Encode}, \mathsf{Add}, \mathsf{Mult}, \mathsf{ZeroTest})$ where:

- Setup takes as input the security parameter, and outputs a private parameter sp and a public parameter pp that, in particular, specifies a ring R.
- Encode takes sp, an element $x \in R$, and a level $S \subseteq [\kappa]$, and outputs a *level-S encoding of x* denoted $[\![x]\!]_S$.
- Add takes pp and two encodings $[\![x]\!]_S$, $[\![y]\!]_S$ at the *same level* S, and outputs an encoding $[\![x + y]\!]_S$.
- Mult takes pp and two encodings $[\![x]\!]_{S_1}$, $[\![y]\!]_{S_2}$ for *disjoint* levels S_1, S_2, and outputs an encoding $[\![x \cdot y]\!]_{S_1 \cup S_2}$.
- $\mathsf{ZeroTest}$ takes pp and an encoding $[\![x]\!]_U$ for $U = [\kappa]$. It outputs 1 if and only if $x = 0$.

Informally, an mmap is secure if the only information that an attacker can figure out from the encodings of random elements is exactly the information that can be obtained from running Add, Mult, and $\mathsf{ZeroTest}$, and no more. (We omit any formal definitions since we do not directly rely on them in this work but instead inherit them from prior work.)

Matrix branching programs. A matrix branching program (MBP) of length n on length-ℓ, base-d inputs is a collection of variable-dimension matrices $\mathbf{B}_{i,j}$ for $i \in [n]$ and $j \in \{0, \dots, d-1\}$, a "final matrix" $\mathbf{P}$, and an "input mapper" function $\mathsf{inp} : [\ell] \to [n]$. We require that, for each $i \in [2, n]$ and $j \in \{0, \dots, d-1\}$, the number of columns of $\mathbf{B}_{i-1,j}$ is equal to the number of rows of $\mathbf{B}_{i,j}$, so that the product of these matrices is well-defined. The evaluation of an MBP on input $x \in \{0, \dots, d-1\}^\ell$ is defined as

$$\mathsf{MBP}(x) = \begin{cases} 1, & \text{if } \prod_{i=1}^{n} \mathbf{B}_{i,x_{\mathsf{inp}(i)}} = \mathbf{P}, \\ 0, & \text{otherwise.} \end{cases}$$

We note that numerous generalizations and extra properties [BLR$^+$15, SZ14] of MBPs have been explored in the literature—however, we will only need to use our simplified definition of MBPs for the remainder of this work.

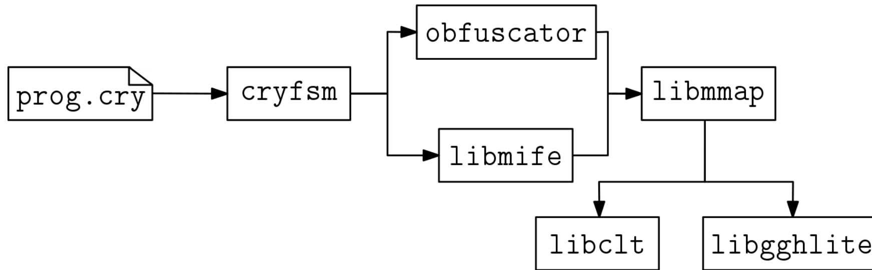

Figure 3.1: Framework architecture. We use `cryfsm` to compile a Cryptol program (here denoted by `prog.cry`) to an MBP, which can either be used as input into our MIFE implementation or our obfuscation implementation. Both these implementations use `libmmap` as a building block, which supports both the CLT (`libclt`) and GGHLite (`libgghlite`) mmaps.

3 Framework Architecture

Our framework incorporates several software components that together enable the construction of applications using mmaps and MBPs. In particular, we use our framework to develop implementations of MIFE and program obfuscation. See Figure 3.1 for the framework architecture.

The top layer of our framework, `cryfsm`, takes as input a program written in Cryptol [Cry], a high-level language designed to express manipulations over bitstreams in a concise syntax, and compiles the program into an MBP. This process, and the various optimizations we introduce, are described in more detail in §4.

The bottom layer of our framework, `libmmap`, provides an API for using various mmaps, which in our case includes the CLT (through the `libclt` library) and GGHLite (through the `libgghlite` library) mmaps. The `libmmap` library, which we describe in §5, is also designed to allow for a straightforward integration of future mmap implementations.

We combine the above components to realize various applications of mmaps and MBPs: in particular, MIFE and program obfuscation. We demonstrate the applicability of our MIFE implementation (cf. §6) through two examples: order-revealing encryption (ORE) and three-input DNF (3DNF) encryption. We implement program obfuscation based on two main approaches: the techniques described by Sahai and Zhandry [SZ14], and also the scheme by Zimmerman [Zim15], which operates over arithmetic circuits, but only applies to the CLT mmap.

4 From Programs to MBPs

One of our key contributions in this work is a compiler, `cryfsm`, that takes as input a program written in Cryptol [Cry], a domain-specific language for specifying algorithms over generic streams of bits, and produces an MBP for the given input program. `cryfsm` does this by translating a Cryptol specification into a *layered state machine*, which can then be transformed into an optimized corresponding MBP.

Our toolchain proceeds as follows. The user writes a Cryptol function of type `[n] -> Bit` for some `n` (that is, the function takes `n` input bits and produces one output bit). This function is interpreted as deciding membership in a language. The toolchain symbolically evaluates this function to produce a new version of the function suitable for input to an SMT solver, as explained in detail below. Queries to the SMT solver take the form of deciding the prefix equivalence relation

between two initial bitstrings, which is sufficient to build the minimal layered state machine, which we then convert to an MBP.

Our solver-based approach results in a substantial dimension reduction of the corresponding output MBPs that we tested. In contrast, the traditional approach would be to heuristically optimize the state machine design in an attempt to achieve a best-effort optimization. The dimension reduction we achieve recovers the most efficient known MBPs for several previously studied bitstring functions, including MBPs for point functions that are smaller than the MBPs constructed from boolean formulas using existing techniques (e.g., [SZ14]). In the remainder of this section, we describe the key steps in this toolchain, along with several optimizations to the MBPs that we use throughout the remainder of this work.

Specifying functions in Cryptol. Cryptol is an existing language widely used in the intelligence community for describing cryptographic algorithms. A well-formed Cryptol program looks like an algorithm specification, and is executable. The Cryptol tool suite supports such execution, along with capabilities to state, verify, and formally prove properties of Cryptol specifications, and capabilities to both prove equivalence of implementation in other languages to Cryptol specifications and automatically generate such implementations. In our work, a user specifies an MBP in Cryptol, and then we use `cryfsm` to transform the high-level specification into a minimal layered state machine, and further transform it into an efficient MBP.

Minimal layered state machines. There is a standard translation from traditional finite state machines to MBPs: create a sequence of matrix pairs (or matrix triples for three-symbol alphabets, etc.) that describe the adjacency relation between states. If state i transitions to state j on input symbol number b, then the bth MBP matrix will have a 1 in the ith row and jth column and 0 elsewhere. For many languages of interest, this is inefficient: for an automaton with $|S|$ states, each matrix must be of size $|S|^2$, even though many states may be unreachable.

In the applications of mmaps that we study in this work, we consider functions on inputs of a fixed length. Hence, for a positive integer n, we can take advantage of this property by restricting ourselves to *layered state machines* of depth n, which are simply (deterministic) finite state machines that only accept length-n inputs. Here, the ith "layer" of transitions in the machine is only used when reading the ith digit of the input. As a result, layered state machines are acyclic.

To generate minimal layered state machines, our compiler must introduce machinery to track which states are reachable at each layer, which allows us to reduce the overall MBP matrix dimensions. To do this, `cryfsm` computes the quotient automaton of the layered state machine using an SMT solver to decide the state equivalence relation. The quotient automaton is then used as the new minimal layered state machine for the specified function. Then, from a layered state machine of depth n, we construct the corresponding MBP on base-d inputs of length n in a manner essentially equivalent to the techniques of Ananth et al. [AGIS14] for constructing layered branching programs. Intuitively, for each $i \in [n]$ and $j \in [d]$, the ith matrix associated with the jth digit is simply the adjacency matrix corresponding to the transitions belonging to the ith layer of the machine, associated with reading the digit j. Then, the "final matrix" (that defines the output of the MBP being 1) is simply the adjacency matrix linking the initial state to the final state of the layered state machine.

