AN ABSTRACT OF THE DISSERTATION OF

Brent Carmer for the degree of Doctor of Philosophy in Computer Science presented on September 19, 2018.

Title: Optimizing Cryptographic Obfuscation

Abstract approved: ____

Mike Rosulek

Cryptographic obfuscation is a powerful tool that makes programs "unintelligible" yet still runnable. It essentially gives programs the ability to keep secrets. The practical applications of obfuscation range from keeping secrets in banking applications to preventing software theft to providing secure messaging applications. The cryptographic applications of obfuscation are also vast – a tool that hides secrets in programs essentially enables all other cryptographic constructions. Despite (or perhaps due to) its power, obfuscation is currently wildly inefficient and on shaky theoretical ground. Its shaky theoretical ground in particular has resulted in a lack of engineering effort at making it more efficient. In this work, we focus largely on efficiency.

We explore the concrete efficiency of multilinear maps, which are the basis of many cryptographic obfuscation constructions. Multilinear maps are mathematical objects that allow oblivious addition and multiplication of encrypted values. Using multilinear maps, we give the first ever implementations of obfuscation and multi-input functional encryption (MIFE: a variant of obfuscation) for branching programs. Along the way, we create the 5Gen framework for implementations of multilinear map-based applications. We apply the 5Gen framework to experiment with obfuscating point functions and MIFE of order-revealing encryption.

We also explore efficiency in the context of obfuscators and MIFE for circuits. Circuits are more efficient than branching programs for many functions. We give the first MIFE construction

for circuits and prove its security in an ideal model. Our scheme is efficient. To compare, we implement all known circuit obfuscation schemes using the 5Gen framework, and experiment with obfuscating a PRF. This results in the most complex PRF obfuscated to date – with 12 bits of security.

Finally, recently Bishop et al. showed an obfuscation scheme for the specific functionality of wildcard pattern-matching [BKM+18]. This is a simple type of string matching where strings must match a pattern exactly except where there are wildcards. This obfuscation scheme simply relies on the generic group model, with no multilinear maps. Inspired by their work, and the deep connection of functional encryption to obfuscation, we give a function-private, public-key functional encryption scheme for the same wildcard pattern-matching functionality. Our scheme is the first such scheme and we prove its security in a generic model.

©Copyright by Brent Carmer
September 19, 2018
All Rights Reserved

Optimizing Cryptographic Obfuscation

by

Brent Carmer

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented September 19, 2018
Commencement June 2019

Doctor of Philosophy dissertation of Brent Carmer presented on September 19, 2018.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Brent Carmer, Author

CONTRIBUTION OF AUTHORS

Chapter 1 is based on 5Gen: A Framework for Prototyping Applications Using Multilinear Maps and Matrix Branching Programs which was accepted to the 2016 ACM CCS conference and has co-authors Kevin Lewi, Alex J. Malozemoff, Daniel Apon, Adam Foltzer, Daniel Wagner, David W. Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova. Chapter 2 is based on 5Gen-C: Multi-input Functional Encryption and Program Obfuscation for Arithmetic Circuits which was accepted to the 2017 ACM CCS conference and has co-authors Alex J. Malozemoff and Mariana Raykova. Chapter 3 is based on an unpublished manuscript with co-authors James Bartusek, Abhishek Jain, Tancrède Lepoint, Fermi Ma, Tal Malkin, Alex J. Malozemoff, and Mariana Raykova.

TABLE OF CONTENTS

Page

1 Overview 1

1.1 Introduction . . . 1

1.2 Contributions . . . 6

1.2.1 Contributions of Chapter 3: 5Gen . . . 6

1.2.2 Contributions of Chapter 4: 5Gen-C . . . 9

1.2.3 Contributions of Chapter 5: Function-Private Wildcard Encryption . . . 11

2 Preliminaries 16

2.0.1 Program Obfuscation . . . 16

2.0.2 Multi-input Functional Encryption . . . 17

2.0.3 Composite-order Multilinear Maps . . . 19

3 5Gen 21

3.1 Framework Architecture . . . 21

3.2 From Programs to MBPs . . . 22

3.2.1 Matrix Branching Programs . . . 22

3.2.2 MBP Compiler . . . 23

3.3 A Library for Multilinear Maps . . . 28

3.3.1 The GGHLite Multilinear Map . . . 29

3.3.2 The CLT Multilinear Map . . . 31

3.4 Multi-Input Functional Encryption Implementation . . . 33

3.4.1 Optimizing Comparisons . . . 33

3.4.2 Order-Revealing Encryption . . . 35

3.4.3 Three-Input DNF Encryption . . . 37

3.5 Program Obfuscation . . . 40

3.6 Experimental Analysis . . . 42

3.6.1 MIFE Experiments . . . 42

3.6.2 Program Obfuscation Experiments . . . 44

3.7 Conclusions . . . 45

4 5Gen-C 47

4.1 Overview of Existing Techniques . . . 48

4.1.1 Circuit Obfuscation . . . 48

4.1.2 Comparison with Obfuscation from Constant-degree Multilinear Maps . 49

TABLE OF CONTENTS (Continued)

Page

4.2 Circuit Obfuscators . . . 51
4.2.1 Obfuscation using Zim . . . 52
4.2.2 Obfuscation using AB . . . 54
4.2.3 Obfuscation using Lin . . . 55
4.3 Multi-input Functional Encryption from Circuits . . . 56
4.3.1 Construction . . . 59
4.3.2 Proof of Security . . . 61
4.3.3 Optimizations . . . 63
4.3.4 Obfuscation from MIFE . . . 64
4.4 Compiling Circuits for Obfuscation . . . 64
4.4.1 Circuit Optimizations . . . 66
4.4.2 Optimization Results . . . 69
4.5 Implementation . . . 69
4.6 Performance Results . . . 70
4.6.1 AES . . . 71
4.6.2 Goldreich-Goldwasser-Micali PRF . . . 72
4.7 Conclusion . . . 77

5 Public-Key Function-Private Functional Encryption for Wildcard Pattern-Matching 78
5.0.1 Related Work . . . 79
5.1 Preliminaries . . . 80
5.1.1 Security Notions . . . 81
5.1.2 Defining Wildcard Encryption . . . 83
5.2 Wildcard Encryption . . . 88
5.2.1 Construction . . . 88
5.2.2 Security . . . 90

Bibliography 116

Appendix 126
A Circuit Information . . . 127

LIST OF FIGURES

Figure		Page
1.1	*Program obfuscators* (notated $\mathcal{O}$) make the code of a program *unreadable*, but preserve the functionality of the program.	1
1.2	*Multilinear maps* provide the power to (a.) *encode* values as level 1 ciphertexts, (b.) *add* encodings at the same level without decrypting them, (c.) *multiply* encodings at different levels without decrypting them, and to (d.) determine whether the value in an encoding at the zero-testing level is zero.	3
3.1	Framework architecture	22
3.2	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.	32
3.3	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 $d \in [2, 25]$.	37
3.4	Estimates of ciphertext size in ORE for varying domain sizes.	38
3.5	Estimates for the ciphertext size (in GB) for point function obfuscation, for domain sizes $N = 2^{80} = 2^\lambda$ and $N = 2^{40} = 2^\lambda$.	41
4.1	Function to compute the multiplicative degree of an arithmetic circuit consisting of Add, Mul, and input gates, with a single output gate.	49
4.2	The 5Gen framework architecture, with new components introduced in this work in bold and gray boxes.	51
5.1	Our public-key function-private wildcard encryption scheme.	87

LIST OF TABLES

Table | Page

3.1 Interfaces exported by the `libmmap` library. 29

3.2 ORE experiments. 44

3.3 3DNF experiments. 44

3.4 Program obfuscation experiments. 45

4.1 Number of encodings (# Enc) and multilinearity values (κ) 53

4.2 Multilinearity values (κ) for the GGM PRF using AB, Zim, Lin, LZ, and MIO for various input lengths (n) and key lengths (k), in bits. 56

4.3 Comparison of DSL optimizations. 69

4.4 Circuits computing Goldreich's PRG for various choices of predicate. 74

4.5 Multilinearity values for the GGM PRF obfuscated using MIO for both boolean and Σ-vector inputs with $|\Sigma| = 16$. 75

4.6 Obfuscation details for various GGM PRF circuits. 76

LIST OF APPENDIX TABLES

Table | Page

A.1 Circuits and their associated attributes. 128

Dedicated to my father,

James Christopher Carmer.

1948 - 2017

1 Overview

1.1 Introduction

An *obfuscator* is a program that takes another program P as input, and produces another program $\mathcal{O}(P)$ that has the same input/output behavior as P, except that its source code is indecipherable. Obfuscation gives software the power to *keep secrets*, whether simply some secret keys for communication or the entire program itself. If obfuscation truly exists, then we learn something about the nature of the universe, namely that it is *non-reductionist* from a computational perspective: that is, there is some high-level entity (an obfuscated program) that cannot be computationally understood from only parts (its obfuscated code) even though it is fully complete (runnable). See Figure 1.1 for a visual overview of program obfuscation.

There are many *heuristic* obfuscators, which attempt to hide secrets in the source code in various ways. These methods can be as simple as removing formatting, renaming variables, and injecting dead code (like a JavaScript "uglifier") or as complex as whitebox cryptography which uses cryptographically *inspired* techniques [Arx17, Thi17, Jav17]. Unfortunately, even the most clever whitebox constructions have been broken. This is exemplified by the latest whitebox competition held at the CHES 2017 conference, where participants competed in creating and breaking heuristic obfuscators. All submissions were broken (including our own!) [GPRW18]. This leads us to search for a more solid foundation for obfuscation.

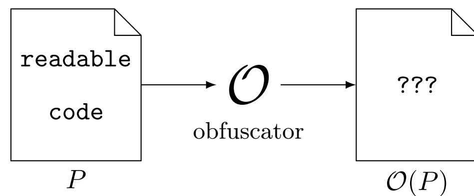

Figure 1.1: *Program obfuscators* (notated $\mathcal{O}$) make the code of a program *unreadable*, but preserve the functionality of the program. In this example, P is a program, then for any input x, the outputs $y_1 = P(x)$ and $y_2 = \mathsf{Obf}(P)(x)$ should be equal. However, the `code` of P is readable, while the code of $\mathcal{O}(P)$ is not.

Cryptographic obfuscation. *Cryptographic* program obfuscators give programs the same power as heuristic obfuscators – the ability to keep secrets – but with security that is defined mathematically. Cryptographic obfuscators come with a proof: an adversary who breaks the security definition has to either solve a hard math problem or spend an exponential amount of time using brute force – trying every possible combination. In the actual practice of obfuscation research so far, the underlying mathematical assumptions of obfuscators are not very far removed from "assume that breaking our scheme is hard." Simplifying and standardizing the assumptions has thus been a major thrust of the community, with some success [LV16, Lin17, LT17].

The area of cryptographic program obfuscation started with a bang in 2001 when Barak et al. showed that for an intuitive definition of obfuscation, there existed un-obfuscatable programs, which implies that obfuscation is impossible in general [BGI+01]. That definition can be summarized as "the adversary should not learn anything more about the program than they would given black-box access to it." This kind of security is known as *virtual black-box* (VBB) security. What we mean by "black-box" is that the program is only seen through its inputs and outputs so nothing at all can be learned from its source code. What Barak et al. showed is a program that *does something with its own code* cannot be obfuscated by a VBB obfuscator, since a VBB obfuscator essentially turns all source code into a black-box. It turns out that this is not such a limitation in practice, since most programs do no such thing, and many VBB and VBB-like obfuscators have appeared in the literature. But it does give pause, and causes us to wonder whether VBB is truly the most natural security definition for obfuscation, and indicates that there will probably never be a VBB obfuscator from natural assumptions.

In the same work, Barak et al. gave an alternative definition of obfuscation called *indistinguishability obfuscation* (IO). What an IO obfuscator guarantees is that it is hard to distinguish the obfuscations of two programs with the same input-output behavior. Intuitively, IO obfuscators scramble the source code of a program in a random way that does not affect the outputs. The difference between IO and VBB is one of scale VBB asks that *nothing at all* be learnable from the source code, but IO simply asks that two obfuscations of programs with the same input

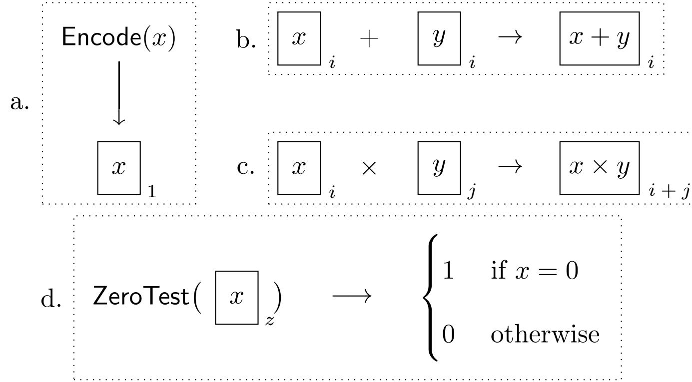

Figure 1.2: *Multilinear maps* provide the power to (a.) *encode* values as level 1 ciphertexts, (b.) *add* encodings at the same level without decrypting them, (c.) *multiply* encodings at different levels without decrypting them, and to (d.) determine whether the value in an encoding at the zero-testing level is zero.

and output are indistinguishable, even if their original source code was different. In addition, there have been no impossibility results for IO. Despite the apparent weakness of IO, there are many surprising applications for it. IO can be used to create public key encryption, signature schemes, non-interactive zero knowledge proofs, oblivious transfer, multiparty non-interactive key exchange, and many other cryptographic primitives [SW14, BZ14]. Its usefulness as a building-block comes from the power it gives to swap two programs with each other.

In parallel with the first obfuscation definitions, Boneh and Silverberg [BS02] first proposed the concept of *multilinear maps* (mmaps). Multilinear maps provide the ability to add and multiply ciphertexts without opening them (see Figure 1.2), much like fully-homomorphic encryption [Gen09, vDGHV10, BGV12], but with the additional power of a public decryption key that tests whether ciphertexts are zero or not. It was already clear that multilinear maps were extremely powerful: Boneh and Silverberg showed many mmap applications including n-party non-interactive key exchange and efficient broadcast encryption [BS02]. However, it was not until 2013 that the obfuscation floodgates opened.

Shortly after Garg, Gentry, and Halevi proposed the first plausible multilinear map [GGH13a], Garg, Gentry, Halevi, Raykova, Sahai, and Waters gave the first candidate obfuscation scheme [GGH+13b]. Neither scheme came with a proof, but subsequent works improved the GGH+13 obfuscator's efficiency and proved its VBB security in an ideal model where the multilinear map is used as a black-box (eg. [BR14, AGIS14]). More multilinear maps appeared [CLT13,

CLT15, GGH15], as well as attacks on both multilinear map candidates and obfuscation schemes [HJ16, CGH+15, MSZ16].

All of the proposed multilinear maps use noise as a way to make encodings hard to open [GGH13a, CLT13, GGH15]. Unfortunately, this noise grows exponentially with every multiplication. If the noise grows too much, it can overflow, making the encoding useless. So, in order to preserve fidelity, the mmap encodings must be large enough to contain all the noise from however many multiplications are needed. This means that obfuscation schemes — all of which use multilinear maps — must be careful to not require multiplications that go too deep. This is a serious limitation: the goal is to obtain obfuscation for all functions, not just those with a small number of multiplications!

The unfortunate fact of noise led to *bootstrapping* methods (eg. [GGH+13b, App14]). The idea of bootstrapping is this: in order to obfuscate a large program, obfuscate a small "core" program and then use other, cheaper, tools to extend to an obfuscation of the whole thing. Here is a bootstrapping metaphor from outside cryptography: consider how orange juice is transported. Water is everywhere, and the orange juice does not depend on having particular water. In addition, it is expensive to transport all that water. It is cheaper to take the water out, then ship the remaining "core" concentrated orange juice. Then, to drink it, add the water back in. Obfuscation bootstrapping works the same way: obfuscate some "core" secret (the concentrated orange juice), and then use that core obfuscation to extend obfuscation to the whole program using cheaper crypto tools (adding water back in).

There have been many bootstrapping theorems in the literature. The most promising are those requiring the fewest multiplications. In fact, a major thrust of research has been to try to push the depth of multiplications down to 1, which would give us obfuscation from solid mathematical assumptions instead of multilinear maps. Until recently, the state-of-the-art required at least depth-2 multiplications, which still require multilinear maps [LT17]. Very recently, the depth of multiplications has been reduced down to 1, but by using non-standard primitives [LM18, Agr18, AJKS18, GJ18]. However, these works have yet to undergo sustained scrutiny. In general,

these low-depth bootstrapping theorems generally build obfuscation out of another primitive – *functional encryption* (FE) [AJ15, BV15].[1]

Functional encryption schemes have special *functional keys* that decrypt ciphertexts to some specific function of their plaintext [BSW11, O'N10]. Except for revealing the output of function, they keep the plaintext secret. For instance, FE can make keys that reveal whether a ciphertext matches a certain pattern. *Multi-input* functional encryption (MIFE) works the same way, except that function keys can contain functions that take more than one input [GGG+14]. An example of MIFE is *order-revealing* encryption, where a function key reveals whether one ciphertext encodes a plaintext that is greater or less-than another. MIFE directly produces obfuscation [GGG+14]. However, FE produces obfuscation through a complex reduction, where the number of arguments is carefully increased one by one, resulting in MIFE [AJ15, BV15].

Another fruitful area is obfuscation for restricted classes of functions, instead of general circuits. Restricting the type of function allows us to create more efficient constructions from more realistic assumptions. For example, there is a line of work on special-purpose obfuscation for pattern-matching functions, with a progression from schemes based on multilinear maps [BR13], to the learning-with-errors assumption [BVWW16, WZ17] to recently simply using generic groups [BKM+18]. This property is true in functional encryption as well. While functional encryption for all circuits is generally hard, requiring IO or multilinear maps, functional encryption schemes for restricted classes of functions are quite well known. Depending on how the functionality is restricted, special-purpose FE can result in protocols such as predicate encryption, attribute-based encryption, or identity-based encryption. In fact, the recent very-low-degree bootstrapping works for general obfuscation rely on special-purpose FE protocols such as FE for constant-degree polynomials [LM18]. We see special-purpose obfuscation and special-purpose FE as one of the most promising directions for this area of research.

We now see the complex geography of the obfuscation landscape: VBB, IO, FE, MIFE and multilinear maps. There are many paths between the landmarks, and many tangled badlands to avoid. The frontier we explore is improving their actual performance.

[1] In fact, as with most cryptography, functional encryption can also be built out of IO [GGH+13b].

1.2 Contributions

In this work, we explore the *practical side* of obfuscation. Previous works have sought to strengthen the theoretical foundation of obfuscation (and FE, and MIFE), but there has been little work on optimizing and building usable constructions with it. In fact, before our work, building a usable obfuscation construction would have been a daunting challenge: without our years of work on multilinear map implementations, circuit formats and compilers, or matrix branching program formats and compilers, an implementation project would have to solve many difficult and subtle problems before even arriving at obfuscation. We emphasize that none of these tools existed before our work. One of our main contributions is therefore the *infrastructure* to conduct experiments in this area. On top of this tool-set, we conduct our research into the practical efficiency of obfuscation and functional encryption. Each of the following chapters delve a little deeper into this world.

1.2.1 Contributions of Chapter 3: 5Gen

Despite the remarkable power of mmaps (introduced in Section 1.2), 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 to carrying out experiments. In Chapter 3 we develop a generic framework called **5Gen**[2] (available at `https://github.com/5GenCrypto`) that lets us experiment with 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 Section 3.3). 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, defined in Section 3.2.1), as

[2]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.

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: GGH-Lite [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, as explained in Section 3.6. 5Gen makes it possible to quickly plug in new mmaps as new proposals emerge, and easily measure their performance in applications like obfuscation and MIFE.

One important application of 2-input MIFE is *order-revealing encryption* (ORE) [GGG+14, BLR+15]. Here the function $f(x, y)$ (associated with the decryption key) 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 Section 3.4.3). 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. 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.

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 functions that we consider in our experiments (see Chapter 4 for a closer examination of the Zimmerman obfuscator).

Summarizing, in Chapter 3, we make the following contributions:

1. 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 Section 3.2 for details.

2. 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 Section 3.3 for more details.

3. 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 and three-input MIFE, as well as performance results that characterize our constructions. See Section 3.4 for details and Section 3.6 for evaluation results.

4. Obfuscation constructions [BGK+14, SZ14, AGIS14, Zim15] using real-world security parameters. We experiment with obfuscating point functions and evaluate their performance. See Section 3.5 for details and Section 3.6 for evaluation results.

This chapter is based on **5Gen: A Framework for Prototyping Applications Using Multilinear Maps and Matrix Branching Programs** with co-authors Kevin Lewi, Alex J. Malozemoff, Daniel Apon, Adam Foltzer, Daniel Wagner, David W. Archer, Dan Boneh, Jonathan

Katz, and Mariana Raykova. It was presented at ACM CCS 2016.

1.2.2 Contributions of Chapter 4: 5Gen-C

The 5Gen framework provides implementations of MIFE and program obfuscation based on *matrix branching programs*, which are a way to represent functions as sequences of matrix multiplications. For simple functions, this approach is sufficient. However, the mapping from general functions to matrix branching programs is exponential, and thus this approach quickly becomes infeasible as complexity grows. Due to this blow-up, in Chapter 3 we were limited to obfuscating an 80-bit point function.

However, matrix branching programs are not the only way to obtain program obfuscation. Both Zimmerman [Zim15] and Applebaum and Brakerski [AB15] showed how to build obfuscators that operate *directly* on the circuit representation of a function. This has several advantages over the branching program approach, not least of which is that one no longer needs to "compile" the function into a branching program. It is worth noting that when we implemented the Zimmerman construction in Chapter 3, we found the branching program approach superior for simple functions such as point functions. This is because the simple structure of these functions is better tailored to the branching program representation. However, for more complex functions, the circuit representation and corresponding circuit obfuscators are more efficient. Thus, extending the 5Gen framework with circuit obfuscators — and also developing MIFE for circuits — substantially enhances the framework, provides a basis for comparison, and pushes cryptographic program obfuscation further towards practicality.

In Chapter 4, we explore in detail MIFE and program obfuscation for circuits. We implement and evaluate existing constructions as well as develop new schemes in the context of PRF obfuscation. Our implementation extends the 5Gen framework, which previously included only MIFE and program obfuscation for branching programs. This enhances the functionality of 5Gen and makes our constructions available for use and experimentation.

Towards this goal, we introduce a new MIFE construction that works directly on arithmetic circuits. Using Goldwasser et al.'s transformation from MIFE to IO [GGG+14], this gives us an obfuscation scheme which avoids the extra n multilinearity overhead inherent in prior circuit obfuscators [Zim15, AB15], where n is the number of inputs. To fully compare our new approach with existing constructions, we implement *all* known circuit obfuscation approaches, including the optimization of Applebaum and Brakerski's scheme [AB15] for low-depth PRFs by Lin [Lin16] and our own adaptation of this optimization to Zimmerman's scheme [Zim15].

Since our constructions work over arithmetic circuits, we develop a compiler for constructing circuits from high-level program descriptions. This compiler includes optimizations to reduce the multiplicative degree of the resulting circuits — a metric not targeted by any existing circuit compilers — which has a *huge* impact on which functions are obfuscatable using existing mmaps.

Finally, given our implementation and compiler suite, we experiment with obfuscating a PRF. Obfuscating a PRF would allow us to *bootstrap* our obfuscator to general functions. In particular, we look at obfuscating both AES and the Goldreich-Goldwasser-Micali (GGM) PRF. While we are still far from obfuscating the complete AES algorithm, we are able to demonstrate some surprising results, such as the obfuscation of the GGM PRF with a 64-bit key and 12 input/output bits with 80 bits of security for the underlying mmap.

Summarizing, in Chapter 4, we make the following contributions:

1. An instantiation of multi-input functional encryption (MIFE) for circuits (cf. Section 4.3). Prior MIFE constructions [BLR+15, LMA+16] required that the function be compiled as a branching program, which becomes infeasible for complex functions, whereas our approach works directly on the arithmetic circuit representation of a function. Additionally, using the Goldwasser et al. [GGG+14] transformation from MIFE to obfuscation, this gives us a new obfuscator which performs better than all existing obfuscators.

2. A new circuit compiler which provides optimizations for generating low degree arithmetic circuits starting from a high-level specification of the functionality (cf. Section 4.4). This tool is of independent interest, as it produces function representations which can be used

in the context of multi-party secure computation or fully homomorphic encryption.

3. Implementations of our new MIFE construction, an obfuscator based on our MIFE construction, and all existing circuit-based obfuscators from the literature (cf. Section 4.5). As part of this, we introduce three new components to the 5Gen framework introduced in Chapter 3 (1) `libacirc`, a language and library for building and computing over arithmetic circuits; (2) `mio`, an implementation of circuit-based multi-input functional encryption and program obfuscation; and (3) `cxs`, a toolkit for compiling and (x) synthesizing arithmetic circuits optimized for minimizing circuit degree. See Figure 4.2 for the enhanced 5Gen architecture.

4. A thorough exploration of the performance of the various obfuscators, with a focus on obfuscating a PRF (cf. Section 4.6).

This chapter is based on **5Gen-C: Multi-input Functional Encryption and Program Obfuscation for Arithmetic Circuits** with co-authors Alex J. Malozemoff and Mariana Raykova. It was presented at ACM CCS 2017.

1.2.3 Contributions of Chapter 5: Function-Private Wildcard Encryption

As we state above, functional encryption is a powerful type of encryption where decryption reveals a function of the plaintext [BSW11, O'N10]. Functional encryption evolved as an abstraction of predicate encryption (introduced in [KSW08]) where secret keys are associated with predicates. Secret keys in predicate encryption schemes will only decrypt messages whose attribute matches the predicate. More formally, if $x \in \{0,1\}^m$ is an attribute and $y \in \{0,1\}^*$ a message, then

predicate encryption schemes are functional encryption schemes for the function

$$f_p(x, y) = \begin{cases} y & \text{if } p(x) = 1 \\ \bot & \text{otherwise} \end{cases}$$

where $p : \{0,1\}^m \to \{0,1\}$ is a predicate. The question naturally arises whether it is also possible to *hide the predicate*. Brakerski and Segev answer in the affirmative for the secret key setting [BS15], but the public-key setting offers unique challenges.

The notion of function-privacy in the public-key setting for predicate encryption first appeared for identity-based encryption (IBE) [BRS13a]. IBE is a predicate encryption scheme where the predicate determines whether the identity (encoded in the ciphertext) is a member of a set of authorized identities (encoded in the secret key). In this context function privacy means the holder of a secret key should not be able to learn any information about the set of identities in the key, except for the fact that decryption succeeded or failed on particular ciphertexts. The actual definition of function privacy is subtle. The adversary should not be able to distinguish two secret keys for different sets of identities. However, since the scheme has a public key, the adversary can create ciphertexts specifically designed to distinguish one secret key from another. Then, if the adversary has some *a priori* knowledge about the set of identities, it can exhaustively search in order to learn the identities that are associated with a particular key.

In order to avoid brute-force attacks, the sets of identities themselves must be drawn from a distribution with a certain amount of entropy. This means that the adversary should not be able determine which set it has a secret key for, even when it gets a polynomial number of guesses using the public key. Concretely, this means the probability that a particular set is drawn should be at least negligible in the security parameter, which results in a generic lower bound of *min-entropy* of $\omega(\log \lambda)$ on the function distribution for all function-private schemes. The min-entropy requirement first given by Boneh, Raghunathan, and Segev for IBE was higher – at least linear in the security parameter λ [BRS13a, BRS13b] – while subsequent works improved it to the minimum $\omega(\log \lambda)$ for various types of special-purpose FE [AAB+15, ITZ16, PM18].

There are a number of public-key function-private functional encryption schemes in the literature. Agrawal et al. gave new simulation-based security definitions (circumventing impossibility results) as well as constructions for function-private inner-product functional encryption [AAB+13]. Iovino et al. show general function-private constructions from a weakened version of iO. Special-purpose function-private protocols for functionalities such as subspace membership encryption [BRS13b, PM18], inner-product encryption [BJK15, AAB+15, KKS17], or hidden-vector encryption [BM18] exist as well.

Special purpose obfuscation constructions also have similar min-entropy constraints. In the obfuscation context, one party wishes to hide a function while still allowing it to be used by another party. This means that the evaluator should not be able to guess the functionality, even though they can query it on arbitrary inputs of their choosing, just as in the public-key function-hiding functional encryption setting.

Bishop, Kowalczyk, Malkin, Pastro, Raykova, Shi very recently showed a special purpose obfuscation scheme for the class of wildcard pattern-matching functions [BKM+18]. A wildcard pattern matches an input string if each bit of the string either matches the pattern exactly or the value at the ith position of the pattern is a wildcard (which we notate as "?"). In the rest of this work, we refer to the matching function as $\mathsf{Match} : \{0, 1, ?\}^n \times \{0, 1\}^n \to \{0, 1\}$. In the BKMPRS obfuscator, both the bit-pattern and the location of the wildcards are hidden to the evaluator, who only learns whether their input matched or not. While it is possible to define distributions of wildcard patterns, Bishop et al. showed distributional virtual black-box security for a single distribution: the distribution of wildcard patterns with at most $3/4n$ wildcards where n is the length of the input. Recently, Bartusek and Ma extended the analysis to support up to $n - \omega(\log n)$ wildcards and gave an improved security bound [BM18].

The functionality of obfuscation is very close to function-private public-key functional encryption – it is essentially obfuscation with secret inputs – and so the following question arises:

Is there an efficient function-private public-key
functional encryption scheme for wildcards?

