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

Brent Carmer
Oregon State University
Galois, Inc.
bcarmer@galois.com

Alex J. Malozemoff
Galois, Inc.
amaloz@galois.com

Mariana Raykova
Yale University
mariana.raykova@yale.edu

Abstract

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 work, we explore in detail cryptographic program obfuscation and the related primitive of multi-input functional encryption (MIFE). In particular, we extend the 5Gen framework (CCS 2016) 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 work 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.

1 Introduction

The goal of program obfuscation is to hide secrets in the implementation of a program, where the secrets can be used to evaluate the program but are guaranteed to remain hidden beyond what can be deduced from the output. Program obfuscation is a powerful technique with a wide range of applications, as is evident from the large number of (heuristic) obfuscators used in practice [e.g., 7, 1, 5].

One of the most prominent applications of program obfuscation is *white-box cryptography* [23], where the goal is to hide the key of a pseudorandom function (PRF), but nothing else. Payment processing, digital rights management, and many other systems use white-box cryptography to secure data, with a large number of companies offering such products [e.g., 4, 6, 2]. Unfortunately, all white-box cryptography schemes have been broken, and thus there is a need for constructions based on a rigorous *cryptographic* foundation. In a breakthrough result, Garg et al. [29] presented

the first such candidate construction for general program obfuscation, spawning a large amount of followup work in both the underlying security claims and the overall efficiency.

A closely related primitive to program obfuscation is multi-input functional encryption (MIFE), where decryption takes a key associated with an n-input function F, as well as n independently generated ciphertexts $\mathsf{Enc}(x_1), \ldots, \mathsf{Enc}(x_n)$, and outputs $\mathsf{F}(x_1, \ldots, x_n)$. Program obfuscation and MIFE are known to imply one another [33].

All existing constructions of (cryptographic) program obfuscation and MIFE use multilinear maps (mmaps) [20], which extend the idea of bilinear maps to arbitrary polynomials. In order to build applications on top of and experiment with mmaps, Lewi et al. [36] introduced the 5Gen framework. This framework implements two mmap constructions [26, 28] and provides implementations of MIFE and program obfuscation based on *(matrix) branching programs*, which provide a way to view arbitrary functions as sequences of matrix multiplications. For simple functions, this approach is sufficient. However, the mapping from function to branching program is exponential, and thus this approach quickly becomes infeasible as function complexity grows. Due to this blow-up, Lewi et al. were limited to obfuscating an 80-bit point function.

However, branching programs are not the only approach for realizing program obfuscation. Both Zimmerman [49] and Applebaum and Brakerski [12] 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. When Lewi et al. [36] implemented the Zimmerman construction, they found the branching program approach superior for the simple functions they considered (e.g., 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 become more efficient. Thus, extending the 5Gen framework with circuit obfuscators — and also developing MIFE for circuits — would substantially enhance the framework, provide a basis for comparison, and push (cryptographic) program obfuscation further towards practicality.

1.1 Our Contributions

In this work, 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 introduced by Lewi et al. [36], 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 [33], this gives us an obfuscation scheme which avoids the extra n multilinearity overhead inherent in prior circuit obfuscators [49, 12], 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 [12] for low-depth PRFs by Lin [38] and our own adaptation of this optimization to Zimmerman's scheme [49].

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.

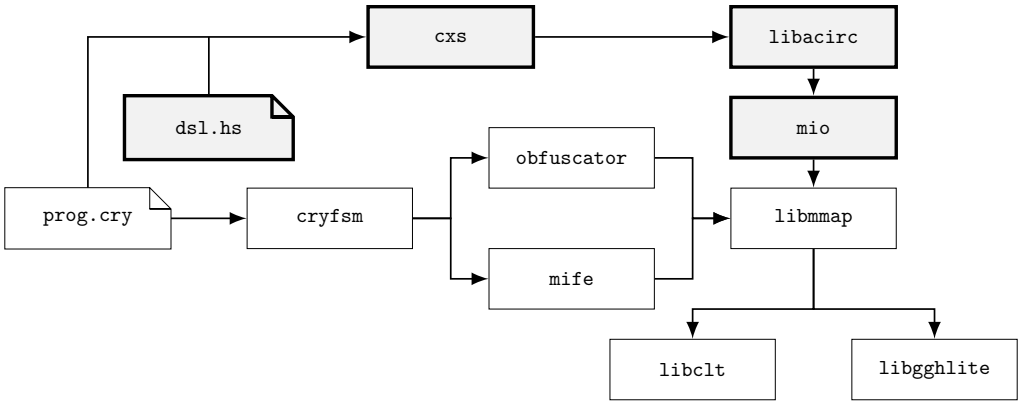

Figure 1.1: The 5Gen framework architecture, with new components introduced in this work in bold and gray boxes. The original 5Gen architecture takes as input a cryptol program (here given by `prog.cry`), which is fed into a compiler, `cryfsm`, to produce a matrix branching program. This is then fed 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.

Finally, given our implementation and compiler suite, we experiment with obfuscating a PRF. 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.

To summarize, our contributions are as follows:

1. An instantiation of multi-input functional encryption (MIFE) for circuits (cf. §5). Prior MIFE constructions [19, 36] 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. [33] 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. §6). 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. §7). As part of this, we introduce three new components to the 5Gen framework introduced by Lewi et al. [36]: (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 1.1 for the enhanced 5Gen architecture.

4. A thorough exploration of the performance of the various obfuscators, with a focus on obfuscating a PRF (cf. §8).

In addition, in §3 we review circuit obfuscation and compare it to the approach using constant-degree mmaps, and in §4 we review existing circuit obfuscators.