Optimizations for MBP creation. Boneh et al. [BLR$^+$15] describe a simple five-state finite state machine appropriate for ORE applications, and describe the translation to MBPs that produces 5×5 matrices at each depth. The MBP we build and use for our ORE application differs from this one via three transformations that can be generalized to other programs: change of base, matrix

premultiplication, and dimension reduction. Of these, matrix premultiplication and dimension reduction are a direct consequence of the technique used by `cryfsm` for constructing MBPs and therefore automatically apply to all programs, whereas choosing an input base remains a manual process because it must be guided by outside knowledge about the performance characteristics of the mmap used to encode the MBPs. While the change of base and matrix pre-multiplication optimizations are described by Boneh et al., we introduce dimension reduction as a new optimization that is useful for ORE yet generalizable to other applications.

For each optimization, we use the integer d to represent the "input base", the integer n to represent the length (number of digits) of each input, the integer N to represent the input domain size (so, we have that $d^n \geq N$), the integer m to represent the length of the MBP, and the integer M to represent the total number of elements across all the matrices of the MBP.

At a high level, the optimizations are as follows.

- **Condensing the input representation** corresponds to processing multiple bits of the input, by increasing d, to reduce the length of the MBP, at the expense of increasing the number M of total elements.
- **Matrix premultiplication** also aims to reduce the parameter m, but without increasing the parameter M.
- **Dimension reduction** aims to directly reduce the number M of total elements, but may not be fully compatible with matrix premultiplication, depending on the function.

To help with the understanding of the intuition behind these optimizations, we use the simple comparison state machine as a running example—however, we stress that these optimizations are in no way specific to the comparison function, and can be applied more generally to any function expressed as a layered state machine.

Condensing the input representation. The most immediate optimization that we apply is to condense the representation of inputs fed to our state machines. MBPs are traditionally defined as operating on bitstrings, so it is natural to begin with state machines that use bits as their alphabet, but using larger alphabets can cut down on the number of state transitions needed (at the potential cost of increasing the state space).

As an example, for evaluating the comparison state machine, this optimization translates to representing the input strings in a larger base $d > 2$, and to adjust the comparison state machine to evaluate using base-d representations. The resulting state machine consists of $d+3$ total states.

A naive representation of an input domain of size N with a state machine that processes the inputs bit-by-bit (in other words, $d = 2$) would induce an MBP length of $m = 2 \cdot \lceil \log_2(N) \rceil$ and $M = 50 \cdot m$ total elements (in two 5×5 matrices). However, by using the corresponding comparison state machine that recognizes the language when the inputs are in base-d, we can then set $m = 2 \cdot \lceil \log_d(N)/\log_2(d) \rceil$ and $M = 2 \cdot (d+3)^2 \cdot m$.

Concretely, setting $N = 10^{12}$, without condensing the input representation, we require $m = 80$ and $M = 2000$ for the resulting MBP. However, if we represent the input in base-4, we can then obtain $m = 20$ and $M = 1960$, a strict improvement in parameters.

Matrix premultiplication. Boneh et al. [BLR+15] informally describe a simple optimization to the comparison state machine, which we explain in more detail here. The natural state machine for evaluating the comparison function on two n-bit inputs x and y reads the bits of x and y in the order $x_1 y_1 x_2 y_2 \cdots x_n y_n$.

However, Boneh et al. show that a slight reordering of the processing of these input bits can result in reduced MBP length without compromising in correctness. When the inputs are instead read in the following order:

$$x_1 y_1 y_2 x_2 x_3 y_3 \cdots y_n x_n, \tag{1}$$

then, rather than producing one matrix for each input bit position during encryption, the two matrices corresponding to y_1 and y_2 can be pre-multiplied, and the result is a *single* matrix representing two digit positions. Naturally, this premultiplication can be performed for each pair of adjacent bit positions belonging to the same input string (such as for $x_2 x_3$, $y_3 y_4$, and so on), and hence the number of matrices produced is slightly over half of the number of matrices in the naive ordering of input bits.

As a result, for evaluating the comparison state machine, where n is the length of the base-d representation of an input, applying this optimization implies $m = n + 1$, a reduction from the naive input ordering, which would result in $m = 2n$, and a reduction from $M = 2 \cdot (d+3)^2 \cdot m$ to $M = (d+3)^2 \cdot m$. When applying this optimization in conjunction with representing the input in base $d = 4$, for example, setting $N = 10^{12}$ only requires $m = 21$ and $M = 1029$, a huge reduction in cost that was emphasized by Boneh et al., and another strict improvement in parameters.

A new optimization: dimension reduction. We now describe a more sophisticated optimization that can be applied to general MBPs which also results in a reduced ciphertext size. As an example, we describe this optimization, called dimension reduction, as it applies to the comparison function state machine (without applying the reordering of input bits from matrix premultiplication), but we emphasize that the technique does not inherently use the structure of this state machine in any crucial way, and can naturally be extended to general MBPs.

Our new optimization stems from the observation that, for each bit position in the automaton evaluation, the transitions in the automaton do not involve all of the states in the automaton. This is the same observation that motivates the use of layered state machines over finite state machines.

In particular, for the even-numbered bit positions, the transitions map from a set of d states to a set of only 3 states. Similarly, for the odd-numbered bit positions, the transitions map from a set of (at most) 3 states to a set of d states. As a result, the corresponding matrices for each bit position need only be of dimension $d \times 3$ or $3 \times d$ (depending on the parity), as opposed to the naive interpretation of the Boneh et al. construction which requires matrices of dimension $(d+3) \times (d+3)$.

Note, however, that the dimension reduction optimization is not fully compatible with matrix premultiplication, since the effectiveness of dimension reduction can degrade if matrix premultiplication is also applied. In particular, when applying matrix premultiplication to the comparison state machine, we notice that there is less room for improvements with dimension reduction, as the transitions for the position $y_1 y_2$ correlate from a domain of d states to a range of also d states.

In §6.1, we concretely show how to apply a mixture of these optimizations to the comparison automaton, and then use these optimizations to obtain asymptotically shorter ciphertexts for order-revealing encryption.

vtable	function	comments
mmap_pp_vtable	fread/fwrite	read/write public parameters
	clear	clear public params
mmap_sk_vtable	init/clear	initialize/clear secret key
	fread/fwrite	read/write secret key
mmap_enc_vtable	init/clear	initialize/clear encoding
	fread/fwrite	read/write encoding
	set	copy encoding
	add	implements Add
	mul	implements Mult
	is_zero	implements ZeroTest
	encode	implements Encode

Table 5.1: Interfaces exported by the libmmap library.

5 A Library for Multilinear Maps

In this section we describe our library, libmmap, which provides an API for interacting with different mmap backends. In this work we implement GGHLite (libgghlite) and CLT (libclt) backends[2], although we believe that it should be relatively straightforward to support future mmap implementations.

The libmmap library exports as its main interface a virtual method table mmap_vtable, which in turn contains virtual method tables for the public parameters (mmap_pp_vtable), the secret key (mmap_sk_vtable), and the encoded values (mmap_enc_vtable). Table 5.1 lists the available functions within each table. Each underlying mmap library must export functions matching these function interfaces and write a wrapper within libmmap to match the virtual method table interface. A user of libmmap then defines a pointer const mmap_vtable * which points to the virtual method table corresponding to the mmap of the user's choice (in our case, either clt_vtable or gghlite_vtable). In the following, we describe the two mmap schemes we support within libmmap: libgghlite (§5.1) and libclt (§5.2).

Figure 5.1 presents estimates for the size of an encoding using GGHLite and CLT for security parameters $\lambda = 80$ and $\lambda = 40$. We describe our parameter choices for arriving at these estimates in Appendix A. As we can see, the CLT mmap produces smaller encodings than GGHLite as we vary both λ and κ. This appears to be due to the growth of the lattice dimension in GGHLite compared to the number of secret primes required by the CLT scheme, among other factors.

5.1 The GGHLite Multilinear Map

Building off of the original mmap candidate construction of Garg et al. (GGH) [GGH13a], Langlois et al. [LSS14] proposed a modification called GGHLite, along with parameter and performance estimates for the resulting encodings of the scheme. More recently, Albrecht et al. [ACLL15] proposed further modifications and optimizations on top of GGHLite, along with an implementation

[2]We also have a "dummy" mmap implementation for testing purposes.