In Chapter 5, we answer in the affirmative in the generic bilinear map model. In Section 5.2, we

give the first function-private public-key functional encryption scheme for the family of functions

$$f_{\mathsf{pat}}(x, y) = \begin{cases} y & \text{if } \mathsf{Match}(\mathsf{pat}, x) \\ \perp & \text{otherwise} \end{cases}$$

where pat is sampled from the distribution of wildcard patterns with at most $n - \omega(\log \lambda)$ wildcards. Our scheme is strongly inspired by BKMPRS wildcard obfuscation [BKM+18]. In fact, our proof of function privacy directly relies on it. Our scheme is efficient – the public key contains $4n + 1$ encodings, the secret key and ciphertext each contain $2n$ encodings, and our decryption routine consists of $2n$ pairings.

Following Boneh, Raghunathan, and Segev, we separate the orthogonal concerns of *message* privacy and *function* privacy [BRS13a, BRS13b]. Message privacy holds if the adversary cannot distinguish two messages, even if it can request secret keys that do not trivially distinguish them. In Section 5.2.2.2 we show the message privacy of our scheme using an analysis in the generic bilinear map model.

Function privacy is trickier due to the min-entropy requirement discussed above. In general, function privacy requires that it is difficult for the adversary to distinguish a secret key for a function sampled from a chosen distribution from a secret key sampled uniformly. Since our setting is for a single distribution, namely the distribution of wildcard patterns containing at most $n - \omega(\log \lambda)$ wildcards, function privacy requires that the adversary cannot distinguish a *valid* secret key from an *invalid* one. In addition, in our scheme, an adversary is also allowed to obtain ciphertexts whose (random) attribute matches a particular secret key (following "enhanced" function privacy in [BRS13a]). This reflects real-world applications where a user may be given a secret key as well as ciphertexts decryptable with that secret key. Our proof of function privacy in Section 5.2.2.3 uses an analysis in the generic bilinear map model to show these encryptions do not reveal anything about the secret keys. We then reduce to BKMPRS to show the indistinguishability of real keys with simulated keys.

Summarizing, our contribution in Chapter 5 is the first function-hiding, public-key functional

encryption scheme for the wildcard matching functionality. This chapter is based on ongoing research with collaborators James Bartusek, Abhishek Jain, Tancrède Lepoint, Fermi Ma, Tal Malkin, Alex Malozemoff, and Mariana Raykova.

2 Preliminaries

Notation. For an integer $n > 0$, we use $[n]$ to denote the set of integers $\{1, \ldots, n\}$. 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. We let κ denote the multilinearity (maximum supported depth of multiplications) of the multilinear map used in our constructions.

2.0.1 Program Obfuscation

Intuitively, *program obfuscation* allows one party to obfuscate a program in such a way that any other party cannot learn anything about the internal workings of the program besides what can be deduced from input/output relationships.

Definition 2.0.1. *A* program obfuscator *is a tuple of algorithms* $(\mathsf{obf}, \mathsf{eval})$ *defined as follows:*

- $\mathsf{obf}(\lambda, C) \to \mathsf{Obf}$*: Takes as input the security parameter* λ *and an arithmetic circuit* $C : \{0,1\}^n \to \{0,1\}^m$*, and returns an obfuscation* Obf*.*
- $\mathsf{eval}(\mathsf{Obf}, \mathbf{x}) \to \mathbf{z}$*: On input obfuscation* Obf *and bitstring* $\mathbf{x} \in \{0,1\}^n$*, output bitstring* $\mathbf{z} \in \{0,1\}^m$*.*

Definition 2.0.2 (Correctness)**.** *A program obfuscator* $(\mathsf{obf}, \mathsf{eval})$ *is* correct *for circuit* C *if for all* $\mathbf{x} \in \{0,1\}^n$*, it holds that*

$$\mathsf{eval}(\mathsf{obf}(\lambda, C), \mathbf{x}) = C(\mathbf{x}).$$

Definition 2.0.3 (Efficiency)**.** *A program obfuscator* $(\mathsf{obf}, \mathsf{eval})$ *is* efficient *for circuit* C *if there exists polynomial* $p(\cdot)$ *such that* $|\mathsf{obf}(\lambda, C)| < p(\lambda)$*.*

Regarding security, the two key notions are *virtual black-box (VBB) obfuscation* and *indistinguishability obfuscation* (IO).

Definition 2.0.4. *A program obfuscator* $(\mathsf{obf}, \mathsf{eval})$ *is an* indistinguishability obfuscator *for circuit class* $\{\mathcal{C}_\lambda\}$ *if for all* $\lambda \in \mathbb{N}$ *and for every pair of circuits* $C_0, C_1 \in \mathcal{C}_\lambda$ *such that* $|C_0| = |C_1|$ *and* $C_0(\mathbf{x}) = C_1(\mathbf{x})$ *for all inputs* $\mathbf{x}$*, then*

$$\{C_0, C_1, \mathsf{obf}(\lambda, C_0)\} \approx \{C_0, C_1, \mathsf{obf}(\lambda, C_1)\}.$$

Definition 2.0.5. *A program obfuscator* $(\mathsf{obf}, \mathsf{eval})$ *is a* virtual black-box obfuscator *for circuit class* $\{\mathcal{C}_\lambda\}$ *if for every efficient adversary* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that for every* $\lambda \in \mathbb{N}$ *and* $C \in \mathcal{C}_\lambda$ *it holds that*

$$\Pr[\mathcal{A}(\mathsf{obf}(\lambda, C)) = 1] - \Pr[\mathcal{S}^C(\lambda) = 1] \text{ is negligible },$$

where $\mathcal{S}^C$ *denotes that the simulator has black-box access to circuit* C*.*

2.0.2 Multi-input Functional Encryption

Multi-input functional encryption (MIFE) [GGG+14] provides a way to compute a function over multiple ciphertexts such that the "decryptor" only learns that function of the ciphertexts and nothing else. In this work we utilize the secret-key variant of MIFE introduced by Boneh et al. [BLR+15].

Definition 2.0.6. *A* single-key secret-key multi-input functional encryption (1SK-MIFE) scheme *is a tuple of algorithms* $(\mathsf{1SK\text{-}MIFE.Setup}, \mathsf{1SK\text{-}MIFE.Enc}, \mathsf{1SK\text{-}MIFE.Dec})$ *defined as follows:*

- $\mathsf{1SK\text{-}MIFE.Setup}(n, C) \to (\mathsf{sk}, \mathsf{ek})$*: Takes as input the security parameter* λ *and an arithmetic circuit* $C : \{0,1\}^{d_1} \times \cdots \times \{0,1\}^{d_n} \to \{0,1\}^m$ *and returns a secret key* sk *and an evaluation key* ek.

- $\mathsf{1SK\text{-}MIFE.Enc}(\mathsf{sk}, i, x) \to \mathsf{ct}$: *Takes as input secret key* sk*, index* i*, and input* $x \in \{0,1\}^{d_i}$*, and outputs a ciphertext* $\mathsf{ct}^{(i)}$.

- $\mathsf{1SK\text{-}MIFE.Dec}(\mathsf{ek}, \mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(n)}) \to \mathbf{y}$: *Takes as input an evaluation key* ek *and ciphertexts* $\mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(n)}$*, and outputs a bitstring* $\mathbf{y}$.

Definition 2.0.7 (1SK-MIFE Correctness)**.** *A* 1SK-MIFE *scheme* Π *is* correct *if for any multi-input circuit* C*, and any inputs* $x^{(1)} \in \{0,1\}^{d_1}, \ldots, x^{(n)} \in \{0,1\}^{d_n}$*, if* $(\mathsf{ek}, \ \mathsf{sk}) \leftarrow \mathsf{1SK\text{-}MIFE.Setup}(1^n, C)$ *and for each* $i \in [n]$*,* $\mathsf{ct}^{(i)} \leftarrow \mathsf{1SK\text{-}MIFE.Enc}(\mathsf{sk}, i, x^{(i)})$*, then,*

$$\mathsf{1SK\text{-}MIFE.Dec}(\mathsf{ek}, \ \mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(m)}) = C(x^{(1)}, \ldots, x^{(m)}).$$

Towards defining security, we introduce the following security game. Given a circuit C, adversary $\mathcal{A}$, and bit $b \in \{0,1\}$, we define the experiment $\mathsf{Expt}^{\mathsf{1SK\text{-}MIFE}}_{C,Q,b}(\mathcal{A})$, parameterized over a number of queries Q:

<u>Experiment $\mathsf{Expt}^{\mathsf{1SK\text{-}MIFE}}_{C,Q,b}(\mathcal{A})$:</u>

1. Compute $(\mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{1SK\text{-}MIFE.Setup}(1^n, C)$ and send ek to $\mathcal{A}$.
2. For $q \in [Q]$, $\mathcal{A}$ sends $(i_q, x_{q,0}, x_{q,1})$ and is given ciphertext $\mathsf{ct}_q \leftarrow \mathsf{1SK\text{-}MIFE.Enc}(\mathsf{sk}, i_q, x_{q,b})$.
3. $\mathcal{A}$ outputs bit $b' \in \{0,1\}$.

We define security for a 1SK-MIFE scheme in the composite-order multilinear map generic model. For this we use the notion of *admissible execution traces* [BLR+15]. An *execution trace* includes the sequence of queries from $\mathcal{A}$ to both its oracle and the mmap. An execution trace is *admissible* if for all tuples of queries $((1, \, x_{q_1,0}, \, x_{q_1,1}), \cdots, (n, \, x_{q_n,0}, \, x_{q_n,1}))$ we have that $C(x_{q_1,0}, \ldots, x_{q_n,0}) = C(x_{q_1,1}, \ldots, x_{q_n,1})$.

Definition 2.0.8 (IND-security for 1SK-MIFE)**.** *A* 1SK-MIFE *scheme is* Q-IND-secure *if, for all circuits* C *and all efficient adversaries* $\mathcal{A}$*, the quantity*

$$\mathsf{Adv}^{\mathsf{1SK\text{-}MIFE}}_{C,Q}(\mathcal{A}) := |W_0 - W_1|$$

is negligible, where

$$W_b = \Pr\left[\begin{array}{c}\mathsf{Expt}^{\mathsf{1SK\text{-}MIFE}}_{C,Q,b}(\mathcal{A}) \text{ outputs } 1 \text{ and yields} \\ \text{an admissible execution trace}\end{array}\right].$$

2.0.3 Composite-order Multilinear Maps

Multilinear maps provide a way to add and multiply secret encoded values up to a certain point, at which a given encoding can be "zero-tested" to determine whether its secret value is zero or non-zero. In particular, mmaps provide an Encode operation that maps a value into its encoded form, and Add and Mul operations that allow adding and multiplying encoded values. At a certain point, the ZeroTest operation can be run to test equality with zero. *Composite-order* mmaps allow using multiple *slots* of encoded values, where now ZeroTest outputs zero if and only if *all* values in *all* slots are zero.

Many obfuscation constructions are proven secure in an mmap generic model, which provides oracle access to the various mmap operations, returning "handles" to encoded values rather than the encoded values themselves. The composite-order multilinear generic model is a slight strengthening of this to allow encoding values across all of the mmap slots. We define this formally below.

Definition 2.0.9. *The* composite-order multilinear generic model *[Zim15] is defined by the operations* Setup*,* Encode*,* Add*,* Mul*, and* ZeroTest*, defined as follows.*

- $\mathsf{Setup}(\mathcal{U}, \lambda, N) \to (\mathsf{pp}, \mathsf{sp}, p_1, \ldots, p_N)$*: Takes as input a top-level index set* $\mathcal{U}$*, security parameter* λ*, and the number of slots* N*, and produces public parameter* pp*, secret parameter* sp*, and*

primes $p_1, \ldots, p_N$.

- $\mathsf{Encode}(\mathsf{sp}, x_1, \ldots, x_N, \mathcal{S}) \to h$: *Takes as input secret parameter* sp, *scalars* $x_1, \ldots, x_N$, *and index set* $\mathcal{S} \subseteq \mathcal{U}$, *and returns an "encoding" which is a fresh "handle"* $h \xleftarrow{\$} \{0,1\}^\lambda$. *An entry* $h \mapsto (x_1, \ldots, x_N, \mathcal{S})$ *is added to the (internal) table* T, *and* h *is returned.*

- $\mathsf{Add}(\mathsf{pp}, h_1, h_2) \to \{h, \perp\}$: *Takes as input public parameter* pp *and handles* h_1 *and* h_2. *If* $h_1 \mapsto (x_{1,1}, \ldots, x_{1,N}, \mathcal{S}_1)$ *and* $h_2 \mapsto (x_{2,1}, \ldots, x_{2,N}, \mathcal{S}_2)$ *are in* T, *and* $\mathcal{S}_1 = \mathcal{S}_2 \subseteq \mathcal{U}$, *then compute a fresh "handle"* h *and add* $h \mapsto (x_{1,1} + x_{2,1} \pmod{p_1}, \ldots, x_{1,N} + x_{2,N} \pmod{p_N}, \mathcal{S}_1)$ *to* T, *returning* h; *otherwise, return* $\perp$.

- $\mathsf{Mul}(\mathsf{pp}, h_1, h_2) \to \{h, \perp\}$: *Takes as input public parameter* pp *and handles* h_1 *and* h_2. *If* $h_1 \mapsto (x_{1,1}, \ldots, x_{1,N}, \mathcal{S}_1)$ *and* $h_2 \mapsto (x_{2,1}, \ldots, x_{2,N}, \mathcal{S}_2)$ *are in* T, *and* $\mathcal{S}_1 \cup \mathcal{S}_2 \subseteq \mathcal{U}$, *then compute a fresh "handle"* h *and add* $h \mapsto (x_{1,1} \cdot x_{2,1} \pmod{p_1}, \ldots, x_{1,N} \cdot x_{2,N} \pmod{p_N}, \mathcal{S}_1 \cup \mathcal{S}_2)$ *to* T, *returning* h; *otherwise, return* $\perp$.

- $\mathsf{ZeroTest}(\mathsf{pp}, n) \to \{\text{"zero"}, \text{"non-zero"}, \perp\}$: *Takes as input public parameter* pp *and handle* h. *If* $h \mapsto (x_1, \ldots, x_N, \mathcal{S})$ *is in* T *and* $\mathcal{S} = \mathcal{U}$, *then return "zero" if* $x_1 \equiv 0 \pmod{p_1}, \ldots, x_N \equiv 0 \pmod{p_N}$, *else "non-zero"; otherwise return* $\perp$.

3 5Gen: A Framework for Prototyping Applications Using Multilinear Maps and Matrix Branching Programs

Secure multilinear maps 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 chapter 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.

3.1 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 Section 3.2.

The bottom layer of our framework, `libmmap`, provides an API for using various mmaps,

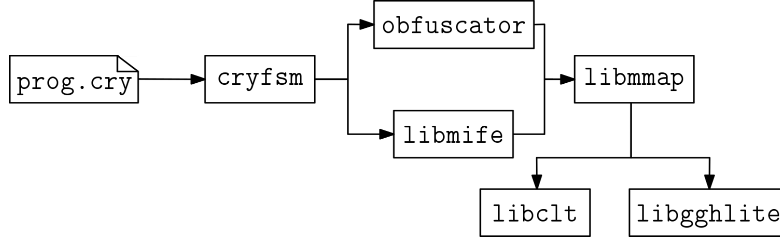

Figure 3.1: Framework architecture. We use `cryfsm` to compile a Cryptol program (here denoted by `prog.cry`) to an MBP, which can either be used as input into our MIFE implementation or our obfuscation implementation. Both these implementations use `libmmap` as a building block, which supports both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

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 Section 3.3, 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. Section 3.4) 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.

3.2 From Programs to MBPs

3.2.1 Matrix Branching Programs

Formally, 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 $j \in \{0, \ldots, d-1\}$, 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 $j \in \{0, \ldots, d-1\}$, 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 $x \in \{0, \ldots, d-1\}^\ell$ 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.

3.2.2 MBP Compiler

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 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. 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 bit-string 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 this work, we consider functions on inputs of a fixed length. Hence, for a positive integer n, we can take advantage *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.

Optimizations for MBP creation. Boneh et al. [BLR+15] describe a simple five-state finite state machine for ORE, 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:

$$x_1 y_1 y_2 x_2 x_3 y_3 \cdots y_n x_n, \tag{3.1}$$

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 Section 3.4.1, 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.

3.3 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[1], 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 3.1 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

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

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

Table 3.1: Interfaces exported by the `libmmap` library.

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 in `libmmap`: `libgghlite` (Section 3.3.1) and `libclt` (Section 3.3.2).

3.3.1 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 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[2] released by Albrecht et al. [ACLL15]. We modified this implementation to add functionality

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

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 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 [MSZ16] 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.

3.3.2 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[3] 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

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

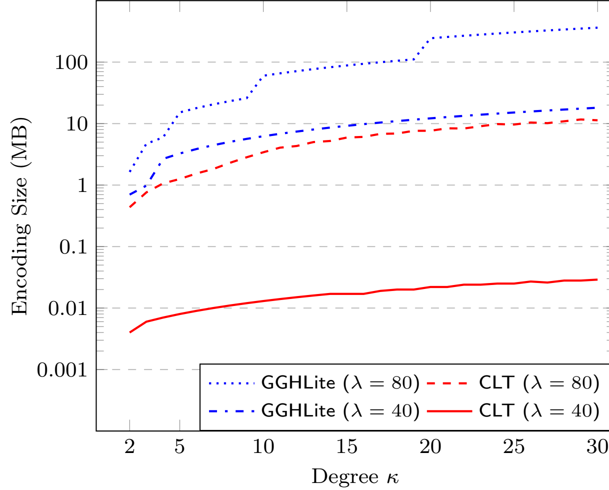

Figure 3.2: Estimates for the size of a single encoding in megabytes (MB) produced for security parameters $\lambda = 80$ and $\lambda = 40$ and varying the multilinearity degree $\kappa \in [2, 30]$ for the GGHLite and CLT mmaps.

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 [BWZ14, 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.

Size estimates. Figure 3.2 presents estimates for the size of an encoding using GGHLite and CLT for security parameters $\lambda = 80$ and $\lambda = 40$. 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.

3.4 Multi-Input Functional Encryption Implementation

MIFE implementation. We implemented the Boneh et al. [BLR+15] MIFE construction using our `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.

3.4.1 Optimizing Comparisons

In this section, we describe a case study of applying the optimizations detailed in Section 3.2.2 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 (3.1). 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

$$\begin{aligned} M &= d + d(d+2) + 3(d+2) + (\kappa - 3)(d+2)^2 \\ &= d^2(\kappa - 2) + (d+1)(4\kappa - 6). \end{aligned} \tag{3.2}$$

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

$$\begin{aligned} M &= d + 3d + 3(\kappa - 2)(d+2) \\ &= 3(\kappa - 2)(d+2) + 4d. \end{aligned} \tag{3.3}$$

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.

3.4.2 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 $\textsc{cmp}$ 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 $k \xleftarrow{\$} \{0,1\}^\lambda$ 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 $r \xleftarrow{\$} \{0,1\}^\lambda$, 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 $N + 2\lambda + \lceil \log_2(N) \rceil$ bits long. For $N = 10^{12}$ and $\lambda = 80$, this amounts to ciphertexts of length 116.42 GB.[4]

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 (3.2) and (3.3), 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 3.3 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.

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.

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

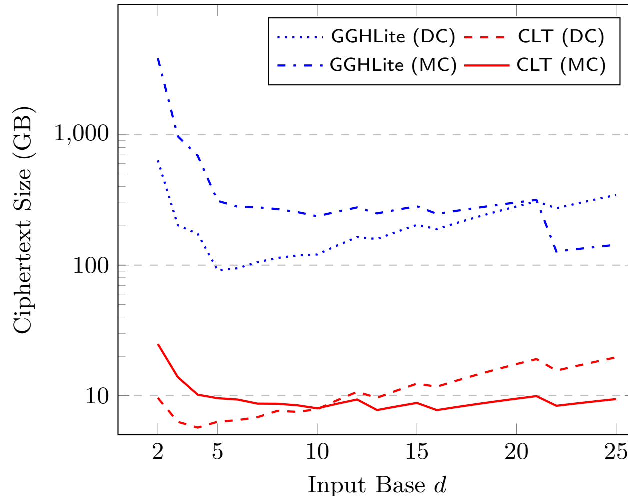

Figure 3.3: Estimates of the ciphertext size (in GB) for ORE with best-possible semantic security at $\lambda = 80$, for domain size $N = 10^{12}$ and for bases $d \in [2, 25]$. We compare GGHLite and CLT, with the DC-variant and MC-variant optimizations.

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 3.4. 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 3.3. 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).

3.4.3 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

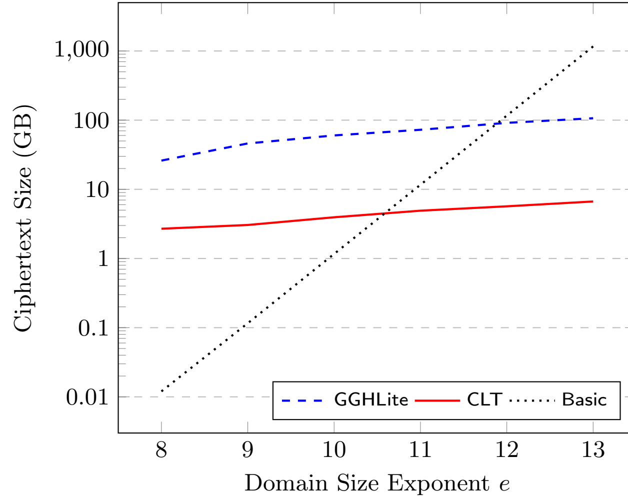

Figure 3.4: Estimates of the ciphertext size (in GB) for ORE with best-possible semantic security at $\lambda = 80$, for varying domain sizes. The exponent e on the x-axis denotes support for plaintexts in the range from 1 to $N = 10^e$. We compare GGHLite map (DC-variant), the CLT map, and the basic construction Π_{ore} (described in Section 3.4.2).

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 is similar 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 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 (3.1) 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.

3.5 Program Obfuscation

We construct a point function obfuscator for the Sahai-Zhandry obfuscator [SZ14]. Our obfuscator operates identically to the Sahai-Zhandry obfuscator, which is VBB secure (cf. Section 2.0.1), except that we discard half of the ciphertext that corresponds to the second input in the "dual-input" branching programs that obfuscator uses. We emphasize that as any attack on our obfuscator immediately results in an attack on the Sahai-Zhandry obfuscator. See the CCS 2016 version of this chapter for more details [LMA+16].

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.

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

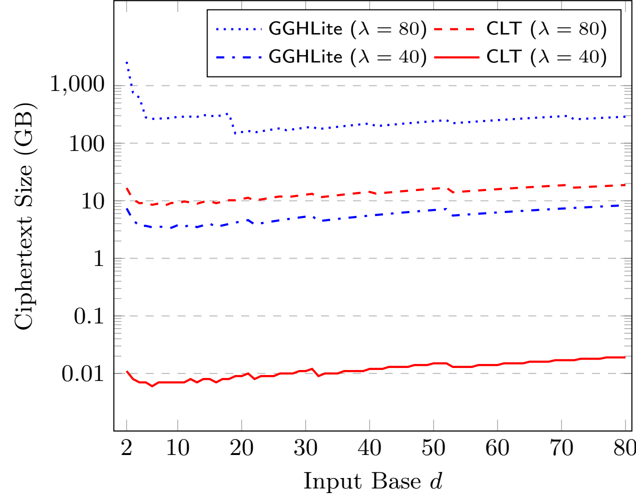

Figure 3.5: Estimates for the ciphertext size (in GB) for point function obfuscation, for domain sizes $N = 2^{80} = 2^{\lambda}$ and $N = 2^{40} = 2^{\lambda}$. In the case of $\lambda = 80$, the minimums are achieved at $d = 19$ for GGHLite and $d = 8$ for CLT. In the case of $\lambda = 40$, the minimums are achieved at $d = 9$ for GGHLite and $d = 6$ for CLT.

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 3.5, 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.

3.6 Experimental Analysis

All of our experiments in this chapter 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.

3.6.1 MIFE Experiments

We evaluated our multi-input functional encryption constructions with two applications: order-revealing encryption (ORE) (cf. Section 3.4.2) and three-input DNF (3DNF) encryption (cf. Section 3.4.3). In Section 3.4, 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 Table 3.2 and Table 3.3. 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 keygen operation 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.

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 3.2: 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 3.3: 3DNF experiments. See Table 3.2 for the column details.

3.6.2 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 the goal of minimizing the total obfuscation size for each obfuscated point function (see Section 3.5 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 3.4. 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

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 3.4: Program obfuscation experiments. See Table 3.2 for the column details. Note that "|obf|" denotes the obfuscation size.

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.

3.7 Conclusions

In this chapter, 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 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.

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.

4 5Gen-C: Multi-input Functional Encryption and Program Obfuscation for Arithmetic Circuits

Program obfuscation is a powerful security primitive with many applications. White-box cryptography studies a particular subset of program obfuscation targeting keyed pseudorandom functions (PRFs), a core component of systems such as mobile payment and digital rights management. Although the white-box obfuscators currently used in practice do not come with security proofs and are thus routinely broken, recent years have seen an explosion of *cryptographic* techniques for obfuscation, with the goal of avoiding this build-and-break cycle.

In this chapter, we explore cryptographic program obfuscation and the related primitive of multi-input functional encryption (MIFE). In particular, we extend the 5Gen framework introduced in Chapter 3 to support circuit-based MIFE and program obfuscation, implementing both existing and new constructions. We then evaluate and compare the efficiency of these constructions in the context of PRF obfuscation.

As part of this chapter we (1) introduce a novel instantiation of MIFE that works directly on functions represented as arithmetic circuits, (2) use a known transformation from MIFE to obfuscation to give us an obfuscator that performs better than all prior constructions, and (3) develop a compiler for generating circuits optimized for our schemes. Finally, we provide detailed experiments, demonstrating, among other things, the ability to obfuscate a PRF with a 64-bit key and 12 bits of input (containing 62k gates) in under 4 hours, with evaluation taking around 1 hour. This is by far the most complex function obfuscated to date.

4.1 Overview of Existing Techniques

In this section, we briefly describe the high-level idea behind circuit obfuscation (Section 4.1.1) and compare it with the approach of obfuscation from constant-degree mmaps (Section 4.1.2).

4.1.1 Circuit Obfuscation

The main idea behind circuit obfuscation is to run the circuit itself directly on inputs encoded by the mmap. While this does not hide the circuit itself (as the circuit is needed by the evaluator to know which encodings to add and multiply), this can be solved by having the circuit itself be a universal circuit, taking the particular function to hide as input. However, in this work we are interested in obfuscating PRFs, where the PRF functionality is public but the (embedded) key should be hidden from the evaluator, and thus making the PRF functionality public is not an issue.

In order to prevent the evaluator from computing something *other* than the specified circuit, these obfuscators encode a "check computation" in the encoded inputs, so that a valid result will be computed if and only if the evaluator correctly computes the circuit. This is done using composite-order mmaps (defined in Section 2.0.3). These mmaps have multiple "slots", where computation occurs in parallel across all the slots. The idea is then to use one of the slots to embed the check computation so that it only cancels out if the circuit was correctly computed. Otherwise, a random value will appear in the resulting top-level encoding, and thus the $\mathsf{ZeroTest}$ operation will return "non-zero". Another slot encodes the main computation, which only returns a valid result if the check computation succeeds.

In circuit obfuscators, every encoded element is indexed by a (multi-)set of special symbols, called the *index set*. Only encodings with the same index set can be added together, resulting in a new encoding at that same index set. Any encodings can be multiplied together, resulting in a new encoding at the product of the index sets. For instance, suppose we have encodings $[x]_{AB}$ at index set $\{A, B\}$ and $[y]_{AB}$ also at index set $\{A, B\}$. These encodings can be both added

$\mathsf{deg}(g : Gate)$

if $g = \mathsf{Add}(x, y)$
  return $\max(\mathsf{deg}(x), \mathsf{deg}(y))$
if $g = \mathsf{Mul}(x, y)$
  return $\mathsf{deg}(x) + \mathsf{deg}(y)$
else
  return 1

Figure 4.1: Function to compute the multiplicative degree of an arithmetic circuit consisting of Add, Mul, and input gates, with a single output gate. The degree of a circuit with multiple outputs is the *maximum* of the degrees of its outputs considered individually.

and multiplied. If we add the encodings, the result $[x + y]_{AB}$ has the same index set. If we multiply the encodings, however, the result $[xy]_{AABB}$ has index set $\{A, A, B, B\}$. The idea then is that as we compute the arithmetic circuit, we add and multiply encodings, increasing the size of the index set. Eventually, we reach an output wire, with the resulting index set viewed as the "top-level" index set (i.e., the index set at which we can successfully zero-test).

As mentioned above, as we evaluate the circuit we multiply and add encodings. Each time we multiply encodings, we increase the noise level of the encoding. Thus, we must generate encodings to support enough noise such that they still retain fidelity upon reaching the top-level. Put another way, in the underlying mmap we must know what the maximum multiplicative degree (defined by Figure 4.1) will be in order to generate encodings with the appropriate noise tolerance.

4.1.2 Comparison with Obfuscation from Constant-degree Multilinear Maps

Starting with the work of Lin [Lin16], a recent and concurrent line of work has looked at building obfuscation from *constant-degree* mmaps [LV16, AS17, Lin17, LT17]. All of these approaches uti-

lize the "bootstrapping" approach for building obfuscation from (succinct) functional encryption (FE) [BV15, AJ15]; namely, the authors design the FE scheme using constant-degree mmaps, and then use that scheme as the underlying FE scheme in the bootstrapping procedure. Thus, the efficiency bottleneck of these schemes becomes the bootstrapping procedure: even if the necessary FE construction can be built using very efficient tools, if the bootstrapping is prohibitively expensive then this approach will not outperform the circuit approach for many functions.