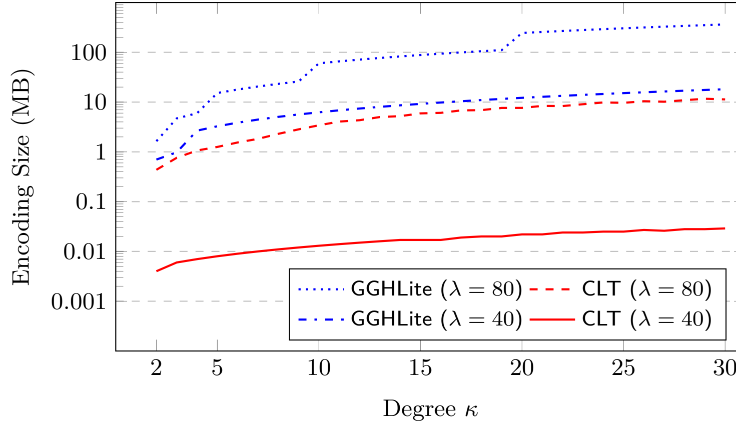

Figure 5.1: Estimates for the size of a single encoding in megabytes (MB) produced for security parameters $\lambda = 80$ and $\lambda = 40$ and varying the multilinearity degree $\kappa \in [2, 30]$ for the GGHLite and CLT mmaps.

of their scheme under an open-source license. In this work, we refer to GGHLite as the construction from the work of Albrecht et al., as opposed to the original work of Langlois et al.

Our GGHLite implementation. We use as our starting point the implementation of GGHLite[3] released by Albrecht et al. [ACLL15]. We modified this implementation to add functionality for handling the reading and writing of encodings, secret parameters, and public parameters to disk. We also extended the implementation to handle more expressive index sets, which are used in MIFE and obfuscation, as follows.

Typically, multilinear maps only support "levels", where each encoding is created with respect to an integer $i \in [\kappa]$ (for an mmap of degree κ). The GGHLite implementation supports more advanced labelings of encodings, by allowing for a universe U of κ indices to be defined, and each encoding can be created with respect to a singleton subset (containing only one element) of this universe U. Multiplication of two encodings with respect to sets of indices S_1 and S_2 produces an encoding with respect to the multiset union of S_1 and S_2. The zero-testing parameter is then created to test for encodings which are labeled with respect to U. However, this functionality is still not sufficiently expressive to match the needs of our implementation and our definition of mmaps.

Consequently, we upgraded the handling of these encodings to support labelings of an encoding with respect to *any subset* S of indices of the universe U. Then, when two encodings labeled with two different subsets are multiplied, the resulting encoding is labeled with respect to their multi-set union. Finally, as before, the zero-testing parameter allows to check for encodings of 0 labeled at U, only.

Finally, we isolated and rewrote the randomness generation procedures used by GGHLite, since the original implementation relied on the randomness obtained from the GMP library, which is not generated securely. We split this into a separate library, `libaesrand`, which uses AES-NI for efficient randomness generation, and which may be useful in other contexts.

Attacks on GGHLite. Recently, Hu and Jia [HJ16] showed how to perform "zeroizing" attacks on GGHLite, to recover the secret parameters given certain public encodings of 0. However, since neither MIFE nor obfuscation publish any encodings of 0, these applications seem to be unaffected by the zeroizing attacks. More recently, Albrecht, Bai, and Ducas [ABD16] gave a quantum break

[3] `https://bitbucket.com/malb/gghlite-flint`

for GGHLite *without* using any encodings of 0 or the public zero-testing parameter. Subsequently, Cheon, Jeong, and Lee [CJL16] showed how to give a (classical) polynomial-time attack on GGHLite, again without using any encodings of 0. However, their attack requires exponential time if the parameters of GGHLite are sufficiently increased (by a polynomial amount).

In concurrent work, Miles, Sahai, and Zhandry [MSZ16a] gave a completely different form of attack, known as an "annihilation" attack, on *applications* of GGHLite, specifically, MIFE and program obfuscation. They show that provably secure instantiations of these primitives from mmaps are in fact insecure when the mmap is instantiated with GGHLite. Despite the annihilation attacks, our implementations of these primitives from GGHLite still serve as a useful benchmark for the efficiency of GGHLite and for the efficiency of future GGH-like schemes resistant to annihilation attacks, which will inevitably arise from improvements to the GGH framework.

5.2 The CLT Multilinear Map

Coron, Lepoint, and Tibouchi [CLT13] proposed a candidate multilinear map over the integers, which works over a composite modulus that is assumed to be hard to factor.

Our CLT implementation. Our implementation started with the implementation[4] of CLT in C++ by Coron et al. [CLT13]. We rewrote it in C and added functionality to save and restore encodings and the public parameters. As in the GGHLite case, we also modified its basic functionality to support indices instead of levels.

Furthermore, in our extension of CLT, we improve the efficiency of the encoding process which allows for us to apply the CLT multilinear map to the large parameter settings that we consider in the remainder of this work. The original CLT implementation applies the Chinese Remainder Theorem in the procedure that produces encodings of plaintext elements. Our implementation employs a certain trade-off that allows for the application of the Chinese Remainder Theorem in a recursive manner, resulting in more multiplications to compute the encoding, but with the efficiency gain that the elements being multiplied are much smaller. Experimentally, this yields a large speedup in the encoding time, more noticeably with larger parameters. In particular, for $\lambda = 80$ and $\kappa = 19$, without this optimization, it takes 134 seconds to produce a CLT encoding, whereas with our optimization, this time drops to 33 seconds.

Attacks on CLT. Similarly to other candidate constructions for multilinear maps, the CLT construction was not based on an existing hardness assumption but rather introduced a new assumption. Subsequently Cheon et al. [CHL$^+$15] demonstrated a zeroizing attack against the construction of CLT, which succeeds in recovering the secret parameters of the scheme. This attack was further extended in the work of Coron et al. [CGH$^+$15], which demonstrated how it can be generalized and applied against some proposed countermeasures [BWZ14b, GGHZ14] to the attack by Cheon et al. [CHL$^+$15]. But again, as with the zeroizing attacks on GGHLite, these results do *not* apply directly to the constructions we consider in this work.

6 Multi-Input Functional Encryption

The notion of multi-input functional encryption (MIFE), introduced by Goldwasser et al. [GGG$^+$14], extends the concept of functional encryption [BSW11] so that a decryption key is associated with a *multi-input* function which is evaluated over multiple ciphertexts. More formally, a secret-key,

[4] `https://github.com/tlepoint/multimap`

fixed-key MIFE scheme for a function f, on m inputs and with output in a range $\mathcal{R}$, is a tuple of algorithms ($\mathsf{keygen}, \mathsf{encrypt}, \mathsf{eval}$) such that:

- $\mathsf{keygen}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The algorithm takes as input the security parameter and generates the public parameters pp and a secret key sk.
- $\mathsf{encrypt}(\mathsf{sk}, i, x) \to \mathsf{ct}$. The algorithm takes as input the secret key sk, an input position index i, and an input x, and outputs a ciphertext ct.
- $\mathsf{eval}(\mathsf{pp}, \mathsf{ct}_1, \ldots, \mathsf{ct}_m) \to z$. The algorithm takes as input a secret key sk and m ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$, and produces an output $z \in \mathcal{R}$.

Correctness requires that for any inputs $x_1, \ldots, x_m$, for $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{keygen}(1^\lambda)$, letting $\mathsf{ct}_i = \mathsf{encrypt}(\mathsf{sk}, i, x_i)$ for each $i \in [m]$, we have that

$$\mathsf{eval}(\mathsf{pp}, \mathsf{ct}_1, \ldots, \mathsf{ct}_m) = f(x_1, \ldots, x_m).$$

Informally, an MIFE scheme is *secure* if the information revealed by a collection of ciphertexts is exactly the information that can be obtained by running eval, and no more. We omit formal security definitions since we do not directly rely on them in this work.

Goldwasser et al. [GGG+14] gave a general MIFE construction that uses indistinguishability obfuscation in a black-box manner. Boneh et al. [BLR+15] proposed a secret-key MIFE construction that is based directly on mmaps (instead of obfuscation) in order to obtain better efficiency. A particular instantiation of this MIFE construction, where the function used in the decryption key is the comparison function, results in a construction for *order revealing encryption* (ORE), which allows comparisons over ciphertexts while hiding all other information about the encrypted messages. More specifically, an ORE scheme is a MIFE scheme with the function $f(x, y)$ that outputs the ordering between x and y. This ORE scheme achieves the optimal security definition for a scheme that allows the comparison functionality over encrypted data, improving over the security level provided by order-*preserving* encryption schemes.

MIFE implementation. We implemented the Boneh et al. [BLR+15] MIFE construction on top of the `libmmap` library, and provide interfaces for keygen, encrypt, and eval, which perform the respective operations supported by MIFE. We parallelize the computation performed during keygen, but for encrypt, we choose to sequentially construct the encodings belonging to the ciphertext, and instead defer the parallelism to the underlying mmap implementation for producing encodings, in the interest of reducing memory usage at the cost of potentially increased running times. We note that, since CLT enjoys much more parallelism than GGHLite when constructing encodings, this optimization causes the encrypt time for GGHLite to be less efficient. Finally, for eval, we multiply encodings in parallel for CLT, since the multiplication of CLT encodings natively does not support parallelism. However, for GGHLite, we choose to multiply encodings sequentially, and instead rely on the parallelism afforded by GGHLite encoding multiplication.

The ciphertexts produced by a call to encrypt on an ℓ-length input are split into ℓ components (one for each input slot), which can be easily separated and combined with different components from other ciphertexts in a later call to eval. Hence, with a collection of full ciphertexts, an evaluator can specify which components from each ciphertext should be passed as input to eval, in order to evaluate the function on components originating from different sources.

6.1 Optimizing Comparisons

In this section, we describe a case study of applying the optimizations detailed in §4 to the comparison function. We establish two distinct "variants" of the comparison function which result in shorter ciphertext sizes. Both variants are built from a combination of condensing the input representation into a larger base $d > 2$, followed by dimension reduction, and optionally applying matrix pre-multiplication.

- **DC-variant.** The *degree-compressed* optimization is to first apply matrix pre-multiplication to re-order the reading of the input bits as in Equation (1). Then, the dimensions of the resulting matrices from the layered state machine are slightly reduced.
- **MC-variant.** The *matrix-compressed* optimization is to directly apply dimension reduction in the normal interleaved ordering of the bits (as $x_1 y_1 x_2 y_2 \cdots x_n y_n$). Here, the dimensions of the matrices can be reduced to depend only linearly in the base representation d, as opposed to quadratically.

We now discuss each optimization in more detail.

The degree-compressed variant (DC-variant). By optimizing the (layered) comparison state machine, we obtain that not all matrices need to be of dimension $(d+3) \times (d+3)$. For example, the first matrix need only be of dimension $1 \times d$, and the second matrix need only be of dimension $d \times (d+2)$. Also, the last matrix can be of dimension $(d+2) \times 3$. And finally, each of the remaining intermediate matrices need only be of dimension $(d+2) \times (d+2)$. Putting these observations together, the total number of encodings in the ciphertext is

$$\begin{aligned} M &= d + d(d+2) + 3(d+2) + (\kappa - 3)(d+2)^2 \\ &= d^2(\kappa - 2) + (d+1)(4\kappa - 6). \end{aligned} \tag{2}$$

The matrix-compressed variant (MC-variant). Note, however, that if we do not apply the matrix pre-multiplication optimization, but instead apply dimension reduction directly to the comparison state machine associated with the normal (not interleaved) ordering of the input digits, then the first matrix is of dimension $1 \times d$, the second matrix is of dimension $d \times 3$, and all other $\kappa - 2$ matrices are of dimension either $3 \times (d+2)$ or $(d+2) \times 3$. Putting this together, we have $\kappa = 2n$ and

$$\begin{aligned} M &= d + 3d + 3(\kappa - 2)(d+2) \\ &= 3(\kappa - 2)(d+2) + 4d. \end{aligned} \tag{3}$$

Concretely, for a domain of size $N = 10^{12}$, if we choose to represent the inputs in base $d = 5$, then this implies $n = 18$ (since $12 \leq \log_{10}(5^{18}) < 13$), and hence with the matrix pre-multiplication optimization along with the Cryptol optimization for dimension reduction, we have $\kappa = 19$ and $M = 845$. Without applying matrix pre-multiplication and only using dimension reduction with base $d = 10$, we have $n = 12$, $\kappa = 24$, and $M = 832$.

Experimentally for the mmaps we tested, we found that the matrix pre-multiplication optimization produces shorter ciphertexts than applying dimension reduction without matrix pre-multiplication. However, we only tested this for an input domain of $N = 10^{12}$ and security parameter $\lambda = 80$. As N grows larger, depending on the asymptotic behavior of encoding sizes

as κ increases and λ varies, future implementations of the comparison state machine may find that one can produce shorter ciphertexts when applying dimension reduction without matrix pre-multiplication.

6.2 Order-Revealing Encryption

To implement order-revealing encryption, we set our plaintext domain to the numbers in the range $[N]$. By taking $N = 10^{12}$, we found that selecting the base representation $d = 5$ and applying the matrix pre-multiplication optimization resulted in using only $\kappa = 19$ levels of the underlying mmap, which achieved the shortest ciphertexts for this domain. In fact, this construction yields the shortest known ciphertexts for ORE on a domain of size 10^{12}, as explained below.

An alternative (basic) construction. The closest competitor to our ORE construction in terms of ciphertext size and overall efficiency is a construction due to Lewi and Wu [LW16], which we refer to as the "basic" ORE scheme, described below.

Let $[N]$ be the message space. Let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a secure pseudorandom function (PRF) and $H : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}$ be a hash function (modeled as a random oracle). Let CMP be the comparison function, defined as $\textsc{cmp}(x, y) = 1$ if $x < y$ and $\textsc{cmp}(x, y) = 0$ if $x > y$. The basic ORE scheme Π_{ore} is defined as follows.

- $\mathsf{keygen}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The algorithm samples a PRF key $k \xleftarrow{R} \{0,1\}^\lambda$ for F, and a random permutation $\pi : [N] \to [N]$. The secret key sk is the pair (k, π), and there are no public parameters.

- $\mathsf{encrypt}(\mathsf{sk}, i, x) \to \mathsf{ct}$. Write sk as (k, π). If $i = 1$, the ciphertext output is simply $\mathsf{ct} = (F(k, \pi(x)), \pi(x))$. If $i = 2$, then the encryption algorithm samples a nonce $r \xleftarrow{R} \{0,1\}^\lambda$, and for $j \in [N]$, it computes $v_j = \textsc{cmp}(\pi^{-1}(j), y) \oplus H(F(k, j), r)$. Finally, it outputs $\mathsf{ct} = (r, v_1, v_2 \ldots, v_N)$.

- $\mathsf{eval}(\mathsf{pp}, \mathsf{ct}_1, \mathsf{ct}_2) \to \{0,1\}$. The compare algorithm first parses $\mathsf{ct}_1 = (k', h)$ and $\mathsf{ct}_2 = (r, v_1, v_2, \ldots, v_n)$, then outputs $v_h \oplus H(k', r)$.

Note that a single ciphertext from this scheme is precisely $N + 2\lambda + \lceil \log_2(N) \rceil$ bits long. For $N = 10^{12}$ and $\lambda = 80$, this amounts to ciphertexts of length 116.42 GB.[5]

Choosing the best optimizations. Our goal is to construct an ORE scheme which achieves shorter ciphertexts than the above construction, without compromising security. To do this, we use our MIFE implementation for the comparison function, and we apply our optimizations to make the ciphertext as short as possible.

We compare the ciphertext sizes for four different ORE constructions, obtained from either using the GGHLite or CLT mmap, and by applying either the DC-variant or MC-variant optimizations. For each of these options, we fix the input domain size $N = 10^{12}$ and vary the input base representation $d \in [2, 25]$. Using Equations (2) and (3), we can compute the estimated ciphertext size as a function of d (since κ is determined by the choice of d and N). See Figure 6.1 for the results. We find that, for $N = 10^{12}$, the shortest ciphertexts for ORE from GGHLite are obtained when $d = 5$ using the DC-variant optimization, and the shortest ciphertexts for ORE from CLT are obtained when $d = 4$ using the DC-variant optimization as well.

[5] Clearly, increasing λ has a relatively unnoticeable effect on the overall ciphertext size for the settings of N we consider.

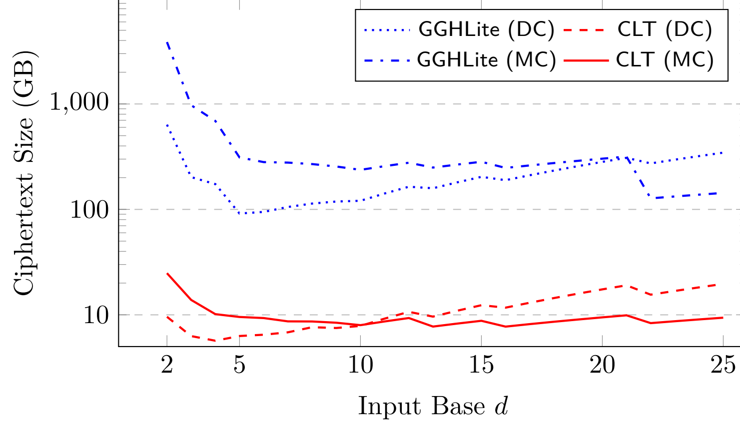

Figure 6.1: Estimates of the ciphertext size (in *GB*) for ORE with best-possible semantic security at $\lambda = 80$, for domain size $N = 10^{12}$ and for bases $d \in [2, 25]$. We compare GGHLite and CLT, with the DC-variant and MC-variant optimizations.