We focus here on the bootstrapping approach of Bitansky and Vaikuntanathan [BV15], who build obfuscation from succinct *public-key* FE (the approaches of Ananth and Jain [AJ15] and Lin et al. [LPST16] are similar). The (simplified) idea is that, given circuit $C(x) : \{0,1\}^n \to \{0,1\}^m$, the $\mathsf{Obfuscate}$ procedure constructs a functional key FSK_{C^*} for circuit C^* defined as

$$C^*(\mathsf{SK}, x) := \mathsf{Sym.Dec}(\mathsf{SK}, \mathsf{CT})(x),$$

where SK is a symmetric key, $\mathsf{CT} = \mathsf{Sym.Enc}(\mathsf{SK}, C)$, and $(\mathsf{Sym.Enc}, \mathsf{Sym.Dec})$ defines a symmetric key scheme. Namely, C^* decrypts a ciphertext, interprets the decrypted object as a circuit, and evaluates it on the input x.

Next, the procedure computes $(\mathsf{PK}_{C^*}, \mathsf{FSK}_{C^*}) \leftarrow \mathsf{FE.Setup}(C^*)$, and then recurses by computing

$$\begin{aligned}\mathsf{Obf}_{n-1} := \big(&\mathsf{Obfuscate}(\lambda, \mathsf{FE.Enc}(\mathsf{PK}_{C^*}, x_1, \ldots, x_{n-1}, 0, \mathsf{SK})),\\ &\mathsf{Obfuscate}(\lambda, \mathsf{FE.Enc}(\mathsf{PK}_{C^*}, x_1, \ldots, x_{n-1}, 1, \mathsf{SK}))\big).\end{aligned}$$

The obfuscation is then $(\mathsf{Obf}_{n-1}, \mathsf{FSK}_{C^*})$.

It is important to note in this construction that when we recurse, we view the $\mathsf{FE.Enc}$ operation as the *circuit* to embed in the functional secret key. Thus, if $\mathsf{FE.Enc}$ is a complex operation, it quickly becomes infeasible to even just view it as a circuit; for example, a single 1024-bit modular exponentiation requires *over four billion* gates [WMK17]. The best current FE construction uses trilinear maps [LT17], and thus the circuit representations would have to include such

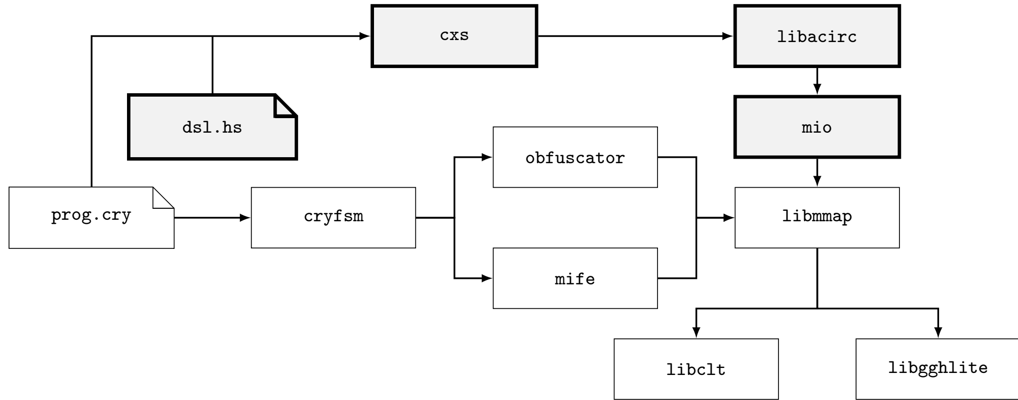

Figure 4.2: The 5Gen framework architecture, with new components introduced in this work in bold and gray boxes. The original 5Gen architecture takes as input a cryptol program (here given by `prog.cry`), which is fed into a compiler, `cryfsm`, to produce a matrix branching program. This is then fed into either an obfuscator or multi-input functional encryption implementation, which uses a multilinear map backend (`libmmap`) that supports both the CLT (`libclt`) and GGHLite (`libgghlite`) multilinear maps. In this work, we introduce four new components: (1) `cxs`, an arithmetic circuit compiler suite which takes as input either a cryptol program as before or a program written in a domain-specific language (here given by `dsl.hs`), (2) `libacirc`, a language for describing arithmetic circuits, and (3) `mio`, an implementation of circuit obfuscation and multi-input functional encryption.

trilinear map operations, resulting in a circuit of infeasible size. Thus, we conclude that the approach to obfuscation utilizing bootstrapping in its current form is much more expensive than the circuit approach we investigate in this work (and we also note that the non-black-box use of constant-degree mmaps seems inherent [PS16]). However, efficiency improvements to either the bootstrapping step or the underlying FE primitive could change this, and are interesting open problems.

4.2 Circuit Obfuscators

In this section we review the three existing circuit obfuscators in the literature: the Zimmerman obfuscator [Zim15], denoted by Zim, the Applebaum-Brakerski obfuscator [AB15], denoted by AB, and the Lin obfuscator [Lin16], denoted by Lin.

4.2.1 Obfuscation using Zim

In Zim, each "public" input bit x_i is encoded under symbol $X_{i,b}$; namely, the obfuscation contains mmap encodings $\hat{x}_{i,b} := [b, \alpha_i]_{X_{i,b}}$, where $b \in \{0,1\}$ and α_i denotes the check value associated with that input. For each "secret" input y_j, the obfuscation contains encoding $\hat{y}_j := [y_j, \beta_j]_Y$, where β_j denotes the check value associated with that input and Y is a common symbol for all secret inputs.

For a multiplication gate with inputs $[x]_{\mathcal{X}}$ and $[y]_{\mathcal{Y}}$, the index set of the output $[xy]_{\mathcal{X} \cup \mathcal{Y}}$ is the union of the input index sets $\mathcal{X}$ and $\mathcal{Y}$. For addition gates, when two inputs with equal index sets are added the encodings can be added directly. To support additions for inputs with unequal index sets, encodings of 1 are provided. These encodings are associated with a particular public input through its index set; namely, $\hat{u}_{i,b} := [1, 1]_{X_{i,b}}$. When an addition gate has input wires with unequal index sets, the evaluator multiplies each input the minimum number of times with the appropriate encodings of one to bring them into alignment.

The best case for an addition gate is when the index set of one input is a subset of the other. Then, the evaluator "raises" that input up the minimum amount, and the degree of the output encoding is not more than the highest degree input. For instance, suppose the inputs are $[x]_A$ and $[y]_{AB}$. Then the evaluator computes $[x]_A[1]_B + [y]_{AB} = [x + y]_{AB}$ and the output degree is two.

The worst case for an addition gate is when the input arguments have completely disjoint index sets. Then, the evaluator must "multiply in" the entire index set of the other for both arguments. After multiplying by each encoding of one, the output degree is then equivalent to a multiplication.

For instance, suppose we wish to add $[x]_A$ and $[y]_B$. Each input has degree two already. In order to add, we must calculate $[x]_A[1]_B + [y]_B[1]_A = [x + y]_{AB}$, resulting in output degree two, and thus the same degree as multiplying $[x]$ and $[y]$ would have given. Fortunately, addition gates with entirely disjoint index sets occur rarely in the circuits we consider. For instance, while a single round of AES has thousands of addition gates, only one gate has completely disjoint

Circuit	AB/Lin # Enc	AB/Lin κ	Zim/LZ # Enc	Zim/LZ κ	MIO # Enc	MIO κ
aes1r	66,691	22,999	66,307	239	33,669	128
ggm_1_32	690	250	594	17	399	14
ggm_1_128	2,706	250	2,322	17	1,551	14
ggm_2_32	1,218	34,706	1,122	81	667	74
ggm_2_128	4,770	34,706	4,386	81	2,587	74
ggm_3_32	1,746	5.0e+06	1,650	385	935	374
ggm_3_128	6,834	5.0e+06	6,450	385	3,623	374
ggm_4_32	2,274	7.2e+08	2,178	1,889	1,203	1,874
ggm_4_128	8,898	7.2e+08	8,514	1,889	4,659	1,874

Table 4.1: Number of encodings (# Enc) and multilinearity values (κ) for several circuits across the circuit-based program obfuscators considered in this work. See Appendix A for circuit details.

indices.

Computing κ. Let n be the number of inputs to the circuit C, and let $\mathsf{deg}(\cdot)$ denote the multiplicative degree of its input. Besides the evaluation of C, there are an additional n multiplications to construct the straddling sets in order to prevent "mix-and-match" attacks. Thus, we have

$$\kappa_{\mathsf{Zim}} \leq n + \sum_{i \in [n]} \mathsf{deg}(x_i).$$

Results. See Table 4.1 for κ values across various circuits. Note that the above formula is exact only when the circuit has a single output bit. However, for circuits with multiple output bits this formula may result in a large over-approximation of the actual value of κ needed. To calculate the "real" κ, we use a "dummy" mmap which tracks the degree of each encoded element. We can then take the maximum degree over all of the output encodings as the "real" κ. This can often make a huge difference in practice. For example, for aes1r, the above formula gives $\kappa = 567$, whereas using the dummy mmap approach gives $\kappa = 239$, a $2.5\times$ improvement.

4.2.2 Obfuscation using AB

The AB approach is similar to Zim in that it uses a "check slot" to enforce that the circuit is correctly computed. However, the particular details differ. In AB, every element is a pair (R, Z) where R contains some randomness and Z contains a secret masked by that randomness. For example, each "public" input bit x_i is encoded as $R^x_{i,b} := [r_{i,b,1}, r_{i,b,2}]$ and $Z^x_{i,b} := [r_{i,b,1} \cdot b, r_{i,b,2} \cdot \alpha_i]$, where $r_{i,b,1}$, $r_{i,b,2}$, and α_i are random. Likewise, "secret" inputs y_j are encoded as $R^y_j := [r_{j,1}, r_{j,2}]$ and $Z^y_j := [r_{j,1} \cdot y_j, r_{j,2} \cdot \beta_j]$ where $r_{j,1}$, $r_{j,2}$, and β_j are random.

This "El-Gamal"-style encoding directly supports both addition and multiplication: Suppose we have two pairs of encodings (R_1, Z_1) and (R_2, Z_2). An addition gate results in the pair of encodings $(R_1R_2, Z_1R_2 + R_1Z_2)$ and a multiplication gate results in the pair of encodings (R_1R_2, Z_1Z_2). The downside to this is of course that both addition and multiplication now require multiplication of encodings, and thus an (often substantial) increase in the overall degree.

To contrast this approach to Zim, consider the following (simplified) example. We wish to add two encodings $[x]_A$ and $[y]_{AB}$. In AB, these values are represented as $(R_x, Z_x) = ([r_x]_A, [r_x x]_A)$ and $(R_y, Z_y) = ([r_y]_{AB}, [r_y y]_{AB})$, with addition producing

$$(R_xR_y, Z_xR_y + Z_yR_x) = ([r_xr_y]_{AAB}, [r_xr_yx + r_xr_yy]_{AAB}).$$

In Zim, these values are represented directly by $[x]_A$ and $[y]_{AB}$, and the addition can be computed by raising x by the appropriate encoding of 1 and then adding the resulting encoding to $[y]_{AB}$. Thus, in AB we increase the degree, whereas in Zim the degree stays the same.

Computing κ. Let n be the input length and d the multiplicative degree of the circuit. Lin [Lin16] defines the notion of *type-degree*, which captures the growth of the degree given the El-Gamal-style encodings of AB. Let $\mathsf{typedeg}(i)$ be the type-degree of the ith input. Lin upper-bounds the multilinearity of this construction as

$$\kappa_{\mathsf{AB}} \leq 3 + 2n + d + \sum_{i \in [n+1]} \mathsf{typedeg}(i) \leq 5(t + n),$$

where t is the maximum type-degree of all the inputs, which Lin proves is $< 2^{\mathsf{depth}}$.

Results. See Table 4.1. As in Zim, we calculated these κ values using a dummy mmap. Note how much worse the κ values are for AB versus Zim. This is because all addition gates in AB require multiplying encodings; as κ is a function of the multiplicative degree of the top-level encodings, needing multiplication for both addition and multiplication quickly blows up κ.

4.2.3 Obfuscation using Lin

In a breakthrough result, Lin showed how to construct obfuscation from *constant-degree* mmaps [Lin16]. To do so, Lin designed a "bootstrap" circuit with constant degree which uses a polynomial-size input domain PRF. Lin's PRF is a variant of the "GGM" PRF of Goldreich, Goldwasser, and Micali [GGM84], which constructs a PRF directly from a PRG. While the particular details for now are not important (see Section 4.6 for more details on Lin's variant of the GGM PRF), we note that the core technique is to split the input into "Σ-*vectors*". A Σ-vector is a vector of bits, where to represent the integer $i \pmod n$ we have a "1" in the ith position and "0" elsewhere:

$$[\ \underbrace{0\ldots0}_{i-1}\ 1\ \underbrace{0\ldots0}_{n-i}\ ].$$

That is, a Σ-vector can be viewed as a unary encoding of a value in the domain $\{0,\ldots,|\Sigma|-1\}$. For example, a Σ-vector of length $|\Sigma| = 16$ can encode $\log_2(16) = 4$ possible "real" inputs. Put another way, a 16-bit input can be viewed as four Σ-vectors each of length $|\Sigma| = 16$, or alternatively, two Σ-vectors each of length $|\Sigma| = 256$ (since $16 = 4 \cdot \log_2(16) = 2 \cdot \log_2(256)$).

The advantage of Σ-vectors is that sub-string selection only requires multiplicative degree two. This works by multiplying the string pairwise with the appropriate Σ-vector and summing up the result. For instance, to get the jth bit of string $x \in \{0,1\}^n$, compute $\sum_{i\in[n]} e_i x_i$, where e is a Σ-vector encoding of j. This approach also generalizes to strings composed of longer sub-strings than single bits. Let ℓ be the length of the sub-string you wish to obtain and let $x \in \{0,1\}^{\ell n}$.

n	k	κ_{AB}	κ_{Zim}	κ_{Lin}	κ_{LZ}	κ_{MIO}
4	128	250	17	17	7	7
8	128	34,706	81	123	34	33
12	128	5.0e+06	385	977	163	161
16	128	7.2e+08	1,889	8,163	797	794

Table 4.2: Multilinearity values (κ) for the GGM PRF using AB, Zim, Lin, LZ, and MIO for various input lengths (n) and key lengths (k), in bits. For the Lin, LZ, and MIO obfuscators the inputs are treated as Σ-vectors with $|\Sigma| = 16$.

Then, to use e to select a sub-string, compute

$$\sum_{i \in [n]} e_i x_{i\ell+1} \| \sum_{i \in [n]} e_i x_{i\ell+2} \| \cdots \| \sum_{i \in [n]} e_i x_{i\ell+\ell}. \tag{4.1}$$

Lin introduced a variant of the Applebaum-Brakerski obfuscator that supports Σ-vectors. We denote this obfuscator as Lin. Note that when Σ-vectors are not used, Lin reduces to (a slight variant of) the AB scheme.

Computing κ. The computation of κ_{Lin} is the same as in AB.

Results. As mentioned above, Lin performs the same as AB when Σ-vectors are not in use; see Table 4.1. However, for certain functions, using Σ-vectors can have a huge improvement. In Table 4.2 we show the effect on κ when using Σ-vectors for the GGM PRF (cf. Section 4.6.2). We can see huge improvements going from κ_{AB} to κ_{Lin}, and κ_{Lin} is competitive with κ_{Zim}, at least for smaller input lengths.

4.3 Multi-input Functional Encryption from Circuits

In this section we present our new construction for single-key multi-input functional encryption, defined in Section 2.0.2. We begin with some high-level intuition before describing the scheme in detail in Section 4.3.1. In Section 4.3.2 we present a security proof, in Section 4.3.3 we describe some optimizations, and in Section 4.3.4 we show how to build an obfuscator from our MIFE construction.

Our MIFE construction expands the functionality of the scheme presented by Boneh et al. [BLR+15], which supports a single key represented as a *branching program*. Our construction allows the functional key to be described as an *arithmetic circuit*. To do so, we leverage techniques from the circuit obfuscation construction of Zimmerman [Zim15].

Program obfuscators provide the capability to evaluate the obfuscated function on any possible input. Thus, existing obfuscators provide encodings for both zero and one for each input bit. On the other hand, in the MIFE setting we need to enforce that the only inputs on which the functionality can be evaluated should correspond to the inputs encrypted in valid ciphertexts. That is, one should not be able to mix-and-match input bit values from different ciphertexts for the same slot.

A possible approach for achieving the above in the circuit setting would be to adapt ideas from Boneh et al.'s branching program construction. There, the authors introduced the notion of an *exclusive partition family*, which allows one to generate "straddling sets" for the bits in each MIFE ciphertext with the property that any evaluation that uses encodings from different ciphertexts can obtain an encoding at the zero-testing level only with negligible probability. However, these techniques crucially rely on the way branching programs are evaluated as a sequence of matrix multiplications. This property is no longer true for circuit evaluations and thus poses substantial challenges to extending the straddling techniques to work with circuits. Instead, we develop a different approach, inspired by the check value idea from circuit obfuscation constructions.

As discussed in Section 4.1 and Section 4.2, circuit obfuscators employ two main techniques: (1) they enforce that an evaluation can reach the zero-testing level if and only if it uses consistent assignment to each input bit, and (2) they ensure that the function evaluated is the intended function by using an additional check slot in the mmap encodings.

The first technique is not easily amenable to changes that could help prevent mix-and-match attacks across MIFE ciphertexts for the same MIFE slot. Instead, we propose a way to extend the check value technique to enforce that all encodings from a ciphertext are used consistently. Recall that in Zimmerman's construction, each input bit encoding has two slots, where the first

slot contains the actual bit value and the second slot has a fixed value; for example, the encodings for input bits zero and one for the ith input bit encode the pairs $[0, \alpha_i]$ and $[1, \alpha_i]$. Thus, any honest evaluation of the obfuscated function f on n input bits should be an encoding at the zero-testing level with value $f(\alpha_1, \ldots, \alpha_n)$ in the second slot. The obfuscation provides an encoding of $[0, f(\alpha_1, \ldots, \alpha_n)]$ at the zero-testing level, which can be subtracted and enables zero-testing encodings obtained with honest evaluation of the obfuscated function. The proof of security uses the Schwartz-Zippel lemma to show that the probability of an adversary obtaining an encoding at the zero-testing level with a zero value in the second slot in any other way than the honest evaluation is negligible.

We adapt this technique to the setting where we want to guarantee that a *subset* of the input bits are used consistently together. In particular, in the MIFE setting these subsets of bits correspond to the MIFE ciphertext for a particular MIFE slot. We handle this by using a designated mmap slot for *each* MIFE slot. The first mmap slot corresponds to the actual circuit evaluation, whereas the other n slots (for an n-input MIFE) correspond to a "check slot" for each MIFE slot.

Let $C : \{0,1\}^{d_1} \times \cdots \times \{0,1\}^{d_n} \to \{0,1\}^m$ be our n-input MIFE circuit, with the ith input being of length d_i, and for simplicity assume $m = 1$ (there is only one output bit) for now. A ciphertext for MIFE slot i contains a set of d_i encodings, where encoding $j \in [d_i]$ has random value α_j in the $(i+1)$th mmap slot. For a ciphertext in MIFE slot $k \neq i$, the $(i+1)$th mmap slot has value 1. The ciphertext for MIFE slot i also provides an encoding $\hat{w}_i$ that has value 1 in all of its mmap slots, except in its $(i+1)$th mmap slot it has the value

$$C(\underbrace{1, \ldots, 1}_{d_1}, \ldots, \underbrace{1, \ldots, 1}_{d_{i-1}}, \alpha_1, \ldots, \alpha_{d_i}, \underbrace{1, \ldots, 1}_{d_{i+1}}, \ldots, \underbrace{1, \ldots, 1}_{d_n}).$$

That is, the $(i+1)$th mmap slot contains the output of circuit C evaluated on all ones except for the ith MIFE slot which contains its associated α values.

On decryption, the special $\hat{w}_i$ values are multiplied in to reach the zero-testing level and then subtracted from the output encoding of the circuit, thus guaranteeing that an evaluator can

obtain an encoding at the zero-testing level with zero in its $(i+1)$th mmap slot *if and only if* it has used the bits of a ciphertext for the ith MIFE slot consistently.

Thus, on honest decryption, the resulting zero-testing encoding is $[C(\mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(n)}), 0, \ldots, 0]$, where $\mathsf{ct}^{(i)}$ denotes the ith ciphertext. On dishonest decryption, the resulting encoding is $[C^*, C_1^*, \ldots, C_n^*]$, where C^* denotes the output of the maliciously evaluated circuit, and C_i^* denotes the output of that same circuit on the ith check slot. By the Schwartz-Zippel lemma, these C_i^* values are non-zero with overwhelming probability.

We note that the increased number of mmap slots required in our construction only affects performance if this is more slots than available in the underlying mmap. However, for the CLT mmap [CLT13] which we use in our implementation, this does not occur when using reasonable security settings and input lengths. In particular, the number of slots in the CLT mmap is a function of both the security parameter and multilinearity. As an example, for a single gate circuit with security parameter 80, the CLT scheme requires 9,632 slots, and the 12-bit PRF we obfuscate (cf. Section 4.6) requires 13,111 slots, well above the number of inputs of these two circuits.

4.3.1 Construction

Let $C : \{0,1\}^{d_1} \times \cdots \times \{0,1\}^{d_n} \to \{0,1\}^m$ be an arithmetic circuit on boolean inputs, let $\mathsf{deg}(\mathbf{x}^{(i)})$ denote the maximum degree of the bits of the ith MIFE slot across all output bits, and let $\mathsf{deg}(\mathbf{x}_o^{(i)})$ denote the maximum degree of the bits of the ith MIFE slot for output bit $o \in [m]$. Our 1SK-MIFE[1] construction works as follows:

<u>$\mathsf{1SK\text{-}MIFE.Setup}(1^n, C)$:</u>

1. Define top-level index set

$$\mathcal{U} := Z \prod_{i \in [n]} W^{(i)} (X^{(i)})^{\mathsf{deg}(\mathbf{x}^{(i)})}.$$

[1]Security for 1SK-MIFE is defined in Section 2.0.2

2. Compute $(\mathsf{pp}, \mathsf{sp}, p_{\mathrm{ev}}, p_{\mathrm{chk}_1}, \ldots, p_{\mathrm{chk}_n}) \leftarrow \mathsf{Setup}(\mathcal{U}, n, 1+n)$.

3. Generate the following encoding:

$$\hat{C}^* := [0, \underbrace{1, \ldots, 1}_{n}]_{Z \prod_{i \in [n]} (X^{(i)})^{\mathsf{deg}(\mathbf{x}^{(i)})}}$$

4. For $i \in [n]$, generate the following encoding:

$$\hat{u}_i := [1, \underbrace{1, \ldots, 1}_{n}]_{X^{(i)}}.$$

5. Generate the following encoding:

$$\hat{z} := [\delta, \underbrace{1, \ldots, 1}_{n}]_{Z \prod_{i \in [n]} W^{(i)}},$$

where $\delta \leftarrow \mathbb{Z}_{p_{\mathrm{ev}}}$.

6. Set $\mathsf{sk} := (\mathsf{sp}, C)$ and $\mathsf{ek} := \left(\mathsf{pp}, C, \hat{z}, \{\hat{u}_i\}_{i \in [n]}, \hat{C}^*\right)$.

<u>$\mathsf{1SK\text{-}MIFE.Enc}(\mathsf{sk}, i, \mathbf{x} \in \{0,1\}^{d_i})$:</u>

1. For $j \in [d_i]$, generate the following encoding:

$$\hat{x}_j := [x_j, \underbrace{1, \ldots, 1}_{i-1}, \alpha_j, \underbrace{1, \ldots, 1}_{n-i}]_{X^{(i)}},$$

where $\alpha_j \leftarrow \mathbb{Z}_{p_{\mathrm{chk}_i}}$.

2. For $o \in [m]$, generate the following encoding:

$$\hat{w}_o := [0, \underbrace{1, \ldots, 1}_{i-1}, C_o^{\dagger}, \underbrace{1, \ldots, 1}_{n-i}]_{W^{(i)}},$$

where

$$C_o^\dagger := C(\underbrace{1,\ldots,1}_{d_1},\ldots,\underbrace{1,\ldots,1}_{d_{i-1}},\{\alpha_j\},\underbrace{1,\ldots,1}_{d_{i+1}},\ldots,\underbrace{1,\ldots,1}_{d_n})_o \in \mathbb{Z}_{p_{\mathrm{chk}_i}}.$$

3. Output ciphertext $\mathsf{ct} := \left(\{\hat{x}_j\}_{j\in[d_i]}, \{\hat{w}_o\}_{o\in[m]}\right)$.

<u>$\mathsf{1SK\text{-}MIFE.Dec}(\mathsf{ek}, \mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(n)})$:</u>

1. Parse ek as $\left(\mathsf{pp}, C, \hat{z}, \{\hat{u}_i\}_{i\in[n]}, \hat{C}^*\right)$.
2. For $i \in [n]$, parse $\mathsf{ct}^{(i)}$ as $\left(\{\hat{x}_j^{(i)}\}_{j\in[d_i]}, \{\hat{w}_o^{(i)}\}_{o\in[m]}\right)$.
3. For $o \in [m]$, evaluate the oth output of circuit C on inputs $\hat{x}_1^{(1)}, \ldots, \hat{x}_{d_1}^{(1)}, \ldots, \hat{x}_1^{(n)}, \ldots, \hat{x}_{d_n}^{(n)}$, using $\{\hat{u}_i\}$ as needed (as is done in Zim [Zim15]). Denote the final term as

$$\hat{C}_o := [C(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)})_o, (C_o^\dagger)^{(1)}, \ldots, (C_o^\dagger)^{(n)}]_{\prod_{i\in[n]} (X^{(i)})^{\deg(\mathbf{x}_o^{(i)})}}$$

4. For $o \in [m]$, compute

$$\hat{t}_o := \hat{z}\hat{C}_o - \hat{C}^* \prod_{i\in[n]} \hat{w}_o^{(i)}.$$

5. Output the bitstring resulting from running $\mathsf{ZeroTest}(\hat{t}_o)$ for $o \in [m]$.

4.3.2 Proof of Security

Theorem 4.3.1. *The construction in Section 4.3.1 is correct according to Definition 2.0.7.*

Proof. Correctness follows directly by inspection. □

Theorem 4.3.2. *The construction in Section 4.3.1 is secure according to Definition 2.0.8.*

Proof. In the generic mmap model the distributions of the encodings obtained during the encryption oracle queries are independent of the bit b. Thus, we need to argue that the answers obtained from successful zero testing queries in $\mathsf{Expt}_{C,Q,0}^{\mathsf{1SK\text{-}MIFE}}(\mathcal{A})$ and $\mathsf{Expt}_{C,Q,1}^{\mathsf{1SK\text{-}MIFE}}(\mathcal{A})$ are negligibly close.

The zero-testing returns a value different from $\perp$ only on encodings at the top-level index set $\mathcal{U}$. We consider the two main ways to obtain an encoding at the zero testing level: either using $\hat{C}^*$ or not using it.

1. *Using* $\hat{C}^*$: The only way to obtain encoding at the zero testing level from $\hat{C}^*$ is by multiplying it with $\prod_{i\in[n]} \hat{w}^{(i)}$.

2. *Not using* $\hat{C}^*$: Monomials that represent encodings at level $\mathcal{U}$ are of the form

$$\left(\prod_{j\in[d_i]} (\hat{x}_j^{(i)})^{m_i} (\hat{u}_j^{(i)})^{\mathsf{deg}(\hat{x}_j^{(i)})-m_i}\right)\hat{z} \tag{4.2}$$

   where $\hat{x}_j^{(i)}, \hat{u}_j^{(i)}, \hat{z}$ belong to ciphertexts for MIFE slot i.

Therefore, valid zero-testing queries will be linear combinations of monomials of the form either $\hat{C}^* \prod_{i\in[n]} \hat{w}^{(i)}$ or Equation 4.2.

An encoding successfully zero-tests if and only if the values in all of its mmap slots are zero. By the Computational Schwartz-Zippel Lemma [Zim15, Lemma 3.12], the values in the mmap slots are zero with non-negligible probability if and only if the polynomial over the encoded values evaluates to zero.

We consider mmap slot j for $j \in [2,\dots,n+1]$. Since each encoding in the ciphertext for MIFE slot $i = j-1$ contains a random value at mmap slot j and the encoding $\hat{w}$ from the ciphertext at MIFE slot i has value $C^\dagger := C(1,\dots,1,\{\alpha_j\}_{j\in[d_i]},1,\dots,1)$ at mmap slot j, in order to obtain a polynomial that evaluates identically to zero, this polynomial needs to have as a divisor the polynomial $C(1,\dots,1,\{\hat{x}_j^{(i)}\}_{j\in[d_i]},1,\dots,1)\hat{z} - \hat{C}^*\hat{w}^{(i)} \cdot \underbrace{\star\cdots\star}$, where $\star$ denotes an encoding that has value one in mmap slot j (for clarity, we have omitted the multiplications with encodings $\hat{u}_j^{(i)}$). In other words, this means that all monomials can use encodings only from a single ciphertext at MIFE slot i. Applying this reasoning to all MIFE slots we conclude that the polynomials that evaluate to zero must be of the form $C(\{\hat{x}_j^{(i)}\}_{i\in[n],j\in[d_i]}) - \hat{C}^* \prod_{i\in[n]} \hat{w}^{(i)}$ (again omitting encodings $\hat{u}_j^{(i)}$), except with negligible probability.

Since for all admissible tuples of queries in the MIFE security game the circuit C evaluates to the same value independent of the challenge bit b, it follows that the encodings that successfully zero-test will also be independent of the challenge bit b, completing the proof. $\square$

4.3.3 Optimizations

We note an optimization that can help reduce the overall multilinearity by two for certain functions. When using MIFE on a circuit with constants, the naive approach is to utilize one slot to encode the constants. However, this slot can instead be "rolled into" the encodings generated in $\mathsf{1SK\text{-}MIFE.Setup}$, as follows. Suppose the $(n+1)$th MIFE slot contains the constants. Instead of computing $\hat{C}^* := [0, 1, \ldots, 1]_{Z \prod_{i \in [n+1]} (X^{(i)})^{\mathsf{deg}(\mathbf{x}^{(i)})}}$, we directly compute the combination of $\hat{C}^*$ and $\hat{w}_o^{(n+1)}$; that is, we replace $\hat{C}^*$ with $\hat{w}_o^{(n+1)} := [0, \underbrace{1, \ldots, 1}_{n}, C_o^\dagger]_{ZW^{(n+1)} \prod_{i \in [n+1]} (X^{(i)})^{\mathsf{deg}(\mathbf{x}^{(i)})}}$ for $o \in [m]$. Now, the right-hand-side computation in Step (4) of $\mathsf{1SK\text{-}MIFE.Dec}$ is just $\prod_{i \in [n+1]} \hat{w}_o^{(i)}$. This reduces the multilinearity by one for functions with constants where the number of inputs is larger than the overall degree of the circuit.

We can reduce this computation further by combining $\hat{w}_o^{(i)}$ with, say, the $\hat{w}_o^{(1)}$ values corresponding to the first MIFE slot. Thus, the final right-hand-side product becomes $\prod_{i \in [n]} \hat{w}^{(i)}$, reducing the multilinearity by two versus the naive approach.

Computing κ. The main cost of our construction is the computation of C, plus one additional multiplication to reach the top-level index set. However, if the multilinearity from computing C is less than the number of inputs, the right-hand-side computation of Step (4) of $\mathsf{1SK\text{-}MIFE.Dec}$ dominates. Thus, we get the following:

$$\kappa \leq \max\{1 + \sum_{i \in [n]} \mathsf{deg}(\mathbf{x}^{(i)}), n\}.$$

4.3.4 Obfuscation from MIFE

Our MIFE construction immediately gives us an obfuscation scheme using the transformation of Goldwasser et al. [GGG+14]: set each MIFE slot to be a single bit, with the obfuscation being the $2n$ encryptions corresponding to the zero and one values in each MIFE slot, and evaluation being the MIFE decryption operation. We denote this construction by MIO. Note that we can also directly support Σ-vectors by encrypting each $\sigma \in \Sigma$ for each MIFE slot.

$\underline{\mathsf{MIO.Obfuscate}(1^n, C)}$:

1. Compute $(\mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{1SK\text{-}MIFE.Setup}(1^n, C)$.
2. For $i \in [n]$, $b \in \{0, 1\}$, compute $\mathsf{ct}_{i,b} \leftarrow \mathsf{1SK\text{-}MIFE.Enc}(\mathsf{sk}, i, b)$.
3. Output the following as the obfuscated program: $\big(\mathsf{ek}, \{\mathsf{ct}_{i,0}, \mathsf{ct}_{i,1}\}_{i \in [n]}\big)$.

$\underline{\mathsf{MIO.Evaluate}(\mathsf{Obf}, \mathbf{x})}$:

1. Parse Obf as $\big(\mathsf{ek}, \{\mathsf{ct}_{i,0}, \mathsf{ct}_{i,1}\}_{i \in [n]}\big)$.
2. Output $\mathsf{1SK\text{-}MIFE.Dec}(\mathsf{ek}, \mathsf{ct}_{1,x_1}, \ldots, \mathsf{ct}_{n,x_n})$.

Computing κ. This approach gives us the same κ values as our MIFE construction. We also note that MIO requires many fewer encodings than existing approaches, as we are in some sense trading the use of *encodings* with *mmap slots* to enforce security. See Table 4.1 for some examples; roughly, MIO requires up to $2\times$ fewer encodings than all existing approaches, which directly impacts the obfuscation time and size.

Results. See Table 4.1 and Table 4.2 for results. We can see that MIO is better than all existing schemes, both in terms of κ values as well as the number of encodings needed.

4.4 Compiling Circuits for Obfuscation

The running time of circuit obfuscation is directly related to the multilinearity of the underlying mmap: the larger the multilinearity, the longer the running time and resulting size. Indeed, for

existing mmaps, the running time is exponential in the multilinearity. As the multilinearity is always greater than or equal to the multiplicative degree, reducing the multiplicative degree often has a direct impact on the multilinearity. In this section we focus on the multiplicative degree, whereas throughout the rest of the paper the focus is on multilinearity.

The degree is calculated as follows. As we evaluate an arithmetic circuit, the degree of an addition gate is the maximum degree of its inputs, whereas the degree of a multiplication gate is the sum of the degree of its inputs (see Figure 4.1). Thus, in the worst case the degree of the whole circuit is exponential in its depth — this occurs when a circuit contains only multiplication gates. However, in the best case, the degree can be much lower — this occurs when the multiplication gates are "spread out" effectively in the circuit. The primary goal when compiling circuits is in reducing the degree required, even at the cost of increasing the total number of gates.

Existing circuit compilers either minimize the number of gates [HFKV12, MGC⁺16] or the depth [BHWK16]. As an early attempt, we took a similar approach using off-the-shelf tools: taking a `Cryptol` [Cry] function as input, we used the Software Analysis Workbench [Gal16] to generate circuits composed of `And` and `Not` gates, optimized them using either `ABC` [BM10], a tool for synthesizing and optimizing binary sequential logic circuits, or `Yosys` [WGK13], a tool for synthesizing and optimizing Verilog scripts, and then finally translated the outputs to an arithmetic circuit. These tools were somewhat effective, but they did not provide the ability to use custom optimizations to reduce the degree and thus often produced poorly performing circuits. Thus, we built our own circuit compiler to address this gap. As an example, for 1-round AES we were able to reduce the degree from 14,900 to 33 when using our compiler.

The compiler itself is an embedded domain specific language (DSL) in Haskell which directly constructs arithmetic circuits from addition, multiplication, and subtraction gates. This is less a limitation than it seems, since we can use the full power of Haskell to construct circuits. For example, access to low-level representations allows for clever circuit optimizations, such as replacing a subroutine with a lookup table. The DSL can also compose circuits, which allows us to use heavily optimized sub-circuits as subroutines. Finally, the DSL provides generic optimizations

specifically tailored to reduce the degree.

In Section 4.4.1 we discuss circuit optimizations. This includes improvements to the way a circuit is initially constructed, as well as optimizations we apply after the circuit is created. Then, in Section 4.4.2 we show how well the optimizations perform on a circuit that computes a single round of AES.

4.4.1 Circuit Optimizations

In this section we describe each of our circuit optimization approaches: using off-the-shelf optimizers (Section 4.4.1.1), using a DSL (Section 4.4.1.2), encoding lookup tables (Section 4.4.1.3), folding constants (Section 4.4.1.4), and circuit flattening (Section 4.4.1.5).

4.4.1.1 Off-the-Shelf Boolean Circuit Optimizers

Our early compiler generated boolean circuits which could then be fed into off-the-shelf optimizers. We were inspired by TinyGarble [SHS+15] to use `ABC` [BM10] and `Yosys` [WGK13], tools which are used by hardware engineers for optimizing circuits. We then converted the optimized boolean circuit to arithmetic. The advantage of this approach is that existing optimizers are well developed, easy to use, and effective. Using this method, if we compile the AES S-Box directly as an arithmetic circuit, it has degree 283. Optimizing with `ABC` takes the degree down to 250, and further optimization with `Yosys` takes the degree down to 220. However, these tools optimize for size, not degree, and the resulting degrees are well beyond values that we could hope to obfuscate in a reasonable amount of time.

4.4.1.2 Using a Domain-specific Language

In order to take advantage of low-level circuit optimizations that are not captured by existing off-the-shelf tools, we developed our own DSL for building circuits. The basic idea is to build a circuit directly from gates in Haskell. The user primarily interacts with "Refs", which are indices into an array of wires. We provide functions for addition, subtraction, multiplication, etc., based on Refs. There is also a mechanism for avoiding duplicate gates. The user can use Haskell functions to create their circuits, dynamically passing Refs around, and existing circuits can be imported as sub-circuits. This allows us to take the best compilation method for every circuit, or to pre-compute sub-circuits that have a long optimization step. Finally, the DSL provides tools for exploring new optimization techniques (both during and after compilation), as described below.

4.4.1.3 Encoding a Lookup Table as a Circuit

One simple but effective optimization we can do is to replace an n-input circuit with a lookup table of degree n. The cost is an exponential blowup in the number of gates in the circuit, corresponding to every possible input. However, since size is not the limiting factor for our obfuscation schemes, we apply this optimization whenever we can.

As an example, consider the circuit corresponding to a truth table with single-bit entries $\begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}$, where on input i the circuit returns the ith bit of the table. We can convert this circuit to the formula

$$((1 - i_1) \cdot i_0) + (i_1 \cdot (1 - i_0)),$$

where $i_1 i_0$ corresponds to the base-2 representation of i, giving us a circuit of degree two.

Using the DSL and the above lookup table encoding reduces the degree of the AES S-Box from 220 to 8, a 27× improvement.

4.4.1.4 Folding Constants

Another simple optimization which is surprisingly effective is "constant folding". Namely, we scan the circuit for gates that add or multiply constants and replace such gates with new constants. We can also remove gates that add or subtract zero or multiply by one. Our circuits often contain these gates since we use $1-x$ to simulate $\mathsf{Not}(x)$. Such gates also occur if we fix input bits to a constant.

4.4.1.5 Circuit Flattening

This optimization uses the fact that our arithmetic circuits emulate boolean circuits in that every wire only ever carries zero or one. We can take advantage of this to reduce the circuit degree as follows: (1) convert the circuit to a polynomial; (2) "expand" the polynomial by converting it from a product of sums to a sum of products; (3) remove all exponentiations, possible because the variables in the polynomial are boolean; and (4) simplify and convert back to circuit form.

As an example, consider the circuit represented by the polynomial $(1-x_1)(1-x_1x_2)$. Expansion produces the polynomial $(1-x_1-x_1x_2+x_1^2x_2)$. Since the inputs are bits we can remove all exponents from the polynomial. Simplifying, we get the polynomial $(1-x_1)$, which has degree one, in comparison to the original which has degree three.

Polynomial expansion has exponential blow-up, so this optimization only works for sufficiently small circuits (experimentally, we found around depth 14 to be the limit). We thus locate high-degree *sub-circuits*, flatten these independently, and then use the DSL to stitch them back in, repeating until a fixed-point is reached. This method takes the degree of 1-round AES-128 from 57 to 33.

Circuit	Opt. Level	# Gates	# Muls	Degree
aes1r	-O0	22,368	5,600	57
	-O1	22,368	5,600	57
	-O2	80,564	9,203	33
aes1r_64_1	-O0	1,101	569	41
	-O1	820	420	26
	-O2	1,324	614	18

Table 4.3: Comparison of DSL optimizations. 'Opt. Level' denotes the optimization level used ('-O0' denotes no optimizations; '-O1' denotes constant folding; and '-O2' denotes constant folding and sub-circuit flattening); '# Gates' denotes the total number of gates in the circuit; '# Muls' denotes the total number of multiplication gates; and 'Degree' denotes the multiplicative degree.

4.4.2 Optimization Results

See Table 4.3 for a comparison of the various DSL optimization approaches on two variants of one-round of AES-128: `aes1r` denotes one-round of AES-128, and `aes1r_64_1` denotes one-round of AES-128 with 64-bits of the input fixed and only one output bit. We can see that in both cases, we see benefits to using our circuit flattening optimization, reducing the degree from 57 to 33 in the case of `aes1r` and reducing the degree from 26 to 18 in the case of `aes1r_64_1`. Constant folding only sees a benefit for `aes1r_64_1`; this is because in that circuit 64 input bits are fixed, and thus can be folded into the computation, reducing the degree from 41 to 26.

Note that the circuit flattening optimization often increases both the total number of gates and the number of multiplication gates. For example, using sub-circuit flattening increases the number of gates in the `aes1r` circuit by almost $4\times$ and the number of multiplication gates by almost $2\times$. However, as mentioned before, *degree* is the main efficiency bottleneck, and thus this extra increase is largely irrelevant from an efficiency standpoint given the reduction in degree.

4.5 Implementation

We implemented the MIFE scheme described in Section 4.3 and all of the obfuscators discussed in Section 4.2 and Section 4.3. We have packaged these into a program, `mio`, containing around 20,000 lines of C code and using `libmmap` as the underlying mmap library (and in particular, the instantiation of the CLT mmap [CLT13] available as part of `libmmap`). We also developed

a language and associated library, `libacirc`, for describing arithmetic circuits. All the code is available at https://github.com/5GenCrypto/.

Attacks on CLT While our MIFE and obfuscation schemes are secure in the composite-order mmap generic model, in our implementation we instantiate this generic model using the CLT mmap [CLT13]. Unfortunately, the CLT mmap is prone to several attacks. Most relevant to our constructions, Coron et al. [CGH+15] demonstrated an attack on the Zimmerman construction for the specific circuit comprised of the product of an odd number of inputs. However, it is not clear how to extend their attack to arbitrary circuits, and in particular, to PRFs. Intuitively, it appears that any successful partitioning of the input space needed in their attack would lead to an attack on the underlying PRF, although this remains to be formalized.

More recently, Coron et al. [CLLT17] demonstrated attacks on matrix branching program constructions using the CLT mmap, but again, we are not aware of how to map this attack to the more general circuit approach. Existing attacks rely on some inherent structure of the computation, be it matrix multiplication using branching programs or multiplications when attacking the product circuit in Zimmerman's construction. In addition, embedding PRFs in obfuscation constructions based on the GGH mmap [GGH13a] has been used to eliminate all known attacks [GMM+16], albeit in the branching program context. This seems to suggest that when obfuscating PRFs themselves, the "lack of structure" provided by the PRF prevents existing attacks from working. Thus, while we do not have a proof that our construction circumvents all known attack techniques, we believe existing attacks do not affect our construction when applied to obfuscating PRFs.

4.6 Performance Results

We found that for functions that can be mapped efficiently to finite automata, the matrix branching program approach of 5Gen is superior. In particular, for order-revealing encryption and point function obfuscation we were unable to produce circuits with smaller κ than 5Gen. However,

as the complexity of the circuit grows, the circuit-based approach quickly becomes superior. Indeed, we were unable to even *compile* branching program representations for any of the circuits considered below.

We investigate two function classes to obfuscate: AES (cf. Section 4.6.1) and the Goldreich, Goldwasser, Micali (GGM) PRF (cf. Section 4.6.2). We explored a number of other PRF constructions, including MiMC [AGR+16] and the PRF of Applebaum and Raykov [AR16]. However, both approaches require finite field operations resulting in circuits of very high degree. For example, we found the Applebaum-Raykov PRF with 8-bit input and 24-bit key to have multiplicative degree greater than 10^{26}.

We note that all of the obfuscators we implemented only satisfy the weaker *indistinguishability obfuscation* definition; thus, we assume heuristically that our constructions are *virtual black-box* obfuscators for the specific functions considered below. Due to its efficiency over the other obfuscators, both in terms of multilinearity and number of encodings, all of the below experiments are done using our MIFE-based construction from Section 4.3.

4.6.1 AES

Obfuscating AES can be seen as the "holy-grail" of program obfuscation due to its wide-spread use in industry. Unfortunately, obfuscating the entire 10-round AES-128 construction is well beyond our capabilities at the moment. For a single round of AES we can achieve $\kappa = 128$, but for even two rounds of AES this balloons to over 2,000.

Recall from Table 4.3 that the multiplicative degree of our circuit is 33, yet the resulting multilinearity is 128. This is due to the fact that our obfuscator has a minimal multilinearity equal to the number of inputs, and thus in some sense we cannot take advantage of the low multiplicative degree in this case. Thus, an interesting open problem is constructing a circuit obfuscation scheme that has multilinearity *independent* of the number of inputs, while also making only black-box queries to the underlying mmap to avoid the efficiency bottleneck of using mmaps

in a non-black-box way (cf. Section 4.1.2).

4.6.2 Goldreich-Goldwasser-Micali PRF

AES is not particularly suited to our program obfuscation approach due to its many rounds, and thus high degree. As mentioned above, we explored other PRFs but found the Goldreich, Goldwasser, and Micali (GGM) PRF [GGM84] as the most feasible approach.

Let $F_k : \{0,1\}^n \to \{0,1\}^n$ denote a PRF. GGM introduced a way to construct F_k directly from a stretch-two[2] PRG $G : \{0,1\}^n \to \{0,1\}^{2n}$ as follows. Let $G(k) = G_0(k)\|G_1(k)$, where $G_i : \{0,1\}^n \to \{0,1\}^n$ denotes the ith n-bit block of output of G. The idea is to repeatedly apply G, using the input bits of the PRF as an index into which half of the output of G to return. Namely, letting $x := x_1 \cdots x_n$, we define $F_k(x) := G_{x_n}(G_{x_{n-1}}(\cdots(G_{x_1}(k))))$.

Lin [Lin16] used a variant of the GGM PRF in her construction of obfuscation from constant-degree mmaps (cf. Section 4.1.2). Her variant uses a PRG with *polynomial* stretch (as opposed to the GGM construction, which only requires stretch two), which allows for minimization of the depth and thus degree of the resulting circuit. In particular, Lin showed that by using a special unary "Σ-vector" encoding of the input (cf. Section 4.2.3) and a polynomial stretch PRG we can minimize the depth of the resulting PRF.

More formally, let $x := \sigma_1 \cdots \sigma_n$, where $\sigma_i \in \Sigma$, and let $|\Sigma|$ denote the length of Σ. Using a PRG $G : \{0,1\}^n \to \{0,1\}^{|\Sigma|n}$, our PRF becomes $F_k(x) := G_{\sigma_n}(G_{\sigma_{n-1}}(\cdots(G_{\sigma_1}(k))))$. Note that the "real" length of x (i.e., the number of bits of input that x can encode) corresponds to $n \cdot \log_2(|\Sigma|)$, since a Σ-vector of length $|\Sigma|$ corresponds to a unary encoding of a value in the set $\{0, \ldots, |\Sigma|-1\}$. In particular, this approach reduces the number of PRG evaluations by $\log_2(|\Sigma|)$.

Thus, the main "cost" (in terms of degree) of the PRF becomes the particular PRG it is instantiated with. To measure this, we calculated the depth, degree, and multilinearity of the GGM PRF with the PRG instantiated by an identity function. When using four Σ-vector inputs each of length 16 (corresponding to 64 bits of "real" input) and a key length of 128, we get a

[2]We define a *stretch-t* PRG to be one that on n-bit input, produces a tn-bit output.

depth of 20, a degree of 5, and a multilinearity of 6. On the other hand, instantiating the PRF with an actual PRG (in this case, Goldreich's PRG as discussed below), we get a depth of 48, degree of 781, and multilinearity of 1,126.

4.6.2.1 Selecting a PRG for the GGM PRF

As the PRG choice greatly effects the overall degree of the circuit, and thus the required multilinearity when obfuscating, choosing an appropriate PRG is of vital importance to efficiency. In particular, we would like a polynomial-stretch PRG in NC^0 (the class of boolean circuits with constant depth) due to its low depth, and thus low degree. One of the main candidate PRGs in this space is by Goldreich [Gol00]: letting $G : \{0,1\}^n \to \{0,1\}^m$ be our PRG, for some fixed k take m random k-tuples of the n input bits and apply a predicate $P : \{0,1\}^k \to \{0,1\}$ to each tuple, producing m bits of output. Due to the parallelizability of this approach, the degree of G is simply the degree of the predicate P.

There are various possible choices of both the predicate P and its input size k. Goldreich [Gol00] suggested choosing P at random. O'Donnell and Witmer [OW14] suggested the `xor-and` predicate

$$P(x_1, x_2, x_3, x_4, x_5) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 x_5 \pmod 2.$$

Finally, Applebaum and Lovett [AL16] suggested the `xor-maj` predicate

$$P(x_1, \ldots, x_d) = x_1 \oplus \cdots \oplus x_{\lfloor d/2 \rfloor} \oplus \mathrm{Majority}(x_{\lfloor d/2 \rfloor + 1}, \ldots, x_d).$$

In order to choose the most efficient predicate, we compare `xor-maj` and `xor-and`, along with the (completely insecure) `linear` predicate $P(x_1, \ldots, x_5) = \bigoplus x_i$ to get an idea of the "simplest" predicate we could hope for. See Table 4.4 for the results.

We found that all three predicates have roughly the same multilinearity, with `xor-and` and

Predicate	n	m	# Gates	# Muls	Depth	Degree	κ
`linear`	32	32	504	126	9	5	33
`linear`	32	128	1,904	476	9	5	33
`linear`	64	64	1,024	256	9	5	65
`linear`	64	128	2,028	507	9	5	65
`linear`	128	128	2,028	507	9	5	129
`xor-and`	32	32	414	126	7	5	33
`xor-and`	32	128	1,609	484	7	5	33
`xor-and`	64	64	827	254	7	5	65
`xor-and`	64	128	1,645	502	7	5	65
`xor-and`	128	128	1,656	510	7	5	129
`xor-maj`	32	32	829	414	9	5	32
`xor-maj`	32	128	3,159	1,622	9	5	32
`xor-maj`	64	64	1,914	895	9	6	64
`xor-maj`	64	128	3,756	1,783	9	6	64
`xor-maj`	128	128	5,350	2,794	10	7	128

Table 4.4: Circuits computing Goldreich's PRG for various choices of predicate. `linear` denotes the predicate $P(x_1, \ldots, x_5) := \bigoplus x_i$; `xor-and` denotes the predicate $P(x_1, \ldots, x_5) := x_1 \oplus x_2 \oplus x_3 \oplus x_4 x_5$; `xor-maj` denotes the predicate $P(x_1, \ldots, x_d) := x_1 \oplus \cdots \oplus x_{\lfloor d/2 \rfloor} \oplus \text{Majority}(x_{\lfloor d/2 \rfloor + 1}, \ldots, x_d)$, where d is set to $\log n$; 'n' denotes the number of input bits; 'm' denotes the number of output bits; '# Gates' denotes the total number of gates; '# Muls' denotes the number of multiplication gates; 'Depth' denotes the depth of the circuit; 'Degree' denotes the multiplicative degree of the circuit; and 'κ' denotes the best multilinearity achieved.

`linear` being slightly better than `xor-maj`. This is due to the fact that we are computing boolean operators using arithmetic circuits. In particular, the XOR operation is no longer cheap: $\text{XOR}(x, y) := x + y - 2xy$, thus requiring one multiplication and three linear operations, whereas $\text{AND}(x, y) := xy$, costing one multiplication but no linear operations. Since the main cost is the multiplicative degree, we can ignore the cost of the linear operations and thus XOR and AND end up costing the same.

We also experimented with the block-local PRG of Barak et al. [BBKK17], but found its performance worse than the above instantiations. Thus, we chose to instantiate our PRG in the GGM PRF construction using the `xor-and` predicate.

		κ	
n	k	Boolean	Σ
4	128	14	7
8	128	74	33
12	128	374	161
16	128	1,874	794

Table 4.5: Multilinearity values for the GGM PRF obfuscated using MIO for both boolean and Σ-vector inputs with $|\Sigma| = 16$. 'κ' denote the multilinearity; 'n' denotes the number of "real" input bits; 'k' denotes the key length; 'Boolean' denotes that the inputs are represented as bits; and 'Σ' denotes that the inputs are represented as Σ-vectors. As an example, for $n = 16$ and using the Σ vector representation, we have four Σ-vectors and thus $4 \cdot 16 = 64$ input bits, which corresponds to $4 \cdot \log_2(16) = 16$ "real" input bits.

4.6.2.2 Implementing the GGM PRF

We implemented the GGM PRF within our DSL for both boolean inputs and Σ-vector inputs; see Table 4.5 for the results. We found that Σ-vector inputs produced much smaller κs than their boolean input counterparts. This is due to two reasons: (1) we require fewer applications of the PRG, reducing the overall depth and thus degree of the circuit, and (2) we can produce very small lookup tables by directly using the Σ-vector as the index (i.e., the e value in Equation (4.1)).

4.6.2.3 Performance Results

See Table 4.6 for performance results. All results were run on a machine with 2 TB of RAM and four Xeon CPUs running at 2.1 GHz, with sixteen cores per processor and two threads per core (resulting in 128 "virtual" cores). We used $\lambda = 80$ throughout. Due to the long running time, the numbers correspond to a single execution.

The largest circuit we were able to obfuscate and evaluate was a 12-bit GGM PRF with a 64-bit key[3]. This took 3.7 hours to obfuscate, resulting in an obfuscation of 120 GB and an evaluation time of 67 minutes. While still far from practical, this is by far the most complex function obfuscated to date that we are aware of. In particular, Lewi et al. [LMA+16] obfuscated an 80-bit point function, and Halevi et al. [HHSSD17] obfuscated a 100-state non-deterministic

[3] We successfully obfuscating a 12-bit GGM PRF with a 128-bit key, but evaluation ran out of memory.

n	$\lvert\Sigma\rvert$	# PRGs	k	# Gates	κ	Obf time	Obf size	Eval time
8	256	1	32	23,710	7	10 h	121 GB	4.4 m
8	256	1	64	27,692	7	10 h	121 GB	4.7 m
8	256	1	128	29,889	7	10 h	121 GB	4.9 m
8	16	2	32	8,580	33	127 m	11 GB	11 m
8	16	2	64	16,138	33	127 m	12 GB	17 m
8	16	2	128	31,431	33	130 m	13 GB	29 m
12	64	2	32	32,998	33	3.7 h	119 GB	42 m
12	64	2	64	62,227	33	3.7 h	120 GB	67 m
12	64	2	128	121,798	33	3.7 h	122 GB	—

Table 4.6: Obfuscation details for various GGM PRF circuits. 'n' denotes both the number of "real" input bits (i.e., $n = \#\mathsf{PRGs} \cdot \log_2(\lvert\Sigma\rvert)$) and the number of output bits of the circuit; '$\lvert\Sigma\rvert$' denotes the size of the Σ-vectors; '# PRGs' denotes the number of applications of the PRG; 'k' denotes the key length; '# Gates' denotes the number of gates in the circuit; 'κ' denotes the required multilinearity of the mmap; 'Obf time' denotes the obfuscation time; 'Obf size' denotes the obfuscation size; 'Obf RAM' denotes the maximum amount of RAM used during obfuscation; 'Eval time' denotes the evaluation time; and 'Eval RAM' denotes the maximum amount of RAM used during evaluation. '—' denotes that we ran out of memory.

finite automaton with 68 input bits. On the other hand, our obfuscated circuit contains 48 input bits (albeit in Σ-vector form), 12 output bits, and 62,227 gates.

We also compared the tradeoff of Σ-vector length versus number of PRG applications. As the number of PRG applications increases, the overall multilinearity does as well. Indeed, for a single PRG application, we can achieve $\kappa = 7$, whereas with two applications this jumps to 33, and with three applications we reach 161, outside the realm of runnability. Thus, playing with the Σ-vector length, we compared the running time of an 8-bit PRF with $\lvert\Sigma\rvert = 256$ (resulting in one application of the PRG and thus $\kappa = 7$) versus $\lvert\Sigma\rvert = 16$ (resulting in two applications of the PRG and thus $\kappa = 33$). Interestingly, the PRF with the lower κ ended up taking much longer to obfuscate and resulted in a much larger obfuscation. This is due in a large part to the huge number of encodings needed when using $\lvert\Sigma\rvert = 256$. As an example, for key length 128 we need 67,872 encodings for $\lvert\Sigma\rvert = 256$ versus 1,064 encodings for $\lvert\Sigma\rvert = 16$. Thus, even though κ is around $4\times$ larger in the latter case, we need to encode $64\times$ fewer values, reducing the obfuscation time (10 hours versus 130 minutes) and size (121 GB versus 13 GB). However, we do note that the evaluation time is much faster for $\lvert\Sigma\rvert = 256$: 4.9 minutes versus 29 minutes using our previous

example. This presents an interesting tradeoff of obfuscation time/size versus evaluation time.

4.7 Conclusion

In this chapter, we present a thorough investigation of circuit-based multi-input functional encryption (MIFE) and program obfuscation, introducing a new MIFE scheme and associated program obfuscator that performs better than all existing approaches when obfuscating pseudorandom functions (PRFs). This allowed us to obfuscate the Goldreich-Goldwasser-Micali (GGM) PRF for a small number of inputs; however, the multilinearity quickly increases as we increase the input size, preventing us from being able to obfuscate "real-world" input sizes.

This work motivates several interesting research questions. The running time of obfuscating the GGM PRF depends heavily on the pseudorandom generator (PRG) used within the PRF; can one construct a more "obfuscation-efficient" PRG for this use case? Can one construct a lower degree PRF than the GGM PRF? More generally, can one construct a more efficient circuit obfuscator than our MIFE construction, and in particular, one that has multilinearity *independent* of the input length while still being black-box in the mmap? Finally, as all of these constructions rely heavily on the efficiency of the underlying mmap, a major open question is the construction of more efficient composite-order mmaps.

5 Public-Key Function-Private Functional Encryption for Wildcard Pattern-Matching

In this chapter, we give the first *function-private* functional encryption scheme for wildcard pattern matching. In order to understand how our construction works, we describe the BKMPRS obfuscator in more detail. The obfuscator first chooses a random polynomial F such that $F(0) = 0$. Then for each index $i \in [n]$ and bit assignment $b \in \{0,1\}$, the obfuscator sets $q_{i,b} := F(2i + b)$ or $q_{i,b} \leftarrow \mathbb{Z}_p$ randomly depending on whether that particular bit assignment is matched by the pattern. That is, if the pattern is $\mathsf{pat} \in \{0,1,?\}^n$, then the bit assignment match occurs when $\mathsf{pat}[i] = b$ or $\mathsf{pat}[i] =?$. The obfuscator encodes the $q_{i,b}$s in the group, and releases the $2 \times n$ table consisting of these encodings. Bishop et al. show that it is hard to distinguish a with purely random encodings from one created for a pattern sampled from the distribution of patterns with at most $3/4n$ wildcards (improved by Bartusek & Ma to at most $1 - \omega(\log \lambda)$ wildcards).