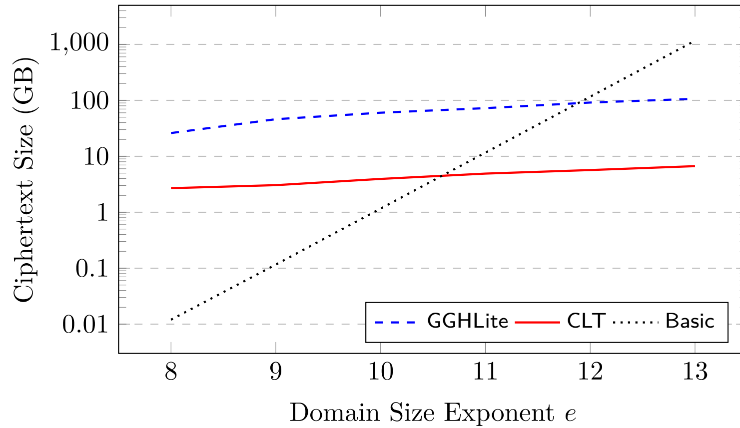

Figure 6.2: Estimates of the ciphertext size (in *GB*) for ORE with best-possible semantic security at $\lambda = 80$, for varying domain sizes. The exponent e on the x-axis denotes support for plaintexts in the range from 1 to $N = 10^e$. We compare GGHLite map (DC-variant), the CLT map, and the basic construction Π_{ore} (described in §6.2).

Under these settings, the DC-variant optimization for GGHLite reads the inputs in base 5, requiring $\kappa = 19$, to produce a total of 845 encodings per ciphertext, for a total size of 91.4 GB. For CLT, the DC-variant optimization reads in the inputs in base 4, requiring $\kappa = 21$, to produce a total of 694 encodings, for a total size of 5.68 GB.

We also measure the ciphertext size as we vary the domain size; see Figure 6.2. We measure the estimated ciphertext size for various domain sizes when using GGHLite, CLT, and the Π_{ore} construction described above. The results for GGHLite and CLT are using the optimal bases as detailed in Figure 6.1. We find that for $N = 10^{11}$ and $N = 10^{12}$, ORE using the CLT mmap and GGHLite mmap, respectively, produces a smaller ciphertext than Π_{ore}. This demonstrates that for certain domain sizes, our ORE construction produces the smallest known ciphertexts (versus ORE schemes that do not require mmaps).

6.3 Three-Input DNF Encryption

We now explore the applications of MIFE to a function on three inputs, which we call the 3DNF function. For n-bit inputs $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$, and $z = z_1 \cdots z_n$, the 3DNF function is defined as

$$\mathsf{3DNF}(x, y, z) = (x_1 \wedge y_1 \wedge z_1) \vee \cdots \vee (x_n \wedge y_n \wedge z_n) \in \{0, 1\}.$$

This function bears resemblance to the "tribes" function studied by Gentry et al. [GLW14], who also use mmaps to construct tribe instances. We refer to a MIFE scheme for the 3DNF function as a *3DNF encryption scheme*, and we refer to each ciphertext as consisting of three components, one for each input slot: the left encryption, middle encryption, and right encryption. To the best of our knowledge, 3DNF encryption schemes do not appear to follow directly from simpler cryptographic assumptions.

Application to fraud detection. An immediate application of 3DNF encryption is in the fraud detection of encrypted transactions. Consider the scenario where a (stateless) user makes payments through transactions that are audited by a payment authority. In this setting, each transaction is associated with a string of n flags, represented as bits pertaining to a set of n properties of the transaction. A payment authority, in the interest of detecting fraud, restricts the user to make at most (say) $\ell = 3$ transactions per hour, and wants to raise an alarm if a common flag is set in all ℓ of the transactions made in the past hour (if less than ℓ transactions were made in the past hour, then the authority does not need to perform a check).

To protect the privacy of the user, the length-n flag string associated with each transaction can be sent to the payment authority as encrypted under a 3DNF encryption scheme, where the stateless user holds the decryption key. Here, the user would send a left encryption for the first transaction, a middle encryption for the second, and a right encryption for the third. Then, since the payment authority cannot decrypt any of the flag strings for the transactions, the privacy of the user's transactions is protected. However, the payment authority can still perform the fraud detection check by evaluating a set of ℓ transactions to determine if they satisfy the 3DNF function. Since we require that the user is stateless between transactions, this application fits the model for a 3DNF encryption scheme, and does not seem to directly follow from simpler primitives.

Optimizing 3DNF encryption. Similar to the case of the comparison function, we can apply the branching program optimizations to the 3DNF function as well, in order to reduce the overall efficiency of the resulting 3DNF encryption scheme. We constructed a 3DNF encryption scheme using our MIFE implementation, for $n = 16$ bit inputs at security parameter $\lambda = 80$. We optimized the 3DNF encryption scheme by condensing the input representation into base $d = 4$. Additionally, we applied the matrix pre-multiplication optimization, which meant that our input bits were read in the order $x_1 y_1 z_1 z_2 y_2 x_2 x_3 y_3 \cdots$ (the natural generalization of the interleaving of Equation (1) to three inputs). This resulted in a setting of degree $\kappa = 17$ for the underlying mmap. Finally, we used `cryfsm` to generate the corresponding MBP, which automatically applied the appropriate dimension reduction optimizations. Under the CLT mmap, a left encryption is 637 MB, a middle encryption is 1.4 GB, and a right encryption is 680 MB.

7 Program Obfuscation

A *program obfuscator* [BGI+01, GGH+13b] is a compiler that aims to make a program "unintelligible" while preserving its functionality. Formally, an obfuscator $\mathcal{O}$ is a tuple of algorithms written as

$\mathcal{O} = (\mathsf{obf}, \mathsf{eval})$, where the obfuscation algorithm obf takes as input a program P (e.g., expressed as a Boolean circuit), and outputs an obfuscated program $\mathsf{obf}(\mathsf{P})$, and the evaluation algorithm takes an obfuscated program $\mathsf{obf}(\mathsf{P})$ and produces an output. An obfuscator is *correct* for a program P if, for all valid inputs x accepted by P, we have that $\mathsf{eval}(\mathsf{obf}(\mathsf{P}))(x) = \mathsf{P}(x)$.

VBB and pseudo-VBB security. An obfuscator $\mathcal{O}$ is *virtual black-box (VBB) secure*[6] for a program P if for any efficient adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$, given only oracle access to $\mathsf{P}(\cdot)$, for which the quantity

$$\left|\Pr\left[\mathcal{A}(1^\lambda, \mathsf{obf}(\mathsf{P})) = 1\right] - \Pr\left[\mathcal{S}^{\mathsf{P}(\cdot)}(1^\lambda) = 1\right]\right|$$

is negligible. We say that an obfuscator $\mathcal{O}$ is *pseudo-VBB secure* (pVBB) for program P and obfuscator $\mathcal{O}' = (\mathsf{obf}', \mathsf{eval}')$ if $\mathcal{O}'$ is both VBB secure and for every efficient adversary $\mathcal{A}$, there exists an efficient adversary $\mathcal{B}$ for which the quantity

$$\left|\Pr\left[\mathcal{A}(1^\lambda, \mathsf{obf}(\mathsf{P})) = 1\right] - \Pr\left[\mathcal{B}(1^\lambda, \mathsf{obf}'(\mathsf{P})) = 1\right]\right|$$

is negligible. In other words, if an obfuscator $\mathcal{O}$ is pVBB secure for a program P and obfuscator $\mathcal{O}'$, then any efficient attack on the security of $\mathcal{O}$ translates directly to an efficient attack on the VBB security of $\mathcal{O}'$ for the program P.

In our work, we construct a point function obfuscator that is pVBB secure for the Sahai-Zhandry obfuscator [SZ14]. Our obfuscator operates identically to the Sahai-Zhandry obfuscator, which is VBB secure, except that we discard half of the ciphertext that corresponds to the second input in the "dual-input" branching programs that obfuscator uses. Effectively, our obfuscator can be seen as operating on "single-input" branching programs, which do not obtain VBB security, but do obtain pseudo-VBB security. We emphasize that this distinction in security is purely definitional from an attacker's point of view, as any attack on our obfuscator immediately results in an attack on the Sahai-Zhandry obfuscator.