The evaluator chooses the encodings from the table that correspond to their input $x \in \{0,1\}^n$. Then, they have picked up the encodings $g^{q_{i,x_i}}$ for $i \in [n]$. They compute the Lagrange coefficients L_i for interpolating F at 0 such that $F(0) = \sum_{i\in[n]} L_i q_{i,x_i}$ under the assumption that each $q_{i,x_i} = F(2i + x_i)$ is a real polynomial evaluation. Then, they exponentiate the obfuscation encodings with the Lagrange coefficients and sum the result, effectively interpolating F at 0 in the exponent. That is, they compute $\sum_{i\in[n]}(g^{q_{i,x_i}})^{L_i}$. If the input is matched by the pattern, then all the chosen encodings will be polynomial evaluations, and the result of interpolation will be $g^{\sum_{i\in[n]} q_{i,x_i} L_i} = g^{F(0)} = g^0$. If the input does not match the pattern, then there will be at least one random value mixed in, and interpolation will fail.

The intuition of our wildcard encryption construction is this: we would like to extend the BKMPRS obfuscator to allow it to be evaluated on secret inputs. A ciphertext must then contain the necessary information to evaluate the obfuscation. In BKMPRS, Lagrange coefficients are

used to evaluate a secret polynomial. These coefficients depend on the plaintext, and would leak information in our scheme if they were given in the clear. So, we hide them in the exponent of the group using a random mask. Then, to decrypt, we multiply the Lagrange coefficients with the appropriate wildcard obfuscation encodings which are evaluations of a random polynomial. Since both the polynomial evaluations and the Lagrange coefficients are encoded in the exponent of a group, we must use a bilinear map to multiply. Finally, if the pattern matches the input, a predetermined value is returned as the result of interpolation, which is then used to decrypt the message payload.

One issue is that we cannot give out the BKMPRS wildcard obfuscation directly as a secret key, even though at first glance this seems like the ideal solution. Since the decryptor has no knowledge of the plaintext – it just holds the ciphertext – it cannot choose the correct obfuscation encodings from the $2 \times n$ table to use, even if it had the correct Lagrange coefficients in the clear. So, we must include a mechanism for *selecting* the correct input encodings from the wildcard obfuscation. Our construction solves this problem by making our secret key the coefficients of a degree-2 polynomial. When this polynomial is evaluated it returns the appropriate underlying polynomial evaluations for the particular input.[1] Then, the Lagrange coefficients can be applied, and the evaluation works correctly. We give the details of our scheme in Section 5.2.1.

5.0.1 Related Work

Patranabis and Mukhopadhyay very recently gave function-private public-key constructions for identity-based encryption, subspace-membership encryption, and hidden-vector encryption [PM18]. These constructions are shown secure using DDH-like assumptions, and are for the weakest possible min-entropy requirement on predicates. In particular, their setting of hidden-vector encryption is very similar to our own. Hidden-vector encryption produces secret keys of patterns

[1] These polynomials could conceivably be degree-1, since they must only have 2 points fixed – namely the BKMPRS encodings corresponding to the possible values of the ith input bit. However, this allows an attack on the ciphertext since one of the values will be adversarially known, allowing an attacker to divide off the known values and compare for equality.

consisting of field elements or wildcards. These secret keys match against ciphertexts that are vectors of field elements. Our scheme hides the location of the wildcards, whereas in PM18, the locations of wildcards are public. Furthermore, PM18 does not directly support binary strings. Each non-wildcard element must be a value from a finite field $\mathbb{F}_q$ where q is a prime of size at least $\omega(\log \lambda)$. This limits the possible locations of wildcards, since each non-wildcard part of a pattern must be of length $\omega(\log \lambda)$. This also requires the wildcards to consume $\omega(\log \lambda)$ chunks of the input vectors.

5.1 Preliminaries

Let λ denote the security parameter. We let $\mathbb{G}$ denote a group of prime order p, where we implicitly assume that p is $\Theta(\lambda)$ bits long. For $n \in \mathbb{N}$ we let $[n]$ denote the ordered list of integers $[1, 2, \ldots, n]$, and for $m < n \in \mathbb{N}$ we let $[m, n]$ denote the ordered list of integers $[m, m+1, \ldots, n]$.

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be distinct groups of order p with generators g_1, g_2, g_T, respectively, such that there exists an efficiently computable bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ such that (1) $e(g_1, g_2) = g_T$, and (2) for all $a, b \in \mathbb{Z}_p$, $e(g_1^a, g_2^b) = g_T^{ab}$. We use the bracket notation of Escala et al. [EHK+13] throughout, and thus we can rewrite the second bilinear map property as $e([\![a]\!]_1, [\![b]\!]_2) = [\![ab]\!]_T$.

We use the notation '\$' to denote a value sampled uniformly at random from some set, where the set will be clear from the context.

Let $P = \{0, 1, ?\}^*$ denote the set of all wildcard patterns, and let $P_n = \{0, 1, ?\}^n$ denote the set of wildcard patterns of length n. We say a string $s \in \{0, 1\}^n$ *matches* pattern $\mathsf{pat} \in P_n$ if for all $i \in [n]$ it holds that either $s[i] = \mathsf{pat}[i]$ or $\mathsf{pat}[i] = ?$, and use the notation $s \in \mathsf{pat}$ to denote that s matches pat. Let $\mathsf{Match}_n : P_n \times \{0, 1\}^n \to \{0, 1\}$ be a polynomial-time algorithm that outputs 1 if and only if the input $m \in \{0, 1\}^n$ matches pattern $\mathsf{pat} \in P_n$.

5.1.1 Security Notions

Definition 5.1.1 (Distributional Virtual Black-Box)**.** *Let* $\mathcal{D} = \{\mathcal{D}_n\}_{n\in\mathbb{N}}$ *be an ensemble of distributions, where* $\mathcal{D}_n$ *is a distribution over* P_n*. We define the* ***distributional virtual black-box (dist-VBB)*** *advantage of adversary* $\mathcal{A}$ *for obfuscation scheme* Obf *and simulator* $\mathcal{S}$ *as*

$$\mathbf{Adv}^{\mathsf{dist\text{-}VBB}}_{\mathsf{Obf},\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n) \stackrel{\text{def}}{=} \left|\Pr\left[\mathsf{Expt}^{\mathsf{dist\text{-}VBB}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,0)=1\right] - \Pr\left[\mathsf{Expt}^{\mathsf{dist\text{-}VBB}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,1)=1\right]\right|$$

where for $\lambda\in\mathbb{N}$*,* $n\in\mathbb{N}$*, and* $b\in\{0,1\}$*, the experiment* $\mathsf{Expt}^{\mathsf{dist\text{-}VBB}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,b)$ *is defined as:*

$\mathsf{Expt}^{\mathsf{dist\text{-}VBB}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,b)$

return $\mathcal{A}^{\mathcal{O}}(1^\lambda)$
 	
$\underline{\mathcal{O}()}$
if $b=0$:
  $\mathsf{pat} \leftarrow \mathcal{D}_n$
  return $\mathsf{Obf}(1^\lambda,\mathsf{pat})$
else:
  return $\mathcal{S}(1^\lambda,n)$

Let $\mathcal{D} = \{\mathcal{D}_n\}$ be the distribution ensemble where $\mathcal{D}_n$ is the uniform distribution over $\{0,1,?\}^n$ such that $\le \frac{3n}{4}$ entries are wildcards. Bishop et al. [BKM+18] showed an obfuscation scheme Obf for $\mathcal{D}$ in the generic group model such that for all adversaries $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that

$$\mathbf{Adv}^{\mathsf{dist\text{-}VBB}}_{\mathsf{Obf},\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n) \le \frac{(Q+2n)^2}{2^n} + 2^{-0.0613n},$$

where Q is the number of queries made by $\mathcal{A}$ to the generic group oracle.

5.1.1.1 Generic Bilinear Group Oracle

We use the definition of the Generic Bilinear Group Oracle from [KLM+16, Definition 2.7].

Definition 5.1.2 (Generic Bilinear Group Oracle). *A generic bilinear group oracle is a stateful oracle* BG *that responds to queries as follows:*

- *On a query* BG.Setup(1^λ)*, the generic bilinear group oracle will generate two fresh nonces* $\mathsf{pp}, \mathsf{sp} \leftarrow \{0,1\}^\lambda$ *and a prime q (as in the real setup procedure). It outputs* $(\mathsf{pp}, \mathsf{sp}, q)$*. It will store the values generated, initialize an empty table $T \leftarrow \{\}$, and set the internal state so subsequent invocations of* BG.Setup *fail.*

- *On a query* BG.Encode(k, x, i) *where $k \in \{0,1\}^\lambda, x \in \mathbb{Z}_q$ and $i \in \{1, 2, T\}$ (for the "left" group $\mathbb{G}_1$, the "right" group $\mathbb{G}_2$, and the target group $\mathbb{G}_T$), the oracle checks that $k =$* sp *(returning $\perp$ if the check fails). The oracle then generates a fresh nonce $h \leftarrow \{0,1\}^\lambda$, adds the entry $h \mapsto (x, i)$ to the table T, and replies with h.*

- *On a query* BG.Add(k, h_1, h_2) *where $k, h_1, h_2 \in \{0,1\}^\lambda$, the oracle checks that $k =$* pp*, that the handles h_1, h_2 are present in its internal table T, and are mapped to the values (x_1, i_1) and (x_2, i_2), respectively, and where $i_1 = i_2$ (returning $\perp$ otherwise). If the checks pass, the oracle generates a fresh handle $h \leftarrow \{0,1\}^\lambda$, computes $x = x_1 + x_2 \in \mathbb{Z}_q$, adds the entry $h \mapsto (x, i_1)$ to T, and replies with h.*

- *On a query* BG.Pair(k, h_1, h_2) *where $k, h_1, h_2 \in \{0,1\}^\lambda$, the oracle checks that $k =$* pp*, that the handles h_1, h_2 are present in T, and are mapped to values $(x_1, 1)$ and $(x_2, 2)$, respectively (returning $\perp$ otherwise). If the checks pass, the oracle generates a fresh handle $h \leftarrow \{0,1\}^\lambda$, computes $x = x_1 x_2 \in \mathbb{Z}_q$, adds the entry $h \mapsto (x_1, x_2, T)$ to T, and replies with h.*

- *On a query* BG.ZeroTest(k, x) *where $k, x \in \{0,1\}^\lambda$, the oracle checks that $k =$* pp*, that the handle h is present in T, and that h maps to some value (x, i) (returning $\perp$ otherwise). If the checks pass, the oracle returns "zero" if $x = 0 \in \mathbb{Z}_q$ and "non-zero" otherwise.*

Note that the generic bilinear group oracle leaks equality by allowing two encodings to be subtracted and then zero-tested. This is equivalent to generic oracles whose handles are in one-to-one correspondence with values. We also repeat the usage of formal polynomials as in KLM+, and repeat their Remark 2.8 verbatim [KLM+16, Remark 2.8].

Remark 5.1.3 (Oracle Queries Referring to Formal Polynomials)**.** *Although the generic bilinear map oracle is defined formally in terms of "handles" (Definition 5.1.2), it is more conducive to regard each oracle query as referring to a formal query polynomial. The formal variables in this formal query polynomial are specified by the expressions supplied to the* BG.Encode *oracle (as determined by the details of the construction), and the adversary can construct terms that refer to new polynomials by making oracle queries for the generic group operations* BG.Add *and* BG.Pair*. Rather than operating on a "handle," each valid* BG.ZeroTest *query refers to a formal query polynomial, and the result of the query is "zero" if the polynomial evaluates to zero when its variables are instantiated with the joint distribution over their values in $\mathbb{Z}_q$ as generated in the real security game.*

We also require the Schwartz–Zippel lemma [Sch80, Zip79], stated as follows.

Lemma 5.1.4 (Schwartz–Zippel Lemma)**.** *Fix a prime p and let $f \in \mathbb{F}_p[x_1, \dots, x_n]$ be a n-variate polynomial with degree at most d and which is not identically zero. Then,*

$$\Pr[x_1, \dots, x_n \leftarrow \mathbb{F}_p : f(x_1, \dots, x_n) = 0] \leq d/p.$$

5.1.2 Defining Wildcard Encryption

Definition 5.1.5 (Public-Key Wildcard Encryption)**.** *A public-key wildcard encryption scheme $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is a tuple of probabilistic polynomial-time algorithms defined as follows:*

- $\mathsf{Setup}(1^\lambda, n, m) \rightarrow (\mathsf{msk}, \mathsf{pk})$ *: On input security parameter $\lambda \in \mathbb{N}$ (provided in unary), pattern length $n \in \mathbb{N}$, and payload length $m \in \mathbb{N}$, output master secret key* msk *and public key* pk.
- $\mathsf{KeyGen}(\mathsf{msk}, \mathsf{pat}) \rightarrow \mathsf{sk}$ *: On input master secret key* msk *and wildcard pattern $\mathsf{pat} \in P_n$, output secret key* sk.

- $\mathsf{Enc}(\mathsf{pk}, x, y) \to \mathsf{ct}$: *On input public key* pk*, string* $x \in \{0,1\}^n$*, and payload* $y \in \{0,1\}^m$*, output ciphertext* ct*.*

- $\mathsf{Dec}(\mathsf{sk}_{\mathsf{pat}}, \mathsf{ct}) \to \{\perp\} \cup \{0,1\}^m$*: On input secret key* sk *for pattern* pat *and ciphertext* ct*, output either* $\perp$ *or payload* $y \in \{0,1\}^m$*.*

Definition 5.1.6 (Correctness)**.** *A public-key wildcard encryption scheme* Π *is correct if for all* $n, m \in \mathbb{N}$*,* $\mathsf{pat} \in P_n$ *and* $(x, y) \in \{0,1\}^n \times \{0,1\}^m$ *such that* $\mathsf{Match}_n(\mathsf{pat}, x)$*:*

$$\Pr\left[\begin{array}{l}(\mathsf{msk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(1^\lambda, n, m)\\ \mathsf{sk} \leftarrow \mathsf{KeyGen}(\mathsf{msk}, \mathsf{pat})\\ \mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, x, y)\end{array} \;\middle|\; \mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) = y\right] = 1 - \nu(\lambda)$$

where the probability is taken over the internal randomness of the algorithms and $\nu(\cdot)$ *is a negligible function.*

5.1.2.1 Message Privacy

Definition 5.1.7 (Message Privacy)**.** *Let* Π *be a public-key wildcard encryption scheme, and let* $\mathcal{A}$ *be a stateful adversary. We define the message privacy (MP) advantage as*

$$\mathbf{Adv}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n) \stackrel{\text{def}}{=} \left|\Pr\left[\mathsf{Expt}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n, m, 0) = 1\right] - \Pr\left[\mathsf{Expt}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n, m, 1) = 1\right]\right|,$$

where for $\lambda \in \mathbb{N}$*,* $n \in \mathbb{N}$*,* $m \in \mathbb{N}$*, and* $b \in \{0,1\}$*, the experiment* $\mathsf{Expt}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n, m, b)$ *is defined as:*

$\mathsf{Expt}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n, m, b)$
$(\mathsf{msk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(1^\lambda, n, m)$ $(x_0, x_1, y_0, y_1) \leftarrow \mathcal{A}^{\mathsf{KeyGen}(\mathsf{msk},\cdot)}(1^\lambda, \mathsf{pk})$ $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, x_b, y_b)$ **return** $\mathcal{A}^{\mathsf{KeyGen}(\mathsf{msk},\cdot)}(\mathsf{ct})$

where $|x_0| = |x_1| = n$ *and* $|y_0| = |y_1| = m$ *and the patterns input to* KeyGen *are of length* n*.*

We require that $\mathcal{A}$ is admissible *in the following sense: for all* KeyGen *queries* pat *made by* $\mathcal{A}$ *the following holds:*

1. $\mathsf{Match}(\mathsf{pat}, x_0) = \mathsf{Match}(\mathsf{pat}, x_1)$*, and*

2. *if* $\mathsf{Match}(\mathsf{pat}, x_0) = \mathsf{Match}(\mathsf{pat}, x_1) = 1$ *then* $y_0 = y_1$*.*

We say Π is message private if for all $\lambda \in \mathbb{N}$, $n \in \mathbb{N}$, $m \in \mathbb{N}$, and PPT adversaries $\mathcal{A}$, there exists a negligible function $\nu(\cdot)$ such that $\mathbf{Adv}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n, m) \leq \nu(\lambda)$.

5.1.2.2 Function Privacy

Definition 5.1.8 (Function Privacy)**.** *Let Π be a public-key wildcard encryption scheme, let $\mathcal{A}$ be a stateful adversary, and let $\mathcal{S}$ be an explicit* PPT *algorithm simulating* KeyGen*. We define the function privacy advantage for distribution ensemble $\mathcal{D} = \{\mathcal{D}_n\}$ as*

$$\mathbf{Adv}^{\mathsf{FP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda, \mathcal{D}, n, m) \stackrel{\text{def}}{=} \left| \Pr\left[\mathsf{Expt}^{\mathsf{FP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda, \mathcal{D}, n, m, 0) = 1\right] - \Pr\left[\mathsf{Expt}^{\mathsf{FP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda, \mathcal{D}, n, m, 1) = 1\right] \right|,$$

where for $\lambda \in \mathbb{N}$, $n \in \mathbb{N}$, $m \in \mathbb{N}$, and $b \in \{0, 1\}$, the experiment $\mathsf{Expt}^{\mathsf{FP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda, \mathcal{D}, n, m, b)$ is defined as:

$\mathsf{Expt}^{\mathsf{FP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda, \mathcal{D}, n, m, 0)$
$(\mathsf{msk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(1^\lambda, n, m)$ **return** $\mathcal{A}^{\mathcal{O}(\mathsf{msk})}(1^\lambda, \mathsf{pk})$ $\underline{\mathcal{O}(\mathsf{msk})}$ $\mathsf{pat} \leftarrow \mathcal{D}_n$ **return** $\mathsf{KeyGen}(\mathsf{msk}, \mathsf{pat})$

$\mathsf{Expt}^{\mathsf{FP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda, \mathcal{D}, n, m, 1)$
$(\mathsf{msk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(1^\lambda, n, m)$ **return** $\mathcal{A}^{\mathcal{S}}(1^\lambda, \mathsf{pk})$

We say Π is function private for distribution ensemble $\mathcal{D}$ if there exists a simulator $\mathcal{S}$ such that for all $\lambda \in \mathbb{N}$, $n \in \mathbb{N}$, $m \in \mathbb{N}$, and PPT adversaries $\mathcal{A}$, there exists a negligible function $\nu(\cdot)$ such that $\mathbf{Adv}^{\mathsf{FP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda, \mathcal{D}, n, m) \leq \nu(\lambda)$.

Definition 5.1.9 (Enhanced Function Privacy)**.** *Let* Π *be a public-key wildcard encryption scheme, let* $\mathcal{A}$ *be a stateful adversary, and let* $\mathcal{S} = (\mathcal{S}_{\mathsf{KeyGen}}, \mathcal{S}_{\mathsf{Enc}})$ *be an explicit* PPT *algorithm simulating* KeyGen *and* Enc*. We define the enhanced function privacy advantage for distribution ensemble* $\mathcal{D} = \{\mathcal{D}_n\}$ *as*

$$\mathbf{Adv}^{\mathsf{eFP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,m) \stackrel{\text{def}}{=} \left|\Pr\left[\mathsf{Expt}^{\mathsf{eFP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,m,0)=1\right] - \Pr\left[\mathsf{Expt}^{\mathsf{eFP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,m,1)=1\right]\right|,$$

where for $\lambda \in \mathbb{N}$*,* $n \in \mathbb{N}$*,* $m \in \mathbb{N}$*, and* $b \in \{0,1\}$*, the experiment* $\mathsf{Expt}^{\mathsf{eFP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,m,b)$ *is defined as:*

$\mathsf{Expt}^{\mathsf{eFP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,m,0)$
$(\mathsf{msk},\mathsf{pk}) \leftarrow \mathsf{Setup}(1^\lambda,n,m)$ Initialize $j=1$ **return** $\mathcal{A}^{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk}),\mathcal{O}_{\mathsf{Enc}}(\mathsf{pk},\cdot,\cdot)}(1^\lambda,\mathsf{pk})$ $\underline{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk})}$ $\mathsf{pat}_j \leftarrow \mathcal{D}_n$ $j{+}{+}$ **return** $\mathsf{KeyGen}(\mathsf{msk},\mathsf{pat}_j)$ $\underline{\mathcal{O}_{\mathsf{Enc}}(\mathsf{pk},j,y)}$ $x \leftarrow \{0,1\}^n$ such that $\mathsf{Match}(\mathsf{pat}_j,x)=1$ **return** $\mathsf{Enc}(\mathsf{pk},x,y)$

$\mathsf{Expt}^{\mathsf{eFP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,m,1)$
$(\mathsf{msk},\mathsf{pk}) \leftarrow \mathsf{Setup}(1^\lambda,n,m)$ **return** $\mathcal{A}^{\mathcal{S}_{\mathsf{KeyGen}}(),\mathcal{S}_{\mathsf{Enc}}(\cdot,\cdot)}(1^\lambda,\mathsf{pk})$

We say Π is enhanced function private for distribution ensemble $\mathcal{D}$ if there exists a simulator $\mathcal{S}$ such that for all $\lambda \in \mathbb{N}$, $n \in \mathbb{N}$, $m \in \mathbb{N}$, and PPT adversaries $\mathcal{A}$, there exists a negligible function $\nu(\cdot)$ such that $\mathbf{Adv}^{\mathsf{eFP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda,\mathcal{D},n,m) \leq \nu(\lambda)$.

$\mathsf{Setup}(1^\lambda, n)$

for $i \in [n]$:
  $r_{i,0}, r_{i,1} \leftarrow \mathbb{Z}_p$
$\alpha \leftarrow \mathbb{Z}_p$
$\mathsf{msk} := (\{r_{i,0}, r_{i,1}\}_{i\in[n]}, \alpha)$
$\mathsf{pk} := (\{[\![r_{i,0}]\!]_1, [\![r_{i,0}^2]\!]_1, [\![r_{i,1}]\!]_1, [\![r_{i,1}^2]\!]_1\}_{i\in[n]}, [\![\alpha]\!]_T)$
return $(\mathsf{msk}, \mathsf{pk})$

$\mathsf{KeyGen}(\mathsf{msk}, \mathsf{pat})$

parse $\mathsf{msk} = (\{r_{i,0}, r_{i,1}\}_{i\in[n]}, \alpha)$
for $i \in [n-1]$:
  $c_i \leftarrow \mathbb{Z}_p$
$F(x) := \alpha + \sum_{i\in[n-1]} c_i x^i$
for $i \in [n]$:
  for $b \in \{0,1\}$:
    if $\mathsf{pat}[i] \in \{b, ?\}$:
      $q_{i,b} := F(2i+b)$
    else:
      $q_{i,b} \leftarrow \mathbb{Z}_p$
  $V_i := \begin{bmatrix} r_{i,0}^2 & r_{i,0} & 1 \\ r_{i,1}^2 & r_{i,1} & 1 \\ 0 & 0 & 1 \end{bmatrix}$
  $\vec{a}_i := V_i^{-1} \cdot \begin{bmatrix} q_{i,0} & q_{i,1} & 0 \end{bmatrix}^T$
return $\{[\![a_{i,2}]\!]_2, [\![a_{i,1}]\!]_2\}_{i\in[n]}$

$\mathsf{Enc}(\mathsf{pk}, x, y)$

parse $\mathsf{pk} = (\{[\![r_{i,b}]\!]_1, [\![r_{i,b}^2]\!]_1\}_{i\in[n], b\in\{0,1\}}, [\![\alpha]\!]_T)$
for $i \in [n]$:
  $L_i := \prod_{j\neq i\in[n]} \frac{-2i-x_j}{2i+x_i-2i-x_j}$
$\beta \leftarrow \mathbb{Z}_p$
$p := H([\![\alpha]\!]_T^\beta) \oplus y$
return $(\{[\![r_{i,x_i}^2]\!]_1^{\beta L_i}, [\![r_{i,x_i}]\!]_1^{\beta L_i}\}_{i\in[n]}, p)$

$\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$

parse $\mathsf{sk} = \{[\![a_{i,2}]\!]_2, [\![a_{i,1}]\!]_2\}_{i\in[n]}$
parse $\mathsf{ct} = (\{[\![r_{i,x_i}^2]\!]_1^{\beta L_i}, [\![r_{i,x_i}]\!]_1^{\beta L_i}\}_{i\in[n]}, p)$
$z := \prod_{i\in[n]} e([\![\beta L_i r_{i,x_i}^2]\!]_1, [\![a_{i,2}]\!]_2) \cdot e([\![\beta L_i r_{i,x_i}]\!]_1, [\![a_{i,1}]\!]_2)$
return $p \oplus H(z)$

Figure 5.1: Our public-key function-private wildcard encryption scheme. $H\colon \mathbb{G}_T \rightarrow \{0,1\}^m$ is a random oracle taking as input an element of $\mathbb{G}_T$ and producing a bit string of length m.

5.2 Wildcard Encryption

5.2.1 Construction

We define our wildcard encryption scheme precisely in Figure 5.1. In more depth, our scheme works as follows. First, the master secret key contains the $r_{i,b}$ values which allows KeyGen to program the degree-2 polynomials to return the correct underlying polynomial evaluations of F. The public key contains encodings of both $r_{i,b}$ and $r_{i,b}^2$, which provides oblivious evaluation of the degree-2 polynomials. The public key allows Enc to choose the right inputs to the degree-2 polynomials in order to obtain the underlying polynomial evaluations of F for a particular message during decryption.

The master secret key also contains the secret value α. This α is used along with β in the target group as an input to H to mask the payload. In order to allow the decryptor to recover this mask, we have two mechanisms. Firstly, F is programmed with $F(0) = \alpha$. Then a matching pattern evaluation will produce an encoding of α during decryption. Secondly, each Lagrange coefficient L_i comes multiplied by β, which scales up each polynomial evaluation of F by β. Subsequently in decryption, this causes us to interpolate the polynomial $F'(x) = \beta \cdot F(x)$. Since $F(0) = \alpha$, we have that $F'(0) = \alpha\beta$. Finally since the payload is masked by $H([\![\alpha\beta]\!]_T)$, the payload message can be recovered.

There is some cleverness in how the secret keys are constructed. At root, each secret key contains a degree-$(n-1)$ polynomial F which is programmed with $F(0) = \alpha$. We then follow BKMPRS and encode the input pattern pat into the $q_{i,b}$ values – each $q_{i,b}$ either random or an evaluation of F depending whether that particular i, b combination represents a match in pat. We use the $q_{i,b}$ values to create the degree-2 polynomial, whose coefficients will be encoded in the secret key. We compute these coefficients by using the inverted Vandermonde matrix on points $(r_{i,0}, q_{i,0}), (r_{i,1}, q_{i,1})$, and $(0, 0)$.[2] Finally, during decryption, the degree-2 polynomials will be evaluated with either $r_{i,0}$ or $r_{i,1}$ depending on the bits of the ciphertext (which determine

[2]We include $(0, 0)$ in order to force this polynomial to be degree-2 but have a 0 constant coefficient, avoiding attacks on message privacy while keeping the secret key size to $2n$ encodings.

which public key encodings to use in encryption), returning the appropriate $q_{i,b}$'s for Lagrange interpolation.

Theorem 5.2.1 (Correctness)**.** *Let* Π *be a public-key wildcard encryption scheme as defined in Figure 5.1. Then* Π *is correct for all* $n \in \mathbb{N}$, $\mathsf{pat} \in P_n$, *and* $(x, y) \in \{0,1\}^n \times \{0,1\}^m$ *where* $\mathsf{Match}_n(\mathsf{pat}, x)$.

Proof. In Dec we have the product of many pairings of ciphertext elements with secret key elements. These pairings serve to simultaneously evaluate the degree-2 polynomials as well as multiply the Lagrange coefficients with the resulting polynomial evaluations of F. Since x matches pat, we require that the final value z must equal $[\![\alpha\beta]\!]_T$. This allows recovery of the payload message. The decryption computation results in

$$\begin{aligned} z &= \prod_{i \in [n]} e([\![\beta L_i r_{i,x_i}^2]\!]_1, [\![a_{i,2}]\!]_2) \cdot e([\![\beta L_i r_{i,x_i}]\!]_1, [\![a_{i,1}]\!]_2) \\ &= [\![\beta \sum_{i \in [n]} L_i (r_{i,x_i}^2 a_{i,2} + r_{i,x_i} a_{i,1})]\!]_T \\ &= [\![\beta \sum_{i \in [n]} L_i q_{i,x_i}]\!]_T. \end{aligned}$$

By construction of KeyGen, we have that for each $i \in [n]$ that $q_{i,x_i} = F(2i + x_i)$. Therefore each L_i is paired with the appropriate $F(2i + x_i)$ value for interpolation of $F(0) = \alpha$. Then we have that

$$[\![\beta \sum_{i \in [n]} L_i q_{i,x_i}]\!]_T = [\![\beta \sum_{i \in [n]} L_i F(2i + x_i)]\!]_T = [\![\alpha\beta]\!]_T,$$

allowing recovery of the payload y via

$$p \oplus H([\![\alpha\beta]\!]_T) = y \oplus H([\![\alpha\beta]\!]_T) \oplus H([\![\alpha\beta]\!]_T) = y.$$

□

5.2.2 Security

We first state and prove Lemma 5.2.2, which will be crucial for proving message privacy (Section 5.2.2.2) and enhanced function privacy (Section 5.2.2.3).

5.2.2.1 Security Lemma

Lemma 5.2.2 roughly states that a generic adversary trying to learn anything non-trivial about a particular ciphertext can only do so by honestly following the decryption procedure. That is, it must honestly pair the $2n$ group elements of the ciphertext with the $2n$ group elements of a particular secret key as dictated by the decryption procedure defined in Figure 5.1.

This holds since an adversary can only learn information in the generic bilinear group model by canceling out the underlying randomness, without knowing the random values themselves. In particular, Lemma 5.2.2 concerns the $r_{i,b}$'s drawn during Setup, and the c_i's and $q_{i,b}$'s drawn during KeyGen. These random values can only be cancelled out honestly by the adversary. The only other randomness used in the scheme is the ciphertext blind β, which is drawn during Enc. Each element the adversary can manipulate is multiplied by β, so it can simply be factored out. This will be made formal when we show message privacy and enhanced function privacy.

Towards defining the lemma, consider a "deterministic" encryption (without β) of an attribute $x \in \{0,1\}^n$. This encryption has the form

$$\{\, L_i(x) r_{i,x_i}, L_i(x) r_{i,x_i}^2 \,\}_{i\in[n]},$$

where $L_i(x)$ is the ith Lagrange coefficient corresponding to x. Also consider a set of J secret keys $\{a_{i,2}^{(j)}, a_{i,1}^{(j)}\}_{i\in[n],j\in[J]}$ for patterns $\{\mathsf{pat}^{(j)}\}_{j\in[J]}$. The adversary can pair arbitrary elements from the ciphertext with arbitrary elements from the secret key. It can also lift ciphertext elements to the target group by paring them with $[\![1]\!]_2$. In this lemma, we do not consider terms that the adversary obtains by lifting secret key elements to the target group. Then, the following is a

polynomial for terms that the adversary can obtain via bilinear group operations:

$$\sum_{\substack{i\in[n],i'\in[n],j\in[J]\\ d\in\{1,2\},e\in\{1,2\}}} \gamma_{i,i',j,d,e} L_i(x) r_{i,x_i}^d a_{i',e,j} + \sum_{i\in[n],d\in\{1,2\}} \eta_{i,d} L_i(x) r_{i,x_i}^d + \kappa. \tag{5.1}$$

Here the γ, η, and κ terms are fixed coefficients which are chosen by the adversary with no knowledge of the underlying randomness. Recall that the $\{r_{i,b}\}_{i\in[n],b\in\{0,1\}}$ terms are each sampled uniformly at random, and the $\{a_{i,1,j}, a_{i,2,j}\}_{i\in[n],j\in[J]}$ terms are secret key elements generated using the random coefficients of the polynomial $\{c_{n-1}^{(j)}, \ldots, c_1^{(j)}\}_{j\in[J]}$ for the jth secret key, and the random values $\{q_{i,b}^{(j)} \mid i, b \text{ where } \mathsf{pat}^{(j)}[i] = 1 - b\}$ which are sampled for the jth secret key. Finally, let

$$J' := \{j \mid \mathsf{Match}(\mathsf{pat}^{(j)}, x) = 1, j \in [J]\}$$

be the set of patterns that x matches.

Lemma 5.2.2. *Let $x \in \{0,1\}^n$ be an attribute, encrypted as*

$$\{\, L_i(x) r_{i,x_i}, L_i(x) r_{i,x_i}^2 \,\}_{i\in[n]}.$$

Then, for Polynomial 5.1 to evaluate to zero with probability greater than $\frac{6n+2}{p}$ over the randomness of the terms $\{r_{i,b}\}_{i\in[n],b\in\{0,1\}}$, $\{c_i^{(j)}\}_{i\in[n],j\in[J]}$, and $\{q_{i,b}^{(j)}\}_{i\in[n],b\in\{0,1\},j\in[J]}$, the following constraints on the coefficients must hold:

1. $\eta_{i,d} = 0 \ \forall i \in [n], d \in \{1,2\}$,
2. $\gamma_{i,i',j,d,e} = 0 \ \forall i \neq i'$,
3. $\gamma_{i,i',j,d,e} = 0 \ \forall d \neq e$,
4. $\gamma_{i,i',j,d,e} = 0 \ \forall j \notin [J']$,
5. $\gamma_{1,1,j,1,1} = \gamma_{1,1,j,2,2} = \ldots = \gamma_{n,n,j,1,1} = \gamma_{n,n,j,2,2} \ \forall j \in [J']$.

Finally, for $j \in [J']$ let k_j to be the value defined by the equality in Item 5. Then,

6. $\kappa = -\sum_{j \in [J']} k_j$.

Furthermore, any setting of coefficients that meets the above constraints makes Polynomial 5.1 evaluate to zero with probability 1.

In words, Item 1 shows that the adversary cannot cancel out the $r_{i,b}$ randomness in a ciphertext without first pairing the ciphertext elements with other elements of the scheme. Item 2 and Item 3 show that the adversary cannot hope to cancel out all randomness via dishonest pairings between ciphertext and secret key elements. Item 4 shows that the adversary cannot hope to cancel out all underlying randomness by pairing its ciphertext with a secret key for a pattern for which the underlying attribute does not match. Finally, Item 5 and Item 6 show that the adversary must sum the results of honestly pairing the ciphertext and secret key elements before introducing scaling factors. An important note which will be used in the following proofs is that the coefficients given by this lemma are independent of the choice of attribute x.

Proof. We begin by turning the underlying randomness in the expression into formal variables, and arguing via the Schwartz–Zippel lemma (Definition 5.1.4) that except with very low probability, the chosen coefficients must cancel out these formal variables in order to make Polynomial 5.1 go to zero. Let $\mathcal{R} := \{\mathbf{r}_{i,0}, \mathbf{r}_{i,1}\}_{i \in [n]}$ be a set of $2n$ formal variables corresponding to the master secret key. For a given pattern $\mathsf{pat}^{(j)}$, let $\mathcal{C}^{(j)} := \{\mathbf{c}_i^{(j)}\}_{i \in [n-1]}$ be the set of formal variables replacing the random polynomial coefficients, and let

$$\mathcal{Q}^{(j)} := \{\mathbf{q}_{i,b}^{(j)} \mid \mathsf{pat}_j[i] = 1 - b\}_{i \in [n], b \in \{0,1\}}$$

be the set of formal variables replacing the remaining randomly drawn values in KeyGen. Now we can write the result of computing KeyGen for pattern pat_j as a set of rational functions over the underlying formal variables:

$$\left\{ \widehat{a}_{i,2}^{(j)} := a_{i,2}^{(j)}(\mathcal{R}, \mathcal{C}^{(j)}, \mathcal{Q}^{(j)}), \, \widehat{a}_{i,1}^{(j)} := a_{i,1}^{(j)}(\mathcal{R}, \mathcal{C}^{(j)}, \mathcal{Q}^{(j)}) \right\}_{i \in [n]}.$$

Now Polynomial 5.1 can be written as

$$\sum_{\substack{i\in[n],i'\in[n],j\in[J]\\ d\in\{1,2\},e\in\{1,2\}}} \gamma_{i,i',j,d,e} L_i(x)\mathbf{r}_{i,x_i}^d \widehat{a}_{i',e}^{(j)} + \sum_{i\in[n],d\in\{1,2\}} \eta_{i,d} L_i(x)\mathbf{r}_{i,x_i}^d + \kappa. \tag{5.2}$$

In the following, we will extensively use the substitution

$$\begin{aligned}\begin{bmatrix}\widehat{a}_{i',2}^{(j)}\\ \widehat{a}_{i',1}^{(j)}\end{bmatrix} &= \begin{bmatrix}\mathbf{r}_{i',1}^2 & \mathbf{r}_{i',1}\\ \mathbf{r}_{i',0}^2 & \mathbf{r}_{i',0}\end{bmatrix}^{-1}\begin{bmatrix}q_{i',1}^{(j)}\\ q_{i',0}^{(j)}\end{bmatrix}\\ &= \frac{1}{\mathbf{r}_{i',1}^2\mathbf{r}_{i',0} - \mathbf{r}_{i',0}^2\mathbf{r}_{i',1}}\begin{bmatrix}q_{i',1}^{(j)}\mathbf{r}_{i',0} - q_{i',0}^{(j)}\mathbf{r}_{i',1}\\ -q_{i',1}^{(j)}\mathbf{r}_{i',0}^2 + q_{i',0}^{(j)}\mathbf{r}_{i',1}^2\end{bmatrix},\end{aligned}$$

which holds for all $j \in [J]$. Applying this substitution in Polynomial 5.2 yields a rational function over the $\mathbf{r}_{i,b}$ variables. As it is easier to reason about polynomials, we multiply by

$$\mathbf{R} := \prod_{i'\in[n]} (\mathbf{r}_{i',1}^2\mathbf{r}_{i',0} - \mathbf{r}_{i',0}^2\mathbf{r}_{i',1})$$

to clear the denominators. This results in the polynomial

$$\sum_{\substack{i\in[n],i'\in[n],j\in[J],\\ d\in\{1,2\},e\in\{1,2\}}} \gamma_{i,i',j,d,e} L_i(x)\mathbf{r}_{i,x_i}^d \mathbf{R}\widehat{a}_{i',e}^{(j)} + \sum_{i\in[n],d\in\{1,2\}} \eta_{i,d} L_i(x)\mathbf{r}_{i,x_i}^d\mathbf{R} + \kappa\mathbf{R} \tag{5.3}$$

where

$$\begin{aligned}\mathbf{R}\widehat{a}_{i',1}^{(j)} &= (-q_{i',1}^{(j)}\mathbf{r}_{i',0}^2 + q_{i',0}^{(j)}\mathbf{r}_{i',1}^2) \prod_{i''\neq i'} (\mathbf{r}_{i'',1}^2\mathbf{r}_{i'',0} - \mathbf{r}_{i'',0}^2\mathbf{r}_{i'',1}),\\ \mathbf{R}\widehat{a}_{i',2}^{(j)} &= (q_{i',1}^{(j)}\mathbf{r}_{i',0} - q_{i',0}^{(j)}\mathbf{r}_{i',1}) \prod_{i''\neq i'} (\mathbf{r}_{i'',1}^2\mathbf{r}_{i'',0} - \mathbf{r}_{i'',0}^2\mathbf{r}_{i'',1}).\end{aligned}$$

We now determine the probability that Polynomial 5.3 equals zero. Observe that Polynomial 5.3 has degree $3n + 2$ over the $\mathcal{R}, \mathcal{C}$, and $\mathcal{Q}$ formal variables. To see why, note that each

term has at most degree $3n+1$ over the $\mathcal{R}$ variables and is linear in the $q_{i,b}^{(j)}$ terms, each of which is either a single formal variable $\mathbf{q}_{i,b}^{(j)}$ or a linear combination of the $\mathcal{C}$ variables.

Now if the coefficients $\gamma, \eta,$ and κ are not fixed so that Polynomial 5.3 is identically zero over its formal variables, then by Schwartz-Zippel, it will evaluate to zero with probability at most $\frac{3n+2}{p}$ when its formal variables are instantiated with random values from $\mathbb{Z}_p$. Similarly by Schwartz-Zippel, $\mathbf{R}$ only evaluates to zero with probability $\frac{3n}{p}$ when the $\mathcal{R}$ variables are instantiated. Thus, if the coefficients are not set so that Polynomial 5.3 is identically zero over its formal variables, then Polynomial 5.1 goes to zero with probability at most $\frac{6n+2}{p}$. We proceed by determining which setting of $\gamma, \eta,$ and κ result in Polynomial 5.3 being an identically zero polynomial.

Lemma 5.2.3. *If Polynomial 5.3 is identically zero, then for all $i \in [n], d \in \{1,2\}$, $\eta_{i,d} = 0$.*

Proof. The monomial $\mathbf{r}_{i,x_i}^d \prod_{i' \in [n]} \mathbf{r}_{i',1}^2 \mathbf{r}_{i',0}$ in Polynomial 5.3 only appears in the second sum due to the multiplication of $\mathbf{r}_{i,x_i}^2$ and $\mathbf{R}$. Its coefficient, which must be zero, is $\eta_{i,d} L_i(x)$, and Lagrange coefficients are always non-zero. □

Given Lemma 5.2.3, we eliminate the second term in Polynomial 5.2 to obtain

$$\sum_{\substack{i \in [n], i' \in [n], j \in [J], \\ d \in \{1,2\}, e \in \{1,2\}}} \gamma_{i,i',j,d,e} L_i(x) \mathbf{r}_{i,x_i}^d \mathbf{R} \widehat{a}_{i',e}^{(j)} + \kappa \mathbf{R}. \tag{5.4}$$

Lemma 5.2.4. *If Polynomial 5.4 is identically zero, then for all j, d, e, $\gamma_{i,i',j,d,e} = 0$ whenever $i \neq i'$.*

This lemma corresponds to the "mismatched" pairing of elements $[\![L_i(x) r_{i,x_i}^d]\!]_1$ from the ciphertext and elements $[\![a_{i',e}^{(j)}]\!]_2$ from the secret key, for different bits. We show that this pairing produces unique monomials which can not be canceled out by other terms.

Proof. Fix some choice of i, i' satisfying $i \neq i'$.

- First we claim $\gamma_{i,i',j,1,1} = 0$ for all $j \in [J]$. To prove this, we consider the following monomial of $\mathcal{R}$ variables:

$$\mathbf{r}_{i,x_i}\mathbf{r}_{i',0}^2 \prod_{i'' \neq i'} \mathbf{r}_{i'',1}^2 \mathbf{r}_{i'',0}.$$

In the summation in Polynomial 5.4, this monomial only appears once for each $j \in [J]$. Namely it appears as the first term of the expansion of $\mathbf{r}_{i,x_i}\mathbf{R}\widehat{a}_{i',1}^{(j)}$. Its coefficient is

$$\sum_{j \in [J]} -q_{i',1}^{(j)} \gamma_{i,i',j,1,1} L_i(x).$$

If we fix i' and consider $\{q_{i',1}^{(j)}\}_{j \in [J]}$, these each involve a distinct $\mathbf{q}$ or $\mathbf{c}$ formal variable, as each arises from a different secret key. Thus, each $-\gamma_{i,i',j,1,1}L_i(x)$ is the coefficient of a distinct monomial over the formal variables $\mathcal{R}$, $\{\mathcal{C}^{(j)}\}_{j \in J}$, and $\{\mathcal{Q}^{(j)}\}_{j \in J}$, so each $-\gamma_{i,i',j,1,1}$ must be zero.

- $\gamma_{i,i',j,1,2} = 0$ for all $j \in [J]$. This follows from applying the above argument but to coefficients of monomials including

$$\mathbf{r}_{i,x_i}\mathbf{r}_{i',0} \prod_{i'' \neq i'} \mathbf{r}_{i'',1}^2 \mathbf{r}_{i'',0}.$$

- $\gamma_{i,i',j,2,1} = 0$ for all $j \in [J]$. Same argument as above, but with monomials that include

$$\mathbf{r}_{i,x_i}^2\mathbf{r}_{i',0}^2 \prod_{i'' \neq i'} \mathbf{r}_{i'',1}^2 \mathbf{r}_{i'',0}.$$

- $\gamma_{i,i',j,2,2} = 0$ for all $j \in [J]$. Same argument as above, but with monomials that include

$$\mathbf{r}_{i,x_i}^2\mathbf{r}_{i',0} \prod_{i'' \neq i'} \mathbf{r}_{i'',1}^2 \mathbf{r}_{i'',0}.$$

This covers all possible $\gamma_{i,i',j,d,e}$ coefficients for each $i \neq i'$, so the claim follows. $\square$

Applying Lemma 5.2.4 allows us to further simplify Polynomial 5.4. Since all the $\gamma_{i,i',j,d,e}$ terms are 0 when $i \neq i'$, we drop the i' subscript, letting $\gamma_{i,j,d,e}$ stand in for $\gamma_{i,i,j,d,e}$. So Polynomial 5.4 becomes

$$\sum_{\substack{i\in[n],j\in[J],\\ d\in\{1,2\},e\in\{1,2\}}} \gamma_{i,j,d,e} L_i(x)\mathbf{r}^d_{i,x_i}\mathbf{R}\widehat{a}^{(j)}_{i,e} + \kappa\mathbf{R} \tag{5.5}$$

We expand Polynomial 5.5 by substituting in the values of $\mathbf{R}\widehat{a}^{(j)}_{i,1}$ and $\mathbf{R}\widehat{a}^{(j)}_{i,2}$, giving

$$\begin{aligned} &\sum_{i\in[n],j\in[J]} \gamma_{i,j,1,1} L_i(x)\mathbf{r}_{i,x_i}(-q^{(j)}_{i,1}\mathbf{r}^2_{i,0} + q^{(j)}_{i,0}\mathbf{r}^2_{i,1})\prod_{i'\neq i}(\mathbf{r}^2_{i',1}\mathbf{r}_{i',0} - \mathbf{r}^2_{i',0}\mathbf{r}_{i',1}) \\ &+ \sum_{i\in[n],j\in[J]} \gamma_{i,j,2,1} L_i(x)\mathbf{r}^2_{i,x_i}(-q^{(j)}_{i,1}\mathbf{r}^2_{i,0} + q^{(j)}_{i,0}\mathbf{r}^2_{i,1})\prod_{i'\neq i}(\mathbf{r}^2_{i',1}\mathbf{r}_{i',0} - \mathbf{r}^2_{i',0}\mathbf{r}_{i',1}) \\ &+ \sum_{i\in[n],j\in[J]} \gamma_{i,j,1,2} L_i(x)\mathbf{r}_{i,x_i}(q^{(j)}_{i,1}\mathbf{r}_{i,0} - q^{(j)}_{i,0}\mathbf{r}_{i,1})\prod_{i'\neq i}(\mathbf{r}^2_{i',1}\mathbf{r}_{i',0} - \mathbf{r}^2_{i',0}\mathbf{r}_{i',1}) \\ &+ \sum_{i\in[n],j\in[J]} \gamma_{i,j,2,2} L_i(x)\mathbf{r}^2_{i,x_i}(q^{(j)}_{i,1}\mathbf{r}_{i,0} - q^{(j)}_{i,0}\mathbf{r}_{i,1})\prod_{i'\neq i}(\mathbf{r}^2_{i',1}\mathbf{r}_{i',0} - \mathbf{r}^2_{i',0}\mathbf{r}_{i',1}) \\ &+ \kappa\mathbf{R}. \end{aligned} \tag{5.6}$$

Lemma 5.2.5. *If Polynomial 5.6 is identically zero, then for all* $i \in [n], j \in [J]$, $\gamma_{i,j,2,1} = \gamma_{i,j,1,2} = 0$ *and* $\gamma_{i,j,1,1} = \gamma_{i,j,2,2}$.

This lemma reflects the fact that the adversary must use the correct public key values with the correct secret key values, or else result in monomials that cannot be canceled by any other terms.

Proof. This proof follows from the same techniques used for the previous lemmas.

- $\gamma_{i,j,2,1} = 0$ for all i, j. This follows from inspecting the $\mathbf{r}^2_{i,x_i}\mathbf{r}^2_{i,0}\prod_{i'\neq i}\mathbf{r}^2_{i',1}\mathbf{r}_{i',0}$ monomial of $\mathcal{R}$ variables, which has coefficient

$$\sum_{j\in[J]} -q^{(j)}_{i,1}\gamma_{i,j,2,1}L_i(x)$$

and relies on the disjointedness of the formal variables included in the $\{q_{i,1}^{(j)}\}_{j\in[J]}$ terms such as in the proof of Lemma 5.2.4.

- $\gamma_{i,j,1,2} = 0$ for all i, j. This follows from inspecting the $\mathbf{r}_{i,x_i}\mathbf{r}_{i,0}\prod_{i'\neq i}\mathbf{r}_{i',1}^2\mathbf{r}_{i',0}$ monomial of $\mathcal{R}$ variables, which has coefficient

$$\sum_{j\in[J]} q_{i,1}^{(j)}\gamma_{i,j,1,2}L_i(x)$$

  and relies on the same argument as above.

- $\gamma_{i,j,1,1} = \gamma_{i,j,2,2}$ for all i, j. This follows from inspecting the $\mathbf{r}_{i,x_i}^3\prod_{i'\neq i}\mathbf{r}_{i',1}^2\mathbf{r}_{i',0}$ monomial of $\mathcal{R}$ variables, which occurs in both the first and fourth terms regardless of the value of x_i. In both cases this monomial has coefficient

$$\sum_{j\in[J]} q_{i,1-x_i}^{(j)}((-1)^{1-x_i}\gamma_{i,j,1,1} + (-1)^{x_i}\gamma_{i,j,2,2})L_i(x)$$

  and relies on the same argument as above.

□

Using Lemma 5.2.5, we define $k_{i,j} := \gamma_{i,j,1,1} = \gamma_{i,j,2,2}$ and simplify Polynomial 5.5 to obtain

$$\mathbf{R}\left(\sum_{i\in[n],j\in[J]} k_{i,j}L_i(x)(\mathbf{r}_{i,x_i}^2 a_{i,2}^{(j)} + \mathbf{r}_{i,x_i}a_{i,1}^{(j)}) + \kappa\right) = \mathbf{R}\left(\sum_{i\in[n],j\in[J]} k_{i,j}L_i(x)q_{i,x_i}^{(j)} + \kappa\right). \quad (5.7)$$

And now since the expression inside the parentheses is a polynomial over the $\{\mathcal{C}^{(j)}\}_{j\in J}$ and $\{\mathcal{Q}^{(j)}\}_{j\in J}$ variables, we drop the $\mathbf{R}$ term and conclude that the following polynomial must be identically zero:

$$\sum_{i\in[n],j\in[J]} k_{i,j}L_i(x)q_{i,x_i}^{(j)} + \kappa \quad (5.8)$$

Lemma 5.2.6. *If Polynomial 5.8 is identically zero, then* $k_{i,j} = 0$ *for all* $j \in [J'], i \notin [n]$.

This lemma shows that secret keys for patterns that don't match x are useless to the adversary.

Proof. For every j considered in the lemma statement, x does not match pat_j, so the corresponding $\sum_{i\in[n]} k_{i,j} L_i(x) q^{(j)}_{i,x_i}$ term involves at least one value of i such that $q^{(j)}_{i,x_i}$ is just the formal variable $\mathbf{q}^{(j)}_{i,x_i}$ (implying that the associated $k_{i,j}$ must be zero). Furthermore, the following holds for at most $n-1$ values of $i \in [n]$:

$$q^{(j)}_{i,x_i} = F^{(j)}(2i + x_i),$$

where $F^{(j)}$ is a degree-$(n-1)$ polynomial with formal coefficients $\{\mathbf{c}^{(j)}_i\}_{i\in[n-1]}$. Now since any set of less than n of $F^{(j)}$'s evaluations are linearly independent over the underlying coefficients, the adversary must set the remaining $k_{i,j}$ to zero. We then conclude that $k_{i,j} = 0$ for all $i \in [n]$. □

We can now restrict the summation in Polynomial 5.8 to only include values $j \in [J']$, and rewrite the $q^{(j)}_{i,x_i}$ values as polynomial evaluations:

$$\sum_{i\in[n],j\in J'} k_{i,j} L_i(x) F^{(j)}(2i + x_i) + \kappa. \tag{5.9}$$

Lemma 5.2.7. *If Polynomial 5.9 is identically zero, then* $k_{1,j} = ... = k_{n,j} := k_j \ \forall j \in [J']$ *and* $\kappa = -\sum_{j\in[J']} k_j$

Proof. We expand each polynomial evaluation $F^{(j)}$ in terms of its coefficients, as

$$F^{(j)}(x) = \mathbf{c}^{(j)}_{n-1} x^{n-1} + \ldots \mathbf{c}^{(j)}_1 x + 1.$$

Plugging this into Polynomial 5.9 yields

$$\begin{aligned}
&\sum_{i\in[n],j\in J'} k_{i,j}L_i(x)F^{(j)}(2i+x_i)+\kappa \\
&=\sum_{i\in[n],j\in J'} k_{i,j}L_i(x)\Big(1+\sum_{\ell\in[n-1]}\mathbf{c}_\ell^{(j)}(2\ell+x_i)^\ell\Big)+\kappa \\
&=\sum_{j\in J',\ell\in[n-1]}\mathbf{c}_\ell^{(j)}\Big(\sum_{i\in[n]}k_{i,j}L_i(x)(2\ell+x_i)^\ell\Big)+\sum_{i\in[n]}k_{i,j}L_i(x)+\kappa.
\end{aligned}$$

Now we solve for the $k_{i,j}$ values that make the coefficient on each $\mathbf{c}_\ell^{(j)}$ term go to zero. For each value of j we have the following system of equations:

$$\begin{bmatrix} L_1^{(x)}(2+x_1)^{n-1} & L_2^{(x)}(4+x_2)^{n-1} & \cdots & L_n^{(x)}(2n+x_n)^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ L_1^{(x)}(2+x_1) & L_2^{(x)}(4+x_2) & \cdots & L_n^{(x)}(2n+x_n) \end{bmatrix}\cdot\begin{bmatrix} k_{1,j} \\ \vdots \\ k_{n,j}\end{bmatrix}=\begin{bmatrix}0\\ \vdots \\ 0\end{bmatrix}.$$

From the correctness of Lagrange interpolation (where row i consists of evaluations of the polynomial $f(y)=y^{n-i}$), one solution (regardless of the setting of b) is $k_{i,j}=1$ for all $i\in[n]$. Also note that regardless of x, this matrix has rank $n-1$ (view it as the transpose of scalar multiples of $n-1$ columns of an $n\times n$ Vandermonde matrix). Therefore, this matrix has the same 1-dimensional kernel, spanned by the all 1's vector, for any attribute string x.

We conclude that $k_{1,j}=k_{2,j}=...=k_{n,j}:=k_j$ for all $j\in Q'$. The resulting expression is

$$\sum_{i\in[n],j\in J'} k_jL_i^{(x_b)}+\kappa=\sum_{j\in J'}k_j+\kappa$$

so κ must be set to $-\sum_{j\in J'}k_j$, which completes the proof.

□

Lemma 5.2.2 follows by combining the above lemmas. □

5.2.2.2 Message Privacy

In this section we prove Theorem 5.2.8, which states that our wildcard encryption construction Π achieves message privacy in the generic group model. Our proof strategy involves considering all possible combinations of group elements the adversary might form. A slight annoyance for this strategy comes from the fact that in our construction Π, the last component of the ciphertext is $p := H([\![\beta]\!]_1) \oplus y$, which is not a group element. To get around this, our proof steps will be written with a modified scheme Π' in mind. Π' is a simple modification of Π where $p := [\![\beta]\!]_1$, and $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$ simply outputs a bit indicating if $p = z$. Intuitively, Π' is a weakened scheme where secret keys allow the user to learn whether or not the encoded attribute matched the pattern but nothing else.

Throughout the course of the proof, we will explain how indistinguishability in the message privacy experiment for Π' implies the same notion for the full construction Π. We do this by modeling H as a random oracle whose domain is $\mathbb{G}_1$. Since the generic adversary can only specify handles to group elements, it requests evaluations of H from the model by submitting a handle. The model checks if the handle corresponds to a group element it has previously seen, and if so, it queries a random oracle H (which it can implement itself) and returns the result to the adversary.

Theorem 5.2.8. *Let Π be the wildcard encryption scheme. Then for any PPT adversary $\mathcal{A}$ that performs K zero test queries in the generic group model,* $\mathsf{Adv}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n) \leq \frac{(12n+6)K}{p}$.

Proof. We consider the behavior of some PPT adversary $\mathcal{A}$ interacting with the alternate scheme Π' in the generic group model. We want to show that $\mathcal{A}$'s view when $b = 0$ is indistinguishable from its view when $b = 1$. $\mathcal{A}$ obtains handles to the public key, the group elements output by $\mathsf{Enc}(\mathsf{pk}, x_b, y_b)$, and the group elements resulting from its J secret key queries (indexed from

$1, \ldots, J$):

$$\begin{aligned}
\textit{Public key: } & [\![1]\!]_1, [\![1]\!]_2, [\![r_{1,0}^2]\!]_1, [\![r_{1,0}]\!]_1, [\![r_{1,1}^2]\!]_1, [\![r_{1,1}]\!]_1, \ldots, [\![r_{n,0}^2]\!]_1, [\![r_{n,0}]\!]_1, [\![r_{n,1}^2]\!]_1, [\![r_{n,1}]\!]_1, \\
\textit{Encryption of } (x_b, y_b)\textit{: } & [\![\beta_b L_1^{(b)} r_{1,x_{b,1}}^2]\!]_1, [\![\beta_b L_1^{(b)} r_{1,x_{b,1}}]\!]_1, \ldots, [\![\beta_b L_n^{(b)} r_{n,x_{b,n}}^2]\!]_1, [\![\beta_b L_n^{(b)} r_{n,x_{b,n}}]\!]_1, [\![\beta_b]\!]_1, \\
\textit{Secret keys: } & \{[\![a_{1,2}^{(j)}]\!]_2, [\![a_{1,1}^{(j)}]\!]_2, \ldots, [\![a_{n,2}^{(j)}]\!]_2, [\![a_{n,1}^{(j)}]\!]_2\}_{j \in [J]}.
\end{aligned}$$

Note that $\mathcal{A}$ can send any element in $\mathbb{G}_1$ or $\mathbb{G}_2$ to $\mathbb{G}_T$ by pairing with $[\![1]\!]_2$ or $[\![1]\!]_1$ respectively, so we restrict attention to $\mathbb{G}_T$.