In this section we show how we use `cryfsm` and `libmmap` to build such a program obfuscator. Apon et al. [AHKM14] gave the first implementation of program obfuscation, using the CLT mmap [CLT13] and a program compiler based on the approaches of Barak et al. [BGK+14] and Ananth et al. [AGIS14]. We extend this codebase in the following ways:

- **Multilinear maps.** We integrate in `libmmap` to support both the CLT and GGHLite mmaps.
- **Program compilers.** We support MBPs output by `cryfsm`, using the Sahai-Zhandry obfuscator [SZ14].

Point function obfuscation. We evaluated our implementation by obfuscating point functions, namely, functions that output 0 on a single (secret) input, and 1 otherwise. Previous work [AHKM14] also evaluated obfuscation for point functions, but was only able to successfully obfuscate 14-bit point functions with an mmap security parameter of $\lambda = 60$. As noted by Bernstein et al. [BHLN15], the secret input of an n-bit point function can be recovered by simply enumerating over all 2^n possible inputs. In our experiments, we set $n = \lambda$, and consider point function obfuscation for 40-bit and 80-bit inputs.

[6] The reason we consider VBB versus *indistinguishability* obfuscation is that we consider point functions, for which VBB obfuscators are believed to exist.

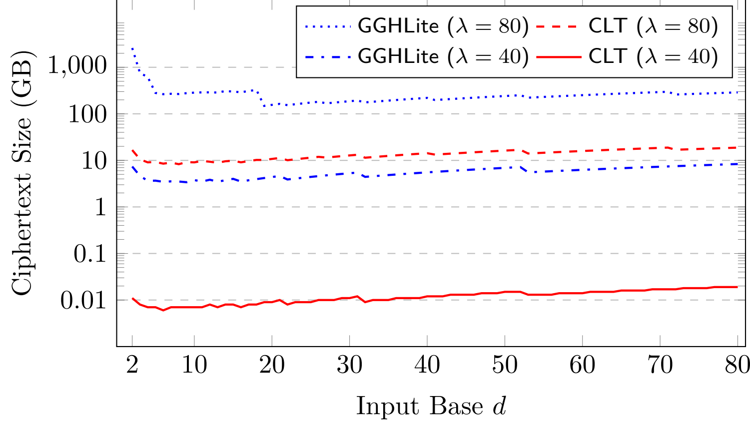

Figure 7.1: Estimates for the ciphertext size (in GB) for point function obfuscation, for domain sizes $N = 2^{80} = 2^{\lambda}$ and $N = 2^{40} = 2^{\lambda}$. In the case of $\lambda = 80$, the minimums are achieved at $d = 19$ for GGHLite and $d = 8$ for CLT. In the case of $\lambda = 40$, the minimums are achieved at $d = 9$ for GGHLite and $d = 6$ for CLT.

The MBP for a λ-bit point function is of length λ and consists of a total of 2λ matrices, each of dimension 2×2. As a small optimization, we can apply dimension reduction to obtain a branching program where the first pair of matrices need only be of dimension 1×2. The more significant optimization comes by condensing the input representation through increasing the input base d.

The total number of encodings that we must publish in the obfuscation of a λ-bit point function can be computed as $M = 2 + 4 \cdot d \cdot \ell$, where ℓ is the length of the MBP. We estimate the ciphertext size for various choices of bases in Figure 7.1, which incorporates our estimations for the size of a single encoding in GGHLite and CLT for $\lambda = 40$ and $\lambda = 80$.

- For $\lambda = 40$, we find that the minimal ciphertext size for domain size $N = 2^{40}$ is produced using MBPs under base 9 and length 13 for GGHLite, and base 6 and length 16 for CLT.
- For $\lambda = 80$, we find that the minimal ciphertext size for domain size $N = 2^{80}$ is produced using MBPs under base 19 and length 19 for GGHLite, and base 8 and length 27 for CLT.

Obfuscator implementation. Our implementation is in a mix of Python and C, with Python handling the frontend and with C handling all the computationally expensive portions, and provides interfaces to both obfuscate (obf) and evaluate (eval) an MBP. We parallelize the encoding of the elements in the MBP by using a threadpool and delegating each encoding operation to a separate thread. Once all the threads for a given matrix in the MBP complete, we then write the (encoded) matrix to disk. Thus, the threadpool approach has a higher RAM usage (due to keeping multiple encodings in memory as we parallelize) than encoding one element at a time and letting the underlying mmap library handle the parallelization, but is more efficient.

Other obfuscators. Our obfuscator is built upon improvements inspired by the Sahai-Zhandry obfuscator, which is built on the general obfuscator described by Barak et al. [BGK+14] and Ananth et al. [AGIS14]. In addition to these obfuscators, we also implemented the Zimmerman [Zim15] obfuscator. However, because the Zimmerman obfuscator induces a seemingly unavoidable lower bound on the degree of multilinearity for the inputs we consider, we found that the Zimmerman

obfuscator was not competitive with the obfuscator we implemented. More specifically, the Zimmerman obfuscator requires that the degree of multilinearity for the obfuscation of *any* program be at least twice the number of inputs that the circuit accepts—a cost that may be insignificant when obfuscating other programs, but was too high for point functions (even when we tried to increase the input base representation to minimize this cost), and hence unsuitable for our purposes.

8 Experimental Analysis

All of our experiments were performed using the Google Compute Engine servers with a 32-core Intel Haswell CPU at 2.5 GHz, 208 GB RAM, and 100 GB disk storage.

8.1 MIFE Experiments

We evaluated our multi-input functional encryption constructions with two applications: order-revealing encryption (ORE) (cf. §6.2) and three-input DNF (3DNF) encryption (cf. §6.3). In §6, we showed how we can accurately estimate the ciphertext size from parameters derived from the input size and the security parameter λ, and our experiments confirmed that these parameter estimates are reasonably accurate (all within 1–2% of our reported values).

Additionally, we assessed the performance of the MIFE interface algorithms keygen, encrypt, and eval, along with memory utilization during the encrypt computation, which was by far the most costly step. We note that, since the files that we are working with are so large, a non-trivial amount of time was spent in the reading and writing of these files to disk, and so an exact reproduction of our numbers may also need to mimic the disk storage specification we use.

As another sidenote, we reiterate that our primary interest in selecting the parameters for our applications is to create the most compact ciphertexts possible. As a result, some of our optimizations come with a cost of increased evaluation time, and hence, we believe that it is possible to reduce our evaluation time (potentially at the expense of having larger ciphertexts).

Experimental results. We summarize our MIFE experiments in Tables 8.1 and 8.2. We evaluated the MIFE constructions for ORE with input domain sizes $N = 10^{10}$ and $N = 10^{12}$, and for 3DNF encryption on 8-bit inputs, testing both GGHLite and CLT as the underlying mmap. For each experiment, we report the computation wall time for encrypt and eval, the overall ciphertext size $|\mathsf{ct}|$, along with the memory usage during the encrypt computation. The keygenoperation varied from several seconds (for CLT with $\lambda = 40$) to 145 minutes (for GGHLite with $\lambda = 80$). The encryption statistics measured were for generating a complete ciphertext, containing all components, as opposed to containing only the left or right (or middle) components.

Since the CLT mmap produces shorter encodings, the encryption and evaluation time for the experiments using CLT were much faster than the corresponding experiments for GGHLite. This is also partly due to the fact that CLT enjoys much more parallelism than GGHLite. We also only present timings for CLT with $\lambda = 80$ because we ran out of RAM during the encryption procedure when using GGHLite.

8.2 Program Obfuscation Experiments

To evaluate our program obfuscation implementation, we chose a random secret 40-bit and a random secret 80-bit point, and used `cryfsm` to create the corresponding MBPs for the point functions associated with these points. We selected the input base representation for these programs with

λ	mmap	N	d	ℓ	encrypt	eval	|ct|	RAM
40	CLT	10^{10}	4	19	1 s	0.3 s	13 MB	17 MB
		10^{12}	4	22	3 s	1.6 s	18 MB	18 MB
	GGH	10^{10}	4	19	38 m	47 s	7.1 GB	23 GB
		10^{12}	4	22	52 m	68 s	9.6 GB	25 GB
80	CLT	10^{10}	4	19	28 m	4 m	4.7 GB	5 GB
		10^{12}	4	22	37 m	6 m	6.0 GB	6 GB

Table 8.1: ORE experiments. "λ" denotes the security parameter of the underlying multilinear map; "mmap" denotes the multilinear map; "N" denotes the domain size; "d" denotes the MBP base; "ℓ" denotes the MBP length; "encrypt" denotes the running time of encryption; "eval" denotes the running time of evaluation, "|ct|" denotes the size of the ciphertext; and "RAM" denotes the RAM required to encrypt. We use "h" for hours, "m" for minutes, and "s" for seconds.