In the most general form possible, any encoding in $\mathbb{G}_T$ that $\mathcal{A}$ computes is of the form

$$\begin{aligned}
& \sum_{\substack{i \in [n], i' \in [n], j \in [J] \\ d \in \{1,2\}, e \in \{1,2\}}} \gamma_{i,i',j,d,e} \beta_b L_i(x_b) r_{i,x_{b,i}}^d a_{i',e}^{(j)} + \sum_{\substack{i \in [n], i' \in [n], j \in [J] \\ d \in \{1,2\}, e \in \{1,2\}, f \in \{0,1\}}} \delta_{i,i',j,d,e,f} r_{i,f}^d a_{i',e}^{(j)} \\
& + \sum_{i \in [n], d \in \{1,2\}} \eta_{i,d} \beta_b L_i(x_b) r_{i,x_{b,i}}^d + \sum_{i \in [n], d \in \{1,2\}, f \in \{0,1\}} \mu_{i,d,f} r_{i,f}^d + \sum_{i' \in [n], j \in [J], e \in \{1,2\}} \nu_{i',j,e} a_{i',e}^{(j)} \\
& + \kappa_1 \beta_b + \kappa_0
\end{aligned} \tag{5.10}$$

for some setting of the coefficients $\{\gamma_{i,i',j,d,e}\}$, $\{\delta_{i,i',j,d,e,f}\}$, $\{\eta_{i,d}\}$, $\{\mu_{i,d,f}\}$, $\{\nu_{i',j,e}\}$, κ_1, κ_0

It will be helpful to factor out β_b. We denote the term multiplying β_b as P_1 and the remaining term with no dependence on x_b as P_0:

$$\begin{aligned}
P_1 &:= \sum_{\substack{i \in [n], i' \in [n], j \in [J] \\ d \in \{1,2\}, e \in \{1,2\}}} \gamma_{i,i',j,d,e} L_i(x_b) r_{i,x_{b,i}}^d a_{i',e}^{(j)} + \sum_{i \in [n], d \in \{1,2\}} \eta_{i,d} L_i(x_b) r_{i,x_{b,i}}^d + \kappa_1 \\
P_0 &:= \sum_{\substack{i \in [n], i' \in [n], j \in [J] \\ d \in \{1,2\}, e \in \{1,2\}, f \in \{0,1\}}} \delta_{i,i',j,d,e,f} r_{i,f}^d a_{i',e}^{(j)} + \sum_{i \in [n], d \in \{1,2\}, f \in \{0,1\}} \mu_{i,d,f} r_{i,f}^d \\
&\quad + \sum_{i' \in [n], j \in [J], e \in \{1,2\}} \nu_{i',j,e} a_{i',e}^{(j)} + \kappa_0.
\end{aligned} \tag{5.11}$$

Equation 5.10 can then be re-expressed as

$$P_1\beta_b + P_0. \tag{5.12}$$

We write the kth zero test query submitted by $\mathcal{A}$ as $P_1^{(k)}\beta_b + P_0^{(k)}$.

First, we claim that for any set of K zero test queries submitted by $\mathcal{A}$, that for all $k \in [K]$,

$$\Pr\left[P_1^{(k)} = 0 \wedge b = 0 \Leftrightarrow P_1^{(k)} = 0 \wedge b = 1\right] \geq 1 - \frac{(12n+4)K}{p}.$$

To show this, consider any fixed setting of the γ, η, and κ_1 coefficients that comprise P_1. Lemma 5.2.2 shows that regardless of the attribute x, and thus regardless of the bit b in this scenario, if the coefficients meet the six constraints stated in the lemma, then $P_1 = 0$ with probability 1. If the coefficients do not meet the constraints, then $P_1 = 0$ with probability at most $\frac{6n+2}{p}$. Thus with probability at least $1 - \frac{6n+2}{p}$, $P_1 = 0$ if and only if the coefficients meet these constraints. By a union bound over the K zero test queries made by $\mathcal{A}$, this holds simultaneously for all zero test queries with probability at least $1 - \frac{(6n+2)K}{p}$. Now since this property holds for both $b = 0$ and $b = 1$, another union bound over the two settings gives the claim.

Now we claim that with probability $1 - \frac{(12n+8)K}{p}$, the values returned by the K zero test queries are completely independent of the bit b. Consider the pair (P_1, P_0) corresponding to a zero test query submitted by $\mathcal{A}$. By Schwartz-Zippel over the randomness of β_b, the result of the zero test query is 'zero' if and only if $P_0 = P_1 = 0$, with probability $1 - 1/p$. Then by a union bound over the K zero test queries submitted by $\mathcal{A}$, with probability at least $1 - K/p$, the result of *each* of the K zero test queries is 'zero' if and only if the corresponding $P_0 = P_1 = 0$. Now by a union bound over the two settings of b and with the previous claim, with probability at least $1 - \frac{(12n+6)K}{p}$ the result of each zero test query will either be 'non-zero' (corresponding to non-zero P_1) or will be 'zero' if and only if $P_0 = 0$ (corresponding to $P_1 = 0$). Since P_0 is the same expression regardless of the bit b, the claim follows.

Finally, we note that in the generic group model, $\mathcal{A}$ gains information *only* from successful zero test. This is because the result of any Add or Pair operation is a new uniformly drawn handle. Thus $\mathcal{A}$ can only hope to distinguish between two implementations of the generic group model by submitting a set of coefficients for zero testing such that the result is 'zero' in one implementation and 'non-zero' in the other with some noticeable difference in probability. However, we showed above that this does not happen except with probability at most $\frac{(12n+6)K}{p}$.

We now split into two cases in order to argue that the above implies message privacy for scheme Π with at most the same advantage. We analyze what additional advantage $\mathcal{A}$ has when interacting with scheme Π instead of scheme Π'. We show that this advantage does not improve $\mathcal{A}$'s distinguishing probability.

Case 1: $J' = \emptyset$. In scheme Π, $\mathcal{A}$ no longer has a priori access to the handle β_b. But since it is given $H(\beta_b) \oplus y_b$, there is a possibility that $\mathcal{A}$ can gain distinguishing information (such as y_b) by obtaining a handle to β_b via Add and Pair operations, since it can evaluate H on the handle and compare with $H(\beta_b) \oplus y_b$. However, obtaining a handle to β_b in $\mathsf{Expt}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n, m, b)$ implies successfully zero testing with non-trivial P_1 coefficients in $\mathsf{Expt}^{\mathsf{MP}}_{\Pi',\mathcal{A}}(\lambda, n, m, b)$. But $J' = \emptyset$ implies via Lemma 5.2.2, Schwartz-Zippel over β_b, and a union bound, that $\mathcal{A}$ cannot obtain a successful zero test in $\mathsf{Expt}^{\mathsf{MP}}_{\Pi,\mathcal{A}}(\lambda, n, m, b)$ with any setting of non-trivial P_1 coefficients, except with probability at most $\frac{(12n+6)K}{p}$. This is because $J = \emptyset$ implies that *all* γ and η terms are 0. Then, in order to obtain 'zero' from zero testing, κ_1 (the coefficient of β_b) must be 0 as well.

Case 2: $J' \neq \emptyset$, which by admissibility implies that $y_0 = y_1$. Now, the adversary can obtain a successful zero test with non-trivial P_1 coefficients when interacting with Π'. This means that there is a possibility of obtaining a handle to β_b when interacting with Π. Since admissibility requires that $y_0 = y_1$, such a handle does not allow distinguishing via payload decryption. However, when interacting with Π, $\mathcal{A}$ knows when it obtains a handle to β_b. The question remains whether the adversary can use this knowledge for a distinguishing advantage. So, assume toward

a contradiction an adversary $\mathcal{B}$ exists that breaks message privacy for scheme Π by obtaining a handle to β_b. By subtracting, this implies an adversary that breaks message privacy for Π' by way of a zero test query, which we ruled out above except with probability at most $\frac{(12n+6)K}{p}$. $\square$

5.2.2.3 Enhanced Function Privacy

Theorem 5.2.9. *Let Π be the wildcard encryption scheme and $\mathcal{D}$ be the uniform distribution on patterns with a fixed number of wildcards $n - w(n)$. Then there exists a simulator $\mathcal{S}$ such that for all* PPT *adversaries $\mathcal{A}$ that performs K zero test queries, J queries to its* KeyGen *oracle, and Q queries to its* Enc *oracle, there exists a* PPT *adversary $\mathcal{B}$ such that for all $\lambda \in \mathbb{N}$ and $n \in \mathbb{N}$, it holds that*

$$\mathbf{Adv}^{\mathsf{eFP}}_{\Pi,\mathcal{S},\mathcal{A}}(\lambda, \mathcal{D}, n, m) \leq \frac{(12n+5)K}{p} + \frac{2JQ}{2^{w(n)}} + J \cdot \mathbf{Adv}^{\mathsf{dist-VBB}}_{\mathsf{Obf},\mathcal{S}_{\mathsf{Obf}},\mathcal{B}}(\lambda, \mathcal{D}, n)$$

Proof. We prove enhanced function privacy for the same modified scheme Π' as in Section 5.2.2.2. Recall that in the scheme Π', the adversary does not include the payload y in the encryption oracle query and $[\![\beta_q]\!]_1$ is given to the adversary instead of $H([\![\beta_q]\!]_1) \oplus y$ on its q'th query to the Enc oracle.

Enhanced function privacy for Π' implies enhanced function privacy for Π with the same advantage, which we show via reduction as follows. Consider a successful adversary $\mathcal{A}$ against enhanced function privacy for scheme Π. We can then use $\mathcal{A}$ to construct a successful adversary $\mathcal{B}$ against enhanced function privacy for scheme Π'. $\mathcal{B}$ simply follows $\mathcal{A}$'s routine, with the following alteration when $\mathcal{A}$ makes query q to the Enc oracle. Since the hash function H is public, $\mathcal{B}$ simply hashes its received handle to β_q and XORs the result of this hash with the y that $\mathcal{A}$ would have queried to the Enc oracle had it been interacting with Π. The result of this is exactly what $\mathcal{A}$ would have received when interacting with Π, so $\mathcal{B}$ returns this value to $\mathcal{A}$, along with the other $2n$ handles received from the model. The rest of the interaction is unchanged, and $\mathcal{B}$ simply outputs whatever $\mathcal{A}$ outputs, completing the reduction.

Proof Strategy: Since KeyGen uses the BKMPRS obfuscation as a subroutine, we would like to reduce function privacy to the security of BKMPRS obfuscation. However, the security of BKMPRS obfuscation relies crucially on the fact that an adversary can't find an accepting input except with negligible probability. In enhanced function privacy, we explicitly give the adversary ciphertexts that correspond to accepting inputs, so at first this notion of security seems like a hopeless goal. However, we are saved by the fact that the accepting input (represented by the appropriate Lagrange coefficients) is encoded in a group. This implies that an adversary can learn that an attribute is an accepting input without learning what the attribute actually is or anything else about the pattern.

Now to argue security, we introduce a sequence of hybrid simulations. The hybrids begin with the real protocol and end with simulations of KeyGen and Enc that are independent of the master secret key. The first thing we do is alter the Enc oracle to be independent of the patterns drawn by the KeyGen oracle. We do this by appealing to Lemma 5.2.2, which intuitively says that an encryption of an accepting attribute relative to some secret key sk_j induces just a single linear relation among the elements of the secret key and the elements of the encryption. Then, once the Enc oracle is independent of patterns drawn by the KeyGen oracle, we are free to replace the BKMPRS obfuscation in KeyGen with random and appeal to the security of BKMPRS.

Once the obfuscation returns random elements, the last observation is that during KeyGen, the master secret key elements are only used in the context of a full rank linear transformation from the BKMPRS output to the resulting secret key elements. Thus, if the BKMPRS output is actually uniformly random, then the secret key elements will also be uniformly random and independent of the secret key elements, completing the proof.

Now we proceed with the proof. In order to describe the hybrid simulations concisely, we introduce some notation. First, let $\mathsf{Obf}_{exp}(\mathsf{pat})$ be a routine that runs BKMPRS obfuscation on pattern pat and outputs the $2n$ *exponents* that result rather than the $2n$ group elements. We also

let $V_i = \begin{bmatrix} r_{i,0}^2 & r_{i,0} \\ r_{i,1}^2 & r_{i,1} \end{bmatrix}$, which is the full rank transformation used by KeyGen that was mentioned above. Finally, whenever we say *sample*, we mean sample uniformly at random from the space that should be clear from context.

Throughout the proof, we assume the adversary makes J queries to its KeyGen oracle and Q queries to its Enc oracle. We'll consider three spaces of handles corresponding to exponents encoded in the three different groups $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_T$. Also, note that our description of the bilinear generic group model does not differentiate between group elements and their exponents. Therefore, we essentially lose the notion of a public key when describing the model simulations, so we'll imagine giving $\mathcal{O}_{\mathsf{Enc}}$ the master secret key rather than the public key when describing the real interaction (experiment 0). The important point is that in our final simulation (experiment 1), neither $\mathcal{O}_{\mathsf{Enc}}$ nor $\mathcal{O}_{\mathsf{KeyGen}}$ receive any information about the master secret key, in fact, their implementation is completely independent of Setup.

We begin by describing how the real protocol, which we call $\mathsf{Sim\text{-}Real}$, proceeds in the generic bilinear group model. As in the generic description of the generic bilinear group oracle given in Definition 5.1.2, $\mathsf{Sim\text{-}Real}$ maintains a table that maps between exponents and handles. In $\mathsf{Sim\text{-}Real}$, Add, Pair, and $\mathsf{ZeroTest}$ are implemented as described in the definition, with $\mathsf{Sim\text{-}Real}$ generating a new handle to represent the result of each Add and Pair operation over exponents in $\mathbb{Z}_p$. Note that this behavior means that multiple handles could end up mapping to the same exponent. However, the adversary can always notice this by zero-testing the difference of the two handles. As a final bit of notation, for an exponent $r \in \mathbb{Z}_p$, let $h_r^{(b)}$ be the result of drawing a random string in the handle space $\mathcal{H}_b$ (for $b \in \{1, 2, T\}$) and associating the result with r. In addition, $\mathsf{Sim\text{-}Real}$ calls the obfuscator Obf_exp explicitly. This is equivalent the functioning of protocol Π. We define $\mathsf{Sim\text{-}Real}$ in the following figure.

$\mathsf{Sim\text{-}Real}(1^\lambda, \mathcal{D}, n)$

Sample $\mathsf{msk} := \{r_{i,0}, r_{i,1}\}_{i\in[n]}$
Initialize table $\mathcal{T}$ with
$\{(r_{i,0}, h^{(1)}_{r_{i,0}}), (r^2_{i,0}, h^{(1)}_{r^2_{i,0}}), (r_{i,1}, h^{(1)}_{r_{i,1}}), (r^2_{i,1}, h^{(1)}_{r^2_{i,1}})\}_{i\in[n]}$
Initialize $j = 1, q = 1$
return $\mathcal{A}^{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk}),\mathcal{O}_{\mathsf{Enc}}(\mathsf{msk},\cdot)}(1^\lambda, \mathsf{pk})$

$\underline{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk})}$
$\{q_{i,b,j}\}_{i\in[n],b\in\{0,1\}} \leftarrow \mathsf{Obf}_{exp}(\mathsf{pat}_j \leftarrow \mathcal{D})$
for $i \in [n]$:
$$\begin{bmatrix} a_{i,2,j} \\ a_{i,1,j} \end{bmatrix} := V_i^{-1} \cdot \begin{bmatrix} q_{i,0,j} \\ q_{i,1,j} \end{bmatrix}$$
Add to table $\mathcal{T}$
$\{(a_{i,1,j}, h^{(2)}_{a_{i,1,j}}), (a_{i,2,j}, h^{(2)}_{a_{i,2,j}})\}_{i\in[n]}$
j++
return $\{h^{(2)}_{a_{i,1,j}}, h^{(2)}_{a_{i,2,j}}\}_{i\in[n]}$

$\underline{\mathcal{O}_{\mathsf{Enc}}(\mathsf{msk}, j)}$
Choose any x_q matching pat_j, calculate $\{L_i(x_q)\}_{i\in[n]}$
Sample β_q
Define $e_{i,2,q} := \beta_q r^2_{i,x_{q,i}} L_i(x_q), e_{i,1,q} := \beta_q r_{i,x_{q,i}} L_i(x_q)$
Add to table $\mathcal{T}$
$\{(e_{i,2,q}, h^{(1)}_{e_{i,2,q}}), (e_{i,1,q}, h^{(1)}_{e_{i,1,q}})\}_{i\in[n]}, (\beta_q, h^{(1)}_{\beta_q})$
q++
return $\{h^{(1)}_{e_{i,2,q}}, h^{(1)}_{e_{i,1,q}}\}_{i\in[n]}, h^{(1)}_{\beta_q}$

Next, we give the hybrid Sim-Enc, where we remove the dependence of $\mathcal{O}_{\mathsf{Enc}}$ on patterns drawn by $\mathcal{O}_{\mathsf{KeyGen}}$. To argue this, we introduce a set of formal variables $\mathcal{E} := \{\mathbf{e}_{i,2,q}, \mathbf{e}_{i,1,q}\}_{i\in n, q\in[Q]}$ into the exponent space. Thus Sim-Enc implements Add, Pair, and ZeroTest over the ring $\mathbb{Z}_p[\mathcal{E}]$ instead of $\mathbb{Z}_p$.

$\mathsf{Sim\text{-}Enc}(1^\lambda, \mathcal{D}, n)$

Sample $\mathsf{msk} := \{r_{i,0}, r_{i,1}\}_{i\in[n]}$
Initialize table $\mathcal{T}$ with
$\{(r_{i,0}, h^{(1)}_{r_{i,0}}), (r^2_{i,0}, h^{(1)}_{r^2_{i,0}}), (r_{i,1}, h^{(1)}_{r_{i,1}}), (r^2_{i,1}, h^{(1)}_{r^2_{i,1}})\}_{i\in[n]}$
Initialize $j = 1, q = 1$
return $\mathcal{A}^{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk}), \mathcal{O}_{\mathsf{Enc}}(\cdot)}(1^\lambda, \mathsf{pk})$

$\underline{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk})}$
$\{q_{i,b,j}\}_{i\in[n], b\in\{0,1\}} \leftarrow \mathsf{Obf}_{exp}(\mathsf{pat}_j \leftarrow \mathcal{D})$
for $i \in [n]$:
$$\begin{bmatrix} a_{i,2,j} \\ a_{i,1,j} \end{bmatrix} := V_i^{-1} \cdot \begin{bmatrix} q_{i,0,j} \\ q_{i,1,j} \end{bmatrix}$$
Add to table $\mathcal{T}$
$\{(a_{i,1,j}, h^{(2)}_{a_{i,1,j}}), (a_{i,2,j}, h^{(2)}_{a_{i,2,j}})\}_{i\in[n]}$
j++
return $\{h^{(2)}_{a_{i,1,j}}, h^{(2)}_{a_{i,2,j}}\}_{i\in[n]}$

$\underline{\mathcal{O}_{\mathsf{Enc}}(j)}$
Let $e_{j,q}(\mathcal{E}) = \sum_{i\in[n], d\in\{1,2\}} a_{i,d,j} \mathbf{e}_{i,d,q}$
Add to table $\mathcal{T}$
$\{(\mathbf{e}_{i,2,q}, h^{(1)}_{\mathbf{e}_{i,2,q}}), (\mathbf{e}_{i,1,q}, h^{(1)}_{\mathbf{e}_{i,1,q}})\}_{i\in[n]}, (e_{j,q}(\mathcal{E}), h^{(1)}_{e_{j,q}(\mathcal{E})})$
q++
return $\{h^{(1)}_{\mathbf{e}_{i,2,q}}, h^{(1)}_{\mathbf{e}_{i,1,q}}\}_{i\in[n]}, h^{(1)}_{e_{j,q}(\mathcal{E})}$

Lemma 5.2.10. *A* PPT *adversary $\mathcal{A}$ making K zero test queries, J $\mathcal{O}_{\mathsf{KeyGen}}$ queries, and Q $\mathcal{O}_{\mathsf{Enc}}$ queries can distinguish between $\mathsf{Sim\text{-}Real}(1^\lambda, \mathcal{D}, n)$ and $\mathsf{Sim\text{-}Enc}(1^\lambda, \mathcal{D}, n)$ with probability at most $\frac{(12n+5)K}{p} + \frac{JQ}{2^{w(n)}}$ for distributions $\mathcal{D}$ that are uniform over strings with $n - w(n)$ wildcards.*

Proof. Let j_q be the value of j submitted on the q'th query to $\mathcal{O}_{\mathsf{Enc}}$ and x_q be the attribute selected uniformly by $\mathcal{O}_{\mathsf{Enc}}$ for that query. First, we show that with probability $\frac{JQ}{2^{w(n)}}$, for every $j \in [J], q \in [Q]$, x_q does not match $\mathsf{pat}_{j'}$ for $j' \neq j_q$. Since the $w(n)$ fixed positions of each $\mathsf{pat}_{j'}$ are uniformly random and independent of x_q, the probability that x_q matches the pattern is $1/2^{w(n)}$. The claim follows from a union bound.

In the case that each queried ciphertext matches exactly one pattern, we follow the same

proof outline as in the proof of message privacy. That is, the general form of a polynomial that $\mathcal{A}$ can construct in the Sim-Real game is

$$\begin{aligned}
&\sum_{q\in[Q]} \beta_q\Bigg(\sum_{\substack{i\in[n],i'\in[n],j\in[J]\\ d\in\{1,2\},e\in\{1,2\}}} \gamma_{i,i',j,d,e,q} L_i(x_q) r^d_{i,x_{q,i}} a_{i',e,j} + \sum_{i\in[n],d\in\{1,2\}} \eta_{i,d,q} L_i(x_q) r^d_{i,x_{q,i}} + \kappa_q\Bigg) \\
&+\Bigg(\sum_{\substack{i\in[n],i'\in[n],j\in[J]\\ d\in\{1,2\},e\in\{1,2\},f\in\{0,1\}}} \delta_{i,i',j,d,e,f} r^d_{i,f} a^{(j)}_{i',e} + \sum_{i\in[n],d\in\{1,2\},f\in\{0,1\}} \mu_{i,d,f} r^d_{i,f} \\
&+ \sum_{i'\in[n],j\in[J],e\in\{1,2\}} \nu_{i',j,e} a^{(j)}_{i',e} + \kappa\Bigg) \\
&:= \sum_{q\in[Q]} \beta_q P_q + P_0.
\end{aligned} \tag{5.13}$$

The general form of a polynomial that $\mathcal{A}$ can construct in the Sim-Enc game is

$$\begin{aligned}
&\sum_{q\in[Q]}\Bigg(\sum_{\substack{i\in[n],i'\in[n],j\in[J]\\ d\in\{1,2\},e\in\{1,2\}}} \gamma_{i,i',j,d,e,q} a_{i',e,j} \mathbf{e}_{i,d,q} + \sum_{i\in[n],d\in\{1,2\}} \eta_{i,d,q} \mathbf{e}_{i,d,q} + \kappa_q\Bigg(\sum_{i\in[n],d\in\{1,2\}} a_{i,d,j_q} \mathbf{e}_{i,d,q}\Bigg)\Bigg) \\
&+\Bigg(\sum_{\substack{i\in[n],i'\in[n],j\in[J]\\ d\in\{1,2\},e\in\{1,2\},f\in\{0,1\}}} \delta_{i,i',j,d,e,f} r^d_{i,f} a^{(j)}_{i',e} + \sum_{i\in[n],d\in\{1,2\},f\in\{0,1\}} \mu_{i,d,f} r^d_{i,f} \\
&+ \sum_{i'\in[n],j\in[J],e\in\{1,2\}} \nu_{i',j,e} a^{(j)}_{i',e} + \kappa\Bigg) \\
&:= \sum_{q\in[Q]} P_q(\mathcal{E}) + P_0.
\end{aligned} \tag{5.14}$$

We express the k'th zero test query submitted by $\mathcal{A}$ to Sim-Real as $\sum_{q\in[Q]} P_q^{(k)}\beta_q + P_0^{(k)}$ and to Sim-Enc as $\sum_{q\in[Q]} P_q^{(k)}(\mathcal{E}) + P_0^{(k)}$. We claim that for any set of K zero test queries submitted by $\mathcal{A}$, with probability at least $1 - \frac{(12n+4)K}{p} - \frac{JQ}{2^{w(n)}}$, the following holds simultaneously for all $k \in [K]$:

$$P_q^{(k)} = 0\ \forall q \in [Q] \text{ in Sim-Real } \Leftrightarrow P_q^{(k)}(\mathcal{E}) = 0\ \forall q \in [Q] \text{ in Sim-Enc.}$$

To show this, we characterize the sets of coefficients that $\mathcal{A}$ can submit when interacting with

$\mathsf{Sim\text{-}Enc}$ that result in $P_q(\mathcal{E}) = 0$ for all $q \in [Q]$. We claim that $\mathcal{A}$ satisfies this with probability 1 if the following constraints on the coefficients hold, and otherwise this is satisfied with probability at most $\frac{(6n+2)K}{p}$ (over the randomness of the $a_{i,e,j}$ values). These constraints are:

- $\gamma_{1,1,j_q,1,1,q} = \gamma_{1,1,j_q,2,2,q} = ... = \gamma_{n,n,j_q,1,1,q} = \gamma_{n,n,j_q,2,2,q} = \kappa_q$ for all $q \in [Q]$
- $\gamma_{1,1,j,1,1,q} = \gamma_{1,1,j,2,2,q} = ... = \gamma_{n,n,j,1,1,q} = \gamma_{n,n,j,2,2,q} = 0$ for all $j \neq j_q$
- $\gamma_{i,i',j,d,e,q} = 0$ for all $i \neq i'$, $d \neq e$
- $\eta_{i,d,q} = 0$ for all $i \in [n], d \in \{1,2\}, q \in [Q]$.

Assume toward a contradiction that for some $q \in [Q]$, there exists a setting of coefficients other than $\gamma_{1,1,j_q,1,1,q} = \gamma_{1,1,j_q,2,2,q} = ... = \gamma_{n,n,j_q,1,1,q} = \gamma_{n,n,j_q,2,2,q} = \kappa_q$ and all others equal to 0, such that $P_q = 0$ with probability greater than $\frac{6n+2}{p}$. Then, replace the $\mathbf{e}_{i,d,q}$ variables in $P_q(\mathcal{E})$ with $L_i(x) r^d_{i,x_i}$ for some x which matches pattern pat_{j_q}. This results in a P_q which can be written in the form of Lemma 5.2.2, since $\sum_{i\in[n],d\in\{1,2\}} a_{i,d,j_q} L_i(x) r^d_{i,x_i} = 1$ by correctness. But by assumption, the coefficients do not match the constraints given by Lemma 5.2.2, and P_q still evaluates to zero with probability at least as large as before ($\frac{6n+2}{p}$) which is a contradiction. Now, by a union bound over $\mathcal{A}$'s set of K zero test queries, $\mathcal{A}$ will only be able to set $P_q^{(k)}(\mathcal{E}) = 0$ for all $q \in [Q], k \in [K]$ by setting coefficients to match the above constraints except with probability $\frac{(6n+2)K}{p}$.

We characterize the set of coefficients that $\mathcal{A}$ can submit that result in $P_q^{(k)} = 0$ for all $q \in Q, k \in [K]$ in $\mathsf{Sim\text{-}Real}$. We argue that $\mathcal{A}$ can only satisfy this with probability greater than $\frac{(6n+2)K}{p} + \frac{JQ}{2^{w(n)}}$ by following the same constraints above for each of its K zero test queries. If for all $q \in [Q]$, x_q only matches the pattern corresponding to the single secret key sk_{j_q} (which we know is true with probability at least $1 - \frac{JQ}{2^{w(n)}}$), Lemma 5.2.2 shows that the adversary can only choose non-trivial coefficients such that $P_q = 0$ for all $q \in [Q]$ with probability greater than $\frac{(6n+2)}{p}$ by setting $\gamma_{1,1,j_q,1,1,q} = \gamma_{1,1,j_q,2,2,q} = ... = \gamma_{n,n,j_q,1,1,q} = \gamma_{n,n,j_q,2,2,q} = \kappa_q$ and the rest 0. The claim follows from a union bound over the K zero test queries. Now combining the two characterizations with a union bound, we see that the original claim follows.

We claim that with probability at least $1 - \frac{(12n+5)K}{p} - \frac{JQ}{2^{w(n)}}$, the answers returned by any K zero test queries submitted by $\mathcal{A}$ are completely independent of whether $\mathcal{A}$ was interacting with $\mathsf{Sim\text{-}Real}$ or $\mathsf{Sim\text{-}Enc}$. By Schwartz-Zippel over the randomness of the β_q values, the result of each of the K zero test queries submitted by $\mathcal{A}$ when interacting with $\mathsf{Sim\text{-}Real}$ is 'zero' if and and only $P_0^{(k)} = P_1^{(k)} = ... = P_Q^{(k)} = 0$ for all $k \in [K]$. Note that this property is true with probability 1 in $\mathsf{Sim\text{-}Enc}$. So by a union bound with the previous claim, with probability at least $1 - \frac{(12n+5)K}{p} - \frac{JQ}{2^{w(n)}}$, the result of each of the K zero test queries will either be 'non-zero' (corresponding to some non-zero P_q for $q \in [Q]$) or 'zero' if and only if $P_0 = 0$ (corresponding to $P_q = 0 \; \forall q \in [Q]$). Since P_0 is the same expression in $\mathsf{Sim\text{-}Real}$ and in $\mathsf{Sim\text{-}Enc}$, the claim follows.

This is now sufficient to complete the proof by using the same generic bilinear group model properties as in the message privacy proof.

□

We now continue with a sequence of hybrids that replace the BKMPRS subroutine in KeyGen with a subroutine that simply outputs random elements in $\mathbb{Z}_p$. We do this for one KeyGen query at a time, resulting in J new hybrids.