λ	mmap	N	d	ℓ	encrypt	eval	|ct|	RAM
40	CLT	16-bit	4	17	0.6 s	0.2 s	7.4 MB	18 MB
	GGH	16-bit	4	17	20 m	28 s	3.9 GB	22 GB
80	CLT	16-bit	4	17	12 m	3 m	2.5 GB	4 GB

Table 8.2: 3DNF experiments. See Table 8.1 for the column details.

the goal of minimizing the total obfuscation size for each obfuscated point function (see §7 for our calculations). Like with MIFE, optimizing for obfuscation or evaluation time could lead to different optimal input base representations.

Experimental results. We tested three settings for point function obfuscation: 40-bit inputs with $\lambda = 40$, 80-bit inputs with $\lambda = 40$, and finally, 80-bit inputs with $\lambda = 80$. We also obfuscated using both CLT and GGHLite for $\lambda = 40$, but only used CLT for $\lambda = 80$, as the GGHLite experiment was too resource-intensive. Our results are summarized in Table 8.3. As we observed in the MIFE experiments, we note that GGHLite performs significantly worse when used in obfuscation compared to CLT. We also note that while obfuscation takes a huge amount of time and resources, evaluation is much less resource-intensive, for both GGHLite and CLT—a consequence of the fact that eval only requires multiplying (encoded) matrices, which is highly parallelizable and also much less costly than the encoding operation itself.

These results, while evidently impractical, are a huge improvement over prior work [AHKM14], which took 7 hours to obfuscate a 14-bit point function with $\lambda = 60$, resulting in an obfuscation of 31 GB. This improvement mainly come from (1) using a much tighter matrix branching program representation of the program, and (2) operating over different sized bases.

9 Conclusions

In this work, we presented `5Gen`, a framework for the prototyping and evaluation of applications that use multilinear maps (mmaps) and matrix branching programs. `5Gen` is built as a multi-layer software stack which offers modularity and easy integration of new constructions for each component type. Our framework offers an optimized compiler that converts programs written

λ	mmap	N	d	ℓ	obf	eval	|obf|	RAM
40	CLT	40-bit	6	16	1.7 s	0.1 s	6.3 MB	1.7 GB
		80-bit	7	29	6.6 s	0.3 s	21.7 MB	1.7 GB
	GGH	40-bit	9	13	28 m	5.9 s	3.5 GB	38 GB
		80-bit	6	31	56 m	39 s	13.7 GB	37 GB
80	CLT	80-bit	8	27	3.3 h	180 s	8.3 GB	11 GB

Table 8.3: Program obfuscation experiments. "λ" denotes the security parameter of the underlying multilinear map; "mmap" denotes the multilinear map; "N" denotes the domain size; "d" denotes the MBP base; "ℓ" denotes the MBP length; "obf" denotes the obfuscation time; "eval" denotes the evaluation time; "|obf|" denotes the obfuscation size; and "RAM" denotes the RAM required to obfuscate (evaluation RAM usage never exceeded 1 GB). We use "h" for hours, "m" for minutes, and "s" for seconds.

in the Cryptol language into matrix branching programs, a representation widely used in mmap-based constructions. `5Gen` includes a library of mmaps available through a common API; we currently support the GGHLite and CLT mmaps, but our library can be easily extended with new candidates. Leveraging the capabilities of our compiler and mmap libraries, we implemented applications from two computing paradigms based on mmaps: multi-input functional encryption (MIFE) and obfuscation.

We measured the efficiency of our MIFE and obfuscation applications with various parameter settings using both the GGHLite and CLT mmaps. While the results show efficiency that is clearly not usable in practice, they provide a useful benchmark for the current efficiency of these techniques.

Constructing multilinear maps is an active and rapidly-evolving area of research. Our `5Gen` framework provides an easy-to-use testbed to evaluate new mmap candidates for various applications and is open-source and freely available at `https://github.com/5GenCrypto`.

Acknowledgments

The work of Dan Boneh and Kevin Lewi was supported by NSF, DARPA, a grant from ONR, and the Simons Foundation. The work of Daniel Apon, Jonathan Katz, and Alex J. Malozemoff was supported in part by NSF awards #1111599 and #1223623. The work of Alex J. Malozemoff was conducted in part with Government support through the National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFG 168a, awarded by DoD, Air Force Office of Scientific Research. The work of Brent Carmer and Mariana Raykova was supported by NSF grants CNS-1633282, 1562888, 1565208, and DARPA SafeWare W911NF-15-C-0236.

This material is based upon work supported by the ARO and DARPA under Contract No. W911NF-15-C-0227. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO and DARPA.

References

[AB15] Benny Applebaum and Zvika Brakerski. Obfuscating circuits via composite-order graded encoding. In *TCC*, 2015.

[ABD16] Martin Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions: Cryptanalysis of some FHE and graded encoding schemes. 2016.

[ACLL15] Martin R. Albrecht, Catalin Cocis, Fabien Laguillaumie, and Adeline Langlois. Implementing candidate graded encoding schemes from ideal lattices. In *Asiacrypt*, 2015.

[AGIS14] Prabhanjan Vijendra Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai. Optimizing obfuscation: Avoiding Barrington's theorem. In *CCS*, 2014.

[AHKM14] Daniel Apon, Yan Huang, Jonathan Katz, and Alex J. Malozemoff. Implementing cryptographic program obfuscation. Cryptology ePrint Archive, Report 2014/779, 2014.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. `http://eprint.iacr.org/2015/046`.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Crypto*, 2001.

[BGK+14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In *Eurocrypt*, 2014.

[BHLN15] Daniel J. Bernstein, Andreas Hülsing, Tanja Lange, and Ruben Niederhagen. Bad directions in cryptographic hash functions. In *ACISP*, 2015.

[BLR+15] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *Eurocrypt*, 2015.

[BMSZ15] Saikrishna Badrinarayanan, Eric Miles, Amit Sahai, and Mark Zhandry. Post-zeroizing obfuscation: The case of evasive circuits. Cryptology ePrint Archive, Report 2015/167, 2015.

[BR14] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In *TCC*, 2014.

[BS02] Dan Boneh and Alice Silverberg. Applications of multilinear forms to cryptography. Cryptology ePrint Archive, Report 2002/080, 2002.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *TCC*, 2011.

[BWZ14a] Dan Boneh, Brent Waters, and Mark Zhandry. Low overhead broadcast encryption from multilinear maps. Cryptology ePrint Archive, Report 2014/195, 2014.

[BWZ14b] Dan Boneh, David J. Wu, and Joe Zimmerman. Immunizing multilinear maps against zeroizing attacks. Cryptology ePrint Archive, Report 2014/930, 2014.

[CGH$^+$15] Jean-Sébastien Coron, Craig Gentry, Shai Halevi, Tancrède Lepoint, Hemanta K. Maji, Eric Miles, Mariana Raykova, Amit Sahai, and Mehdi Tibouchi. Zeroizing without low-level zeroes: New MMAP attacks and their limitations. In *Crypto*, 2015.

[CHL$^+$15] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In *Eurocrypt*, 2015.

[CJL16] Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An algorithm for NTRU problems and cryptanalysis of the GGH multilinear map without an encoding of zero. Cryptology ePrint Archive, Report 2016/139, 2016.

[CLT13] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In *Crypto*, 2013.

[CLT15] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. New multilinear maps over the integers. In *Crypto*, 2015.

[Cry] Cryptol. `http://cryptol.net/`. Accessed: 2016-05-02.

[DGG$^+$16] Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Pratyay Mukherjee. Obfuscation from low noise multilinear maps. Cryptology ePrint Archive, Report 2016/599, 2016.

[GGG$^+$14] Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In *Eurocrypt*, 2014.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *Eurocrypt*, 2013.

[GGH$^+$13b] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, 2013.

[GGH15] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In *TCC*, 2015.

[GGHZ14] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. Fully secure attribute based encryption from multilinear maps. Cryptology ePrint Archive, Report 2014/622, 2014.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *STOC*, 2013.

[GLSW15] Craig Gentry, Allison Bishop Lewko, Amit Sahai, and Brent Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. In *FOCS*, 2015.