$\mathsf{Sim\text{-}KeyGen\text{-}}j'(1^\lambda, \mathcal{D}, n)$

Sample $\mathsf{msk} := \{r_{i,0}, r_{i,1}\}_{i\in[n]}$
Initialize table $\mathcal{T}$ with
$\{(r_{i,0}, h^{(1)}_{r_{i,0}}), (r^2_{i,0}, h^{(1)}_{r^2_{i,0}}), (r_{i,1}, h^{(1)}_{r_{i,1}}), (r^2_{i,1}, h^{(1)}_{r^2_{i,1}})\}_{i\in[n]}$
Initialize $j = 1, q = 1$
return $\mathcal{A}^{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk}), \mathcal{O}_{\mathsf{Enc}}(\cdot)}(1^\lambda, \mathsf{pk})$

$\underline{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk})}$
if $j \le j'$:
Sample $\{q_{i,b,j}\}_{i\in[n], b\in\{0,1\}}$
else
$\{q_{i,b,j}\}_{i\in[n], b\in\{0,1\}} \leftarrow \mathsf{Obf}_{exp}(\mathsf{pat}_j \leftarrow \mathcal{D})$
for $i \in [n]$:
$\begin{bmatrix} a_{i,2,j} \\ a_{i,1,j} \end{bmatrix} := V_i^{-1} \cdot \begin{bmatrix} q_{i,0,j} \\ q_{i,1,j} \end{bmatrix}$
Add to table $\mathcal{T}$
$\{(a_{i,1,j}, h^{(2)}_{a_{i,1,j}}), (a_{i,2,j}, h^{(2)}_{a_{i,2,j}})\}_{i\in[n]}$
j++
return $\{h^{(2)}_{a_{i,1,j}}, h^{(2)}_{a_{i,2,j}}\}_{i\in[n]}$

$\underline{\mathcal{O}_{\mathsf{Enc}}(j)}$
Let $e_{j,q}(\mathcal{E}) = \sum_{i\in[n], d\in\{1,2\}} a_{i,d,j}\mathbf{e}_{i,d,q}$
Add to table $\mathcal{T}$
$\{(\mathbf{e}_{i,2,q}, h^{(1)}_{\mathbf{e}_{i,2,q}}), (\mathbf{e}_{i,1,q}, h^{(1)}_{\mathbf{e}_{i,1,q}})\}_{i\in[n]}, (e_{j,q}(\mathcal{E}), h^{(1)}_{e_{j,q}(\mathcal{E})})$
q++
return $\{h^{(1)}_{\mathbf{e}_{i,2,q}}, h^{(1)}_{\mathbf{e}_{i,1,q}}\}_{i\in[n]}, h^{(1)}_{e_{j,q}(\mathcal{E})}$

Note: Sim-KeyGen-0 is equivalent to Sim-Enc

Lemma 5.2.11. *For all $j' \in [J]$, a* PPT *adversary $\mathcal{A}$ making K zero test queries, J $\mathcal{O}_{\mathsf{KeyGen}}$ queries, and Q $\mathcal{O}_{\mathsf{Enc}}$ queries can distinguish between Sim-KeyGen-$(j'-1)(1^\lambda, \mathcal{D}, n)$ and Sim-KeyGen-$j'(1^\lambda, \mathcal{D}, n)$ with probability at most $\mathbf{Adv}^{\mathsf{dist-VBB}}_{\mathsf{Obf}, \mathcal{S}_{\mathsf{Obf}}, \mathcal{B}}(\lambda, \mathcal{D}, n)$.*

Proof. We define the following reduction $\mathcal{B}$ which interacts with the BKMPRS obfuscation generic group model security game and presents a simulation of the generic group model to $\mathcal{A}$. We define

$\mathcal{O}_{\mathsf{Obf}}$ to be the oracle which implements the BKMPRS security game - that is, it either honestly implements the generic group model for the obfuscation or outputs handles to random group elements. We introduce an additional set of formal variables $\mathcal{G} := \{\mathbf{g}_{i,b}\}_{i\in[n],b\in\{0,1\}}$ which $\mathcal{B}$ will associate with the $2n$ handles $\{h^*_{i,b}\}_{i\in[n],b\in\{0,1\}}$ that it receives from $\mathcal{O}_{\mathsf{Obf}}$. Now $\mathcal{B}$ implements its Add and Pair operations over the ring $\mathbb{Z}_p[\mathcal{E},\mathcal{G}]$. zero test will still be over the ring $\mathbb{Z}_p[\mathcal{E}]$ and will be discussed on more detail below. One thing to note is that the handles $\{h^*_{i,b}\}_{i\in[n],b\in\{0,1\}}$ will only be used (and $\mathcal{O}_{\mathsf{Obf}}$ will only be interacted with) during zero test queries to $\mathcal{B}$.

$\mathcal{B}^{\mathcal{O}_{\mathsf{Obf}}}(1^\lambda, \mathcal{D}, n)$

Sample $\mathsf{msk} := \{r_{i,0}, r_{i,1}\}_{i \in [n]}$
Initialize table $\mathcal{T}$ with
$\{(r_{i,0}, h^{(1)}_{r_{i,0}}), (r^2_{i,0}, h^{(1)}_{r^2_{i,0}}), (r_{i,1}, h^{(1)}_{r_{i,1}}), (r^2_{i,1}, h^{(1)}_{r^2_{i,1}})\}_{i \in [n]}$
Initialize $j = 1, q = 1$
Receive $\{h^*_{i,b}\}_{i \in [n], b \in \{0,1\}} \leftarrow \mathcal{O}_{\mathsf{Obf}}$
return $\mathcal{A}^{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk}), \mathcal{O}_{\mathsf{Enc}}(\cdot)}(1^\lambda, \mathsf{pk})$

$\underline{\mathcal{O}_{\mathsf{KeyGen}}(\mathsf{msk})}$
if $j < k$:
Sample $\{q_{i,b,j}\}_{i \in [n], b \in \{0,1\}}$
for $i \in [n]$:
$$\begin{bmatrix} a_{i,2,j} \\ a_{i,1,j} \end{bmatrix} := V_i^{-1} \cdot \begin{bmatrix} q_{i,0,j} \\ q_{i,1,j} \end{bmatrix}$$
Add to table $\mathcal{T}$
$\{(a_{i,1,j}, h^{(2)}_{a_{i,1,j}}), (a_{i,2,j}, h^{(2)}_{a_{i,2,j}})\}_{i \in [n]}$
else if $j = j'$
for $i \in [n]$:
$$\begin{bmatrix} a_{i,2,j}(\mathcal{G}) \\ a_{i,1,j}(\mathcal{G}) \end{bmatrix} := V_i^{-1} \cdot \begin{bmatrix} \mathbf{g}_{i,0} \\ \mathbf{g}_{i,1} \end{bmatrix}$$
Add to table $\mathcal{T}$
$\{(a_{i,1,j}(\mathcal{G}), h^{(2)}_{a_{i,1,j}(\mathcal{G})}), (a_{i,2,j}(\mathcal{G}), h^{(2)}_{a_{i,2,j}(\mathcal{G})})\}_{i \in [n]}$
else
$\{q_{i,b,j}\}_{i \in [n], b \in \{0,1\}} \leftarrow \mathsf{Obf}_{exp}(\mathsf{pat}_j \leftarrow \mathcal{D})$
for $i \in [n]$:
$$\begin{bmatrix} a_{i,2,j} \\ a_{i,1,j} \end{bmatrix} := V_i^{-1} \cdot \begin{bmatrix} q_{i,0,j} \\ q_{i,1,j} \end{bmatrix}$$
Add to table $\mathcal{T}$
$\{(a_{i,1,j}, h^{(2)}_{a_{i,1,j}}), (a_{i,2,j}, h^{(2)}_{a_{i,2,j}})\}_{i \in [n]}$
j++
return $\{h^{(2)}_{a_{i,1,j}}, h^{(2)}_{a_{i,2,j}}\}_{i \in [n]}$

$\underline{\mathcal{O}_{\mathsf{Enc}}(\mathsf{j})}$
Let $e_{j,q}(\mathcal{E}, \mathcal{G}) = \sum_{i \in [n], d \in \{1,2\}} a_{i,d,j}(\mathcal{G}) \mathbf{e}_{i,d,q}$
Add to table $\mathcal{T}$
$\{(\mathbf{e}_{i,2,q}, h^{(1)}_{\mathbf{e}_{i,2,q}}), (\mathbf{e}_{i,1,q}, h^{(1)}_{\mathbf{e}_{i,1,q}})\}_{i \in [n]}, (e_{j,q}(\mathcal{E}, \mathcal{G}), h^{(1)}_{e_{j,q}(\mathcal{E},\mathcal{G})})$
q++
return $\{h^{(1)}_{\mathbf{e}_{i,2,q}}, h^{(1)}_{\mathbf{e}_{i,1,q}}\}_{i \in [n]}, h^{(1)}_{e_{j,q}(\mathcal{E},\mathcal{G})}$

On a zero test query, $\mathcal{B}$ first writes the associated expression in the form

$$\sum_{i\in[n],b\in\{1,2\},q\in[Q]} P_{i,b,q}(\mathcal{G})\mathbf{e}_{i,b,q} + P_0$$

That is, it stratifies by the $\mathcal{E}$ variables and writes each coefficient as a linear polynomial in the $\mathcal{G}$ variables. This is possible because formal variables in $\mathcal{E}$ are only encoded in $\mathbb{G}_1$ and formal variables in $\mathcal{G}$ are only encoded in $\mathbb{G}_2$. Now note that this expression is zero over $\mathbb{Z}[\mathcal{E}]$ if and only if $P_{i,b,q}(\mathcal{G}) = 0$ for all $i \in [n], b \in \{1,2\}, q \in [Q]$ and $P_0 = 0$. Since each $P_{i,b,q}(\mathcal{G})$ is a linear polynomial in the $\mathcal{G}$ variables, $\mathcal{B}$ can check if it evaluates to 0 through a series of Add operations to $\mathcal{O}_{\mathsf{Obf}}$ over the $\{h^*_{i,b}\}_{i\in[n],b\in\{0,1\}}$ handles followed by a zero test.

Now if $\mathcal{O}_{\mathsf{Obf}}$ is implementing an honest obfuscation, then the j''th KeyGen query is no different than the queries for $j > j'$, and $\mathcal{A}$'s view of the model is *exactly* its view of the $\mathsf{Sim\text{-}KeyGen}$-$(j'-1)$ hybrid. Otherwise, there are uniformly random elements of $\mathbb{Z}_p$ associated with the $\{h^*_{i,b}\}_{i\in[n],b\in\{0,1\}}$ handles and the j''th query is no different than all the queries for $j < j'$. Thus $\mathcal{A}$'s view of the model is *exactly* its view of $\mathsf{Sim\text{-}KeyGen}$-$j'$, which completes the proof. □

Finally, we give the simulator $\mathsf{Sim\text{-}Ideal}$, which is independent of the master secret key.

$\mathsf{Sim\text{-}Ideal}(1^\lambda, \mathcal{D}, n)$

Sample $\mathsf{msk} := \{r_{i,0}, r_{i,1}\}_{i\in[n]}$
Initialize table $\mathcal{T}$ with
  $\{(r_{i,0}, h^{(1)}_{r_{i,0}}), (r^2_{i,0}, h^{(1)}_{r^2_{i,0}}), (r_{i,1}, h^{(1)}_{r_{i,1}}), (r^2_{i,1}, h^{(1)}_{r^2_{i,1}})\}_{i\in[n]}$
Initialize $j = 1, q = 1$
return $\mathcal{A}^{\mathcal{O}_{\mathsf{KeyGen}}, \mathcal{O}_{\mathsf{Enc}}(\cdot)}(1^\lambda, \mathsf{pk})$

$\underline{\mathcal{O}_{\mathsf{KeyGen}}}$
Sample $\{a_{i,b,j}\}_{i\in[n], b\in\{0,1\}}$
Add to table $\mathcal{T}$
  $\{(a_{i,1,j}, h^{(2)}_{a_{i,1,j}}), (a_{i,2,j}, h^{(2)}_{a_{i,2,j}})\}_{i\in[n]}$
j++
return $\{h^{(2)}_{a_{i,1,j}}, h^{(2)}_{a_{i,2,j}}\}_{i\in[n]}$

$\underline{\mathcal{O}_{\mathsf{Enc}}(\mathsf{j})}$
Let $e_{j,q}(\mathcal{E}) = \sum_{i\in[n], d\in\{1,2\}} a_{i,d,j}\mathbf{e}_{i,d,q}$
Add to table $\mathcal{T}$
  $\{(\mathbf{e}_{i,2,q}, h^{(1)}_{\mathbf{e}_{i,2,q}}), (\mathbf{e}_{i,1,q}, h^{(1)}_{\mathbf{e}_{i,1,q}})\}_{i\in[n]}, (e_{j,q}(\mathcal{E}), h^{(1)}_{e_{j,q}(\mathcal{E})})$
q++
return $\{h^{(1)}_{\mathbf{e}_{i,2,q}}, h^{(1)}_{\mathbf{e}_{i,1,q}}\}_{i\in[n]}, h^{(1)}_{e_{j,q}(\mathcal{E})}$

Lemma 5.2.12. *Sim-KeyGen-J is perfectly indistinguishable from Sim-Ideal*

Proof. This follows immediately by considering

$$\begin{bmatrix} a_{i,2,j} \\ a_{i,1,j} \end{bmatrix} = V_i^{-1} \cdot \begin{bmatrix} q_{i,0,j} \\ q_{i,1,j} \end{bmatrix}$$

for any $i \in [n], j \in [J]$ and noting that V_i is full rank, so even conditioned on the entries of V_i, the $a_{i,2,j}$ and $a_{i,1,j}$ values are uniformly random and independent of the entries of V_i if the $q_{i,0,j}$ and $q_{i,1,j}$ values are. □

This completes the proof of enhanced message privacy by summing the advantages in each lemma, since the oracles in Sim-Ideal are implemented independently of the master secret key. □

Bibliography

[AAB+13] Shashank Agrawal, Shweta Agrawal, Saikrishna Badrinarayanan, Abishek Kumarasubramanian, Manoj Prabhakaran, and Amit Sahai. Functional encryption and property preserving encryption: New definitions and positive results. Cryptology ePrint Archive, Report 2013/744, 2013. http://eprint.iacr.org/2013/744.

[AAB+15] Shashank Agrawal, Shweta Agrawal, Saikrishna Badrinarayanan, Abishek Kumarasubramanian, Manoj Prabhakaran, and Amit Sahai. On the practical security of inner product functional encryption. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 777–798. Springer, Heidelberg, March / April 2015.

[AB15] Benny Applebaum and Zvika Brakerski. Obfuscating circuits via composite-order graded encoding. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 528–556. Springer, Heidelberg, March 2015.

[ABD16] Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 153–178. Springer, Heidelberg, August 2016.

[ACLL15] Martin R. Albrecht, Catalin Cocis, Fabien Laguillaumie, and Adeline Langlois. Implementing candidate graded encoding schemes from ideal lattices. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 752–775. Springer, Heidelberg, November / December 2015.

[AGIS14] Prabhanjan Vijendra Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai. Optimizing obfuscation: Avoiding Barrington's theorem. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 646–658. ACM Press, November 2014.

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 191–219. Springer, Heidelberg, December 2016.

[Agr18] Shweta Agrawal. New methods for indistinguishability obfuscation: Bootstrapping and instantiation. Cryptology ePrint Archive, Report 2018/633, 2018. https://eprint.iacr.org/2018/633.

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

[AJ15] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 308–326. Springer, Heidelberg, August 2015.

[AJKS18] Prabhanjan Ananth, Aayush Jain, Dakshita Khurana, and Amit Sahai. Indistinguishability obfuscation without multilinear maps: io from lwe, bilinear maps, and weak pseudorandomness. Cryptology ePrint Archive, Report 2018/615, 2018. https://eprint.iacr.org/2018/615.

[AL16] Benny Applebaum and Shachar Lovett. Algebraic attacks against random local functions and their countermeasures. In Daniel Wichs and Yishay Mansour, editors, *48th ACM STOC*, pages 1087–1100. ACM Press, June 2016.

[App14] Benny Applebaum. Bootstrapping obfuscators via fast pseudorandom functions. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 162–172. Springer, Heidelberg, December 2014.

[AR16] Benny Applebaum and Pavel Raykov. Fast pseudorandom functions based on expander graphs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 27–56. Springer, Heidelberg, October / November 2016.

[Arx17] Arxan | Obfuscation, 2017. https://www.arxan.com/technology/obfuscation/.

[AS17] Prabhanjan Ananth and Amit Sahai. Projective arithmetic functional encryption and indistinguishability obfuscation from degree-5 multilinear maps. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 152–181. Springer, Heidelberg, April / May 2017.

[BBKK17] Boaz Barak, Zvika Brakerski, Ilan Komargodski, and Pravesh K. Kothari. Limits on low-degree pseudorandom generators (or: Sum-of-squares meets program obfuscation). Cryptology ePrint Archive, Report 2017/312, 2017. http://eprint.iacr.org/2017/312.

[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 Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

[BGK+14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 221–238. Springer, Heidelberg, May 2014.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[BHLN15] Daniel J. Bernstein, Andreas Hülsing, Tanja Lange, and Ruben Niederhagen. Bad directions in cryptographic hash functions. In Ernest Foo and Douglas Stebila, editors, *ACISP 15*, volume 9144 of *LNCS*, pages 488–508. Springer, Heidelberg, June / July 2015.

[BHWK16] Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. Compiling low depth circuits for practical secure computation. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 80–98. Springer, Heidelberg, September 2016.

[BJK15] Allison Bishop, Abhishek Jain, and Lucas Kowalczyk. Function-hiding inner product encryption. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 470–491. Springer, Heidelberg, November / December 2015.

[BKM+18] Allison Bishop, Lucas Kowalczyk, Tal Malkin, Valerio Pastro, Mariana Raykova, and Kevin Shi. A simple obfuscation scheme for pattern-matching with wildcards. Cryptology ePrint Archive, Report 2018/210, 2018. https://eprint.iacr.org/2018/210.

[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 Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 563–594. Springer, Heidelberg, April 2015.

[BM10] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, 2010.

[BM18] James Bartusek and Fermi Ma. Where the wildcards are, 2018.

[BR13] Zvika Brakerski and Guy N. Rothblum. Obfuscating conjunctions. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 416–434. Springer, Heidelberg, August 2013.

[BR14] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 1–25. Springer, Heidelberg, February 2014.

[BRS13a] Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private identity-based encryption: Hiding the function in functional encryption. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 461–478. Springer, Heidelberg, August 2013.

[BRS13b] Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private subspace-membership encryption and its applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part I*, volume 8269 of *LNCS*, pages 255–275. Springer, Heidelberg, December 2013.

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

[BS15] Zvika Brakerski and Gil Segev. Function-private functional encryption in the private-key setting. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 306–324. Springer, Heidelberg, March 2015.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 253–273. Springer, Heidelberg, March 2011.

[BV15] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In Venkatesan Guruswami, editor, *56th FOCS*, pages 171–190. IEEE Computer Society Press, October 2015.

[BVWW16] Zvika Brakerski, Vinod Vaikuntanathan, Hoeteck Wee, and Daniel Wichs. Obfuscating conjunctions under entropic ring LWE. In Madhu Sudan, editor, *ITCS 2016*, pages 147–156. ACM, January 2016.

[BWZ14] Dan Boneh, David J. Wu, and Joe Zimmerman. Immunizing multilinear maps against zeroizing attacks. Cryptology ePrint Archive, Report 2014/930, 2014. http://eprint.iacr.org/2014/930.

[BZ14] Dan Boneh and Mark Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 480–499. Springer, Heidelberg, August 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 Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 247–266. Springer, Heidelberg, August 2015.

[CHL+15] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 3–12. Springer, Heidelberg, April 2015.

[CJL16] Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An algorithm for NTRU problems and cryptanalysis of the GGH multilinear map without a low level encoding of zero. Cryptology ePrint Archive, Report 2016/139, 2016. http://eprint.iacr.org/2016/139.

[CLLT17] Jean-Sébastien Coron, Moon Sung Lee, Tancrède Lepoint, and Mehdi Tibouchi. Zeroizing attacks on indistinguishability obfuscation over CLT13. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 41–58. Springer, Heidelberg, March 2017.

[CLT13] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 476–493. Springer, Heidelberg, August 2013.

[CLT15] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. New multilinear maps over the integers. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 267–286. Springer, Heidelberg, August 2015.

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

[EHK+13] Alex Escala, Gottfried Herold, Eike Kiltz, Carla Ràfols, and Jorge Villar. An algebraic framework for Diffie-Hellman assumptions. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 129–147. Springer, Heidelberg, August 2013.

[Gal16] Galois, Inc. SAW: The software analysis workbench, 2016.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford University, 2009.

[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 Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 578–602. Springer, Heidelberg, May 2014.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 1–17. Springer, Heidelberg, May 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 *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

[GGH15] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 498–527. Springer, Heidelberg, March 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. http://eprint.iacr.org/2014/622.

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th FOCS*, pages 464–479. IEEE Computer Society Press, October 1984.

[GJ18] Craig Gentry and Charanjit S. Jutla. Obfuscation using tensor products. Cryptology ePrint Archive, Report 2018/756, 2018. https://eprint.iacr.org/2018/756.

[GLW14] Craig Gentry, Allison B. Lewko, and Brent Waters. Witness encryption from instance independent assumptions. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 426–443. Springer, Heidelberg, August 2014.

[GMM+16] Sanjam Garg, Eric Miles, Pratyay Mukherjee, Amit Sahai, Akshayaram Srinivasan, and Mark Zhandry. Secure obfuscation in a weak multilinear map model. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 241–268. Springer, Heidelberg, October / November 2016.

[Gol00] Oded Goldreich. Candidate one-way functions based on expander graphs. Cryptology ePrint Archive, Report 2000/063, 2000. http://eprint.iacr.org/2000/063.

[GPRW18] Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang. How to reveal the secrets of an obscure white-box implementation. Cryptology ePrint Archive, Report 2018/098, 2018. https://eprint.iacr.org/2018/098.

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 772–783. ACM Press, October 2012.

[HHSSD17] Shai Halevi, Tzipora Halevi, Victor Shoup, and Noah Stephens-Davidowitz. Implementing BP-obfuscation using graph-induced encoding. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 783–798. ACM Press, October / November 2017.

[HJ16] Yupu Hu and Huiwen Jia. Cryptanalysis of GGH map. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 537–565. Springer, Heidelberg, May 2016.

[ITZ16] Vincenzo Iovino, Qiang Tang, and Karol Zebrowski. On the power of public-key function-private functional encryption. In Sara Foresti and Giuseppe Persiano, editors, *CANS 16*, volume 10052 of *LNCS*, pages 585–593. Springer, Heidelberg, November 2016.

[Jav17] Excelsior | Java code obfuscators, 2017. https://www.excelsior-usa.com/articles/java-obfuscators.html.

[KKS17] Sungwook Kim, Jinsu Kim, and Jae Hong Seo. A new approach for practical function-private inner product encryption. Cryptology ePrint Archive, Report 2017/004, 2017. http://eprint.iacr.org/2017/004.

[KLM+16] Sam Kim, Kevin Lewi, Avradip Mandal, Hart Montgomery, Arnab Roy, and David J. Wu. Function-hiding inner product encryption is practical. Cryptology ePrint Archive, Report 2016/440, 2016. http://eprint.iacr.org/2016/440.

[KSW08] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 146–162. Springer, Heidelberg, April 2008.

[Lin16] Huijia Lin. Indistinguishability obfuscation from constant-degree graded encoding schemes. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 28–57. Springer, Heidelberg, May 2016.

[Lin17] Huijia Lin. Indistinguishability obfuscation from SXDH on 5-linear maps and locality-5 PRGs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 599–629. Springer, Heidelberg, August 2017.

[LM18] Huijia Lin and Christian Matt. Pseudo flawed-smudging generators and their application to indistinguishability obfuscation. Cryptology ePrint Archive, Report 2018/646, 2018. https://eprint.iacr.org/2018/646.

[LMA+16] Kevin Lewi, Alex J. Malozemoff, Daniel Apon, Brent Carmer, Adam Foltzer, Daniel Wagner, David W. Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova. 5Gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 981–992. ACM Press, October 2016.

[LPST16] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Output-compressing randomized encodings and applications. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 96–124. Springer, Heidelberg, January 2016.

[LSS14] Adeline Langlois, Damien Stehlé, and Ron Steinfeld. GGHLite: More efficient multilinear maps from ideal lattices. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 239–256. Springer, Heidelberg, May 2014.

[LT17] Huijia Lin and Stefano Tessaro. Indistinguishability obfuscation from trilinear maps and block-wise local PRGs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 630–660. Springer, Heidelberg, August 2017.

[LV16] Huijia Lin and Vinod Vaikuntanathan. Indistinguishability obfuscation from DDH-like assumptions on constant-degree graded encodings. In Irit Dinur, editor, *57th FOCS*, pages 11–20. IEEE Computer Society Press, October 2016.

[LW16] Kevin Lewi and David J. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 1167–1178. ACM Press, October 2016.

[MGC+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *EuroS&P*, 2016.

[MSZ16] Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In Matthew

Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 629–658. Springer, Heidelberg, August 2016.

[O'N10] Adam O'Neill. Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556, 2010. http://eprint.iacr.org/2010/556.

[OW14] Ryan O'Donnell and David Witmer. Goldreich's PRG: Evidence for near-optimal polynomial stretch. In *CCC*, 2014.

[PM18] Sikhar Patranabis and Debdeep Mukhopadhyay. New lower bounds on predicate entropy for function private public-key predicate encryption. Cryptology ePrint Archive, Report 2018/190, 2018. https://eprint.iacr.org/2018/190.

[PS16] Rafael Pass and Abhi Shelat. Impossibility of VBB obfuscation with ideal constant-degree graded encodings. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 3–17. Springer, Heidelberg, January 2016.

[Sch80] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *46th ACM STOC*, pages 475–484. ACM Press, May / June 2014.

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

[Thi17] Semantic Designs | Thicket™ family of source code obfuscators, 2017. http://www.semdesigns.com/Products/Obfuscators/.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 24–43. Springer, Heidelberg, May / June 2010.

[WGK13] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys - a free Verilog synthesis suite. In *Austrochip*, 2013.

[WMK17] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 399–424. Springer, Heidelberg, April / May 2017.

[WZ17] Daniel Wichs and Giorgos Zirdelis. Obfuscating compute-and-compare programs under LWE. In *58th FOCS*, pages 600–611. IEEE Computer Society Press, 2017.

[Zim15] Joe Zimmerman. How to obfuscate programs directly. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 439–467. Springer, Heidelberg, April 2015.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Symbolic and algebraic computation*, pages 216–226. Springer, 1979.

APPENDIX

A Circuit Information

In Table A.1, we list the circuits we consider in Chapter 4, as well as attributes about those circuits relevant to obfuscation. We describe each circuit below.

- `aes1r`: One-round AES.
- `aes1r_`x`_`y: One-round AES with x input bits and y output bits.
- `ggm_`x`_`y: The GGM PRF using x applications of Goldreich's PRG, using the `xor-and` predicate, and y bits of output.
- `ggm_sigma_`x`_`y: The GGM PRF using Σ-vectors with x applications of Goldreich's PRG, using the `xor-and` predicate, and y bits of output.

Circuit	n	m	# Gates	# Muls	Depth	Degree	κ
aes1r*	128	128	80,564	9,203	25	33	128
aes1r_2_1**	2	1	4	1	3	2	3
aes1r_4_1**	4	1	44	17	7	5	6
aes1r_8_1**	8	1	1,282	389	15	9	10
aes1r_16_1**	16	1	1,282	389	15	9	16
aes1r_32_1**	32	1	1,282	389	15	9	32
aes1r_64_1*	64	1	1,324	614	23	18	64
aes1r_128_1*	128	1	1,951	967	23	33	128
ggm_1_32	4	32	7,031	2,212	12	9	14
ggm_1_64	4	64	14,453	4,690	12	9	14
ggm_1_128	4	128	29,696	9,775	12	9	14
ggm_2_32	8	32	13,647	4,348	24	49	74
ggm_2_64	8	64	28,727	9,315	24	49	74
ggm_2_128	8	128	59,247	19,519	24	49	74
ggm_3_32	12	32	20,188	6,391	36	249	374
ggm_3_64	12	64	43,063	14,002	36	249	374
ggm_3_128	12	128	88,905	29,244	36	249	374
ggm_4_32	16	32	27,102	8,570	48	1,249	1,874
ggm_4_64	16	64	57,538	18,600	48	1,249	1,874
ggm_4_128	16	128	118,315	38,916	48	1,249	1,874
ggm_sigma_1_16_32	16	4	937	310	12	6	7
ggm_sigma_1_16_64	16	4	954	318	12	6	7
ggm_sigma_1_16_128	16	4	956	320	12	6	7
ggm_sigma_1_32_32	32	5	2,326	758	13	6	7
ggm_sigma_1_32_64	32	5	2,375	786	13	6	7
ggm_sigma_1_32_128	32	5	2,393	798	13	6	7
ggm_sigma_1_64_32	64	6	5,309	1,676	14	6	7
ggm_sigma_1_64_64	64	6	5,629	1,855	14	6	7
ggm_sigma_1_64_128	64	6	5,726	1,904	14	6	7
ggm_sigma_1_256_32	256	8	23,710	7,021	16	6	7
ggm_sigma_1_256_64	256	8	27,692	8,666	16	6	7
ggm_sigma_1_256_128	256	8	29,889	9,780	16	6	7
ggm_sigma_2_16_32	32	8	8,580	2,730	24	31	33
ggm_sigma_2_16_64	32	8	16,138	5,224	24	31	33
ggm_sigma_2_16_128	32	8	31,431	10,332	24	31	33
ggm_sigma_2_32_32	64	10	17,022	5,227	26	31	33
ggm_sigma_2_32_64	64	10	31,997	10,099	26	31	33
ggm_sigma_2_32_128	64	10	62,416	20,171	26	31	33
ggm_sigma_2_64_32	128	12	32,998	9,913	28	31	33
ggm_sigma_2_64_64	128	12	62,227	19,090	28	31	33
ggm_sigma_2_64_128	128	12	121,798	38,383	28	31	33
ggm_sigma_3_16_32	48	12	16,173	5,064	36	156	161
ggm_sigma_3_16_64	48	12	31,259	10,085	36	156	161
ggm_sigma_3_16_128	48	12	61,989	20,400	36	156	161
ggm_sigma_4_16_32	64	16	23,682	7,461	48	781	794
ggm_sigma_4_16_64	64	16	46,418	15,050	48	781	794
ggm_sigma_4_16_128	64	16	92,394	30,324	48	781	794

Table A.1: Circuits and their associated attributes. All of these circuits were compiled with constant folding (-O1), with * denoting those run through the sub-circuit flattener optimization (-O2) and ** denoting those run through the full-circuit flattener (-O3). 'n' denotes the number of input bits; 'm' denotes the number of output bits; '# Gates' denotes the total number of gates; '# Muls' denotes the number of multiplication gates; 'Depth' denotes the multiplicative depth of the circuit; 'Degree' denotes the multiplicative degree of the circuit; and 'κ' denotes the multilinearity value computed using MIO.