[GLW14] Craig Gentry, Allison B. Lewko, and Brent Waters. Witness encryption from instance independent assumptions. In *Crypto*, 2014.

[GMS16] Sanjam Garg, Pratyay Mukherjee, and Akshayaram Srinivasan. Obfuscation without the vulnerabilities of multilinear maps. Cryptology ePrint Archive, Report 2016/390, 2016.

[HJ16] Yupu Hu and Huiwen Jia. Cryptanalysis of GGH map. In *Eurocrypt*, 2016.

[Lep14] Tancrède Lepoint. *Design and Implementation of Lattice-based Cryptography*. PhD thesis, Université du Luxembourg, May 2014.

[Lin16] Huijia Lin. Indistinguishability obfuscation from constant-degree graded encoding schemes. In *Eurocrypt*, 2016.

[LPST16] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Indistinguishability obfuscation with non-trivial efficiency. In *PKC*, 2016.

[LS14] Hyung Tae Lee and Jae Hong Seo. Security analysis of multilinear maps over the integers. In *Crypto*, 2014.

[LSS14] Adeline Langlois, Damien Stehlé, and Ron Steinfeld. GGHLite: More efficient multilinear maps from ideal lattices. In *Eurocrypt*, 2014.

[LW16] Kevin Lewi and David J. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *CCS*, 2016.

[MSZ16a] Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In *Crypto*, 2016.

[MSZ16b] Eric Miles, Amit Sahai, and Mark Zhandry. Secure obfuscation in a weak multilinear map model: A simple construction secure against all known attacks. Cryptology ePrint Archive, Report 2016/588, 2016.

[PST13] Rafael Pass, Karn Seth, and Sidharth Telang. Indistinguishability obfuscation from semantically-secure multi-linear encodings. Cryptology ePrint Archive, Report 2013/781, 2013.

[SZ14] Amit Sahai and Mark Zhandry. Obfuscating low-rank matrix branching programs. Cryptology ePrint Archive, Report 2014/773, 2014.

[Zim15] Joe Zimmerman. How to obfuscate programs directly. In *Eurocrypt*, 2015.

A Parameter Selection

In this section, we discuss the parameter selection for both the GGHLite and CLT mmaps that we consider in this work. Throughout, we use λ as the security parameter and κ as the multilinearity parameter.

A.1 GGHLite

We now discuss the parameter selection used for the GGHLite mmap discussed in §5.1. We discuss the various parameters, and where applicable, the values we set them to and why. As we build off of the code from Albrecht et al. [ACLL15], many of these parameter settings come directly from that work; we simply document them explicitly here.

- n: The dimension of the lattice. We set n by iteratively producing the below parameters and then checking whether they satisfy the security parameter according to [APS15], increasing n by a power of two each iteration (cf. [ACLL15, §4.4]).
- σ: The Gaussian parameter. We set $\sigma = 4\pi n\sqrt{e \ln(8n)/\pi}$ according to [LSS14, Eq. (4)].
- ℓ_g^{-1}: The upper bound on the size of $\|g^{-1}\|$. We set $\ell_g^{-1} = 4\sqrt{e\pi n}/\sigma$ according to [LSS14, Eq. (4)].
- ℓ: The number of bits to extract from the level-κ encoding. We set $\ell = \lceil \log_2(8\sigma n) \rceil$ according to the discussion in [ACLL15, §4.2].
- σ': The Gaussian parameter for encoding values. We set $\sigma' = \max \left\{ \begin{matrix} 2n^{1.5}\sigma\sqrt{e\log(8n)/\pi} \\ 7n^{2.5}\ln^{1.5}(n)\sigma \end{matrix} \right\}$ according to the discussion in [LSS14, §6].
- q: The modulus. Selected according to [ACLL15, §4.2, pg. 10].

A.2 CLT

We now discuss the parameter selection used for the CLT mmap discussed in §5.2. As above, we discuss the various parameters, and where applicable, the values we set them to and why.

- ρ: The bitlength of the randomness used for encodings. We set $\rho = \lambda$ to avoid the attack of Lee and Seo [LS14].
- α: The bitlength of the message slots g_i. We set $\alpha = \lambda$ as suggested by [CLT13].
- β: The bitlength of the random h_i values. We set $\beta = \lambda$ to avoid a GCD attack similar to [CLT13, §5.2].
- ρ_f: The maximum bitlength of the randomness in a level-κ encoding. For the specific usecase of obfuscation we can set $\rho_f = \kappa(\rho + \alpha)$.
- n: The number of secret primes p_i. In Appendix B we show how to adapt the lattice attack on encodings by Coron et al. [CLT13, §5.1] to the "general" setting where no 0-level encodings of zero are available, and thus we need to set $n = \omega(\eta \log \lambda)$. However, rather than setting n to match some asymptotic, such as $\eta\lambda$, we consider the concrete costs of the various attacks to give an accurate estimate of n. We use the approach detailed by Tancrède Lepoint [Lep14, §7.2] using the more conservative estimate for the running time of the LLL algorithm [Lep14, §7.2.5].
- η: The bitlength of the secret primes p_i. We set $\eta = \rho_f + \alpha + \beta + \log_2(n) + 9$ according to [CLT13, Lemma 8].

- ν: The bitlength of the image of the mmap. We set $\nu = \eta - \beta - \rho_f - \lambda - 3$ according to [CLT13, Lemma 8].

Note that n, η, and ν depend on each other. Thus, to compute these values we simply loop until we reach a fixed point, using the requirement that $\beta + \alpha + \rho_f + \log_2(n) \leq \eta - 9$ and $\nu \geq \alpha + 6$, as detailed in [CLT13, Lemma 8].

B Lattice Attack on Encodings

In this section we describe how to adapt the lattice attack on encodings by Coron et al. [CLT13, §5.1] to our particular use of the CLT mmap.[7] In particular, we consider the case where an attacker has access to $t > n$ level-1 encodings. Without loss of generality, we assume a single z.

Let $x_0 = \prod_{i=1}^{n} p_i$. For $j \in [t]$ consider the level-1 encoding $x'_j = x_j/z \bmod x_0$ of secret message $\boldsymbol{m}_j = (m_{ij})_i$, where $x_j \in \mathbb{Z}_{x_0}$ is such that $x_j \bmod p_i = r_{ij} g_i + m_{ij}$. Note that $x_j \bmod p_i$ is of size $\rho + \alpha$ bits and can be considered as a level-0 encoding of $\boldsymbol{m}_j$. Let $\boldsymbol{x'} = (x'_j)_j$ and let $\boldsymbol{x} = (x_j)_j$.

As detailed by Coron et al., we want to use $\boldsymbol{x'}$ to relate the lattice of vectors $\boldsymbol{u}$ orthogonal to $\boldsymbol{x'} \bmod x_0$ to the lattice of vectors orthogonal to each "plaintext" vector $\boldsymbol{s}_i = (s_{ij})_j = (r_{ij} g_i + m_{ij})_j$. We do so as follows.

By construction we have that

$$\begin{aligned} \boldsymbol{u} \cdot \boldsymbol{x'} &\equiv 0 \pmod{x_0} \\ \iff z(\boldsymbol{u} \cdot \boldsymbol{x'}) &\equiv 0 \pmod{x_0} \\ \iff \boldsymbol{u} \cdot \boldsymbol{x} &\equiv 0 \pmod{x_0}, \end{aligned}$$

where the last equivalence comes from the fact that z and x_0 are coprime with high probability. We can thus apply the attack detailed by Coron et al. [CLT13, §5.1] on vector $\boldsymbol{x'}$ to directly recover the vectors $\boldsymbol{s}_i$.

Now, given these vectors, we can recover p_i for $i \in [n]$ with high probability as follows [CHL+15]. Note that

$$\begin{aligned} \frac{x'_1}{s_{i1}} &\equiv \frac{x'_2}{s_{i2}} \pmod{p_i} \\ \iff x'_1 s_{i2} - x'_2 s_{i1} &\equiv 0 \pmod{p_i}. \end{aligned}$$

Thus, we can compute $\gcd(x'_1 s_{i2} - x'_2 s_{i1}, x_0)$ to learn p_i.

[7] We thank Tancrède Lepoint for detailing this adaptation.

Changelog

- Version 2.1 (November 15, 2016):
  - Removed erroneous reference to 128-core machine in Table 8.3.
- Version 2.0 (October 26, 2016):
  - Updated tables and graphs.
  - Added appendices.
  - Edits throughout.
- Version 1.0 (June 14, 2016): First release.