As part of our exploration, we developed a modification of Zimmerman's construction [49] that adapted ideas from Lin [38] to support "Σ-vectors", which are used for more efficient obfuscation of low depth PRFs (cf. §4). While this new construction performed significantly better than all existing obfuscators for a specific class of GGM PRFs, we found that our MIFE transformation described in §5 resulted in even better performance, and thus focus on that construction throughout this paper. However, for completeness we describe our "Linnerman" construction in Appendix B.

1.2 Why PRFs?

One of the most popular algorithms to obfuscate in practice has been AES, due to its widespread use in payment systems, digital rights management, etc. While there have been a plethora of constructions for obfuscating AES, the majority of them have been quickly broken, and for the rest of them it remains unclear what security guarantees they provide. Indeed, there is an ongoing competition for a secure white-box implementation of AES-128 [3]. Therefore, cryptographic obfuscation of a keyed PRF would have a significant impact.

Obfuscation of a PRF is also a central building block in "bootstrapping" program obfuscation for larger function classes [11, 38]. Another interesting property related to PRF obfuscation is that existing attacks on obfuscation constructions leveraging weaknesses in the underlying mmaps are not known to break obfuscations of PRFs[1]. In fact, the most recent obfuscation schemes that provide security against all known attacks embed a PRF that is evaluated in parallel with the obfuscated function, and leverage the PRF hardness to prove security [30].

In light of these facts, we view PRFs as both a primary candidate for which we should aim to obtain efficient obfuscation and a useful benchmark for evaluating the efficiency of new approaches.

2 Preliminaries

We let λ denote the security parameter, and let κ denote the multilinearity of the underlying multilinear map (mmap) used in our constructions. In this section we review (composite-order) mmaps (§2.1), multi-input functional encryption (§2.2), and program obfuscation (§2.3).

2.1 Composite-order Multilinear Maps

An mmap provides 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. This can be thought of in some sense as fully homomorphic encryption with a "broken" decryption procedure that allows *anyone* to test the top-level value for zero. In particular, mmaps provide an Encode operation that maps a value into its encoded form, and Add and Mult operations that allow adding and multiplying encoded values. At a certain point, the $\mathsf{ZeroTest}$ operation can be run

[1] Such attacks rely on specific properties of the obfuscated function and do not work for all functions.

```
deg(g : Gate):
  if g = ADD(x, y)
    return max(deg(x), deg(y))
  if g = MUL(x, y)
    return deg(x) + deg(y)
  else:
    return 1
```

Figure 2.1: 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.

to test equality with zero. *Composite-order* mmaps allow using multiple *slots* of encoded values, where now $\mathsf{ZeroTest}$ outputs zero if and only if *all* values in *all* slots are zero.

Most 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 mmap generic model is a slight strengthening of this to allow encoding values across all of the mmap slots. We define this formally below.

Definition 2.1. The *composite-order mmap generic model* [49] is defined by the operations Setup, Encode, Add, Mult, and $\mathsf{ZeroTest}$, 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 "encoding" $[x_1, \ldots, x_N]_{\mathcal{S}}$, 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 $h \mapsto (x_{1,1} + x_{2,1}, \ldots, x_{1,N} + x_{2,N}, \mathcal{S}_1)$ to T, returning h; otherwise, return $\perp$.
- $\mathsf{Mult}(\mathsf{pp}, h_1, h_2) \to \{h, \perp\}$: 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 $h \mapsto (x_{1,1} \cdot x_{2,1}, \ldots, x_{1,N} \cdot x_{2,N}, \mathcal{S}_1 \cup \mathcal{S}_2)$ 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 $h \mapsto (x, \mathcal{S})$ 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 $\perp$.

The *multilinearity*, κ, of the mmap is defined as the multiplicative degree (defined by Figure 2.1) required to reach the top-level index set.

2.2 Multi-input Functional Encryption

Multi-input functional encryption (MIFE) [33] 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. [19].

Definition 2.2. 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:

- $\mathsf{1SK\text{-}MIFE.Setup}(\lambda, C) \to (\mathsf{sk}, \mathsf{ek})$: 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.
- $\mathsf{1SK\text{-}MIFE.Enc}(\mathsf{sk}, i, \mathbf{x}) \to \mathsf{ct}$: Takes as input secret key sk, index i, and input $\mathbf{x}$, and outputs a ciphertext ct.
- $\mathsf{1SK\text{-}MIFE.Dec}(\mathsf{ek}, \mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(n)}) \to \mathbf{y}$: Takes as input evaluation key ek and ciphertexts $\mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(n)}$, and outputs a bitstring $\mathbf{y}$.

Definition 2.3 (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 $(\mathsf{ek},\ \mathsf{sk}) \leftarrow \mathsf{1SK\text{-}MIFE.Setup}(1^\lambda, C)$ and for each $i \in [n]$, $\mathsf{ct}^{(i)} \leftarrow \mathsf{1SK\text{-}MIFE.Enc}(\mathsf{sk}, i, x^{(i)})$, then,

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

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:

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

1. Compute $(\mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{1SK\text{-}MIFE.Setup}(1^\lambda, C)$ 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 mmap generic model. For this we use the notion of *admissible execution traces* [19]. 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 any tuple of queries $\{(i_{q_j}, x_{q_j,0}, x_{q_j,1})\}_{j\in[n]}$ where $i_{q_j} = j$ for all $j \in [n]$, we have that $C(x_{q_1,0}, \ldots, x_{q_n,0}) = C(x_{q_1,1}, \ldots, x_{q_n,1})$.

Definition 2.4 (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

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

2.3 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 2.5. A *program obfuscator* is a tuple of algorithms $(\mathsf{Obfuscate}, \mathsf{Evaluate})$ defined as follows:

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

Definition 2.6 (Correctness)**.** A program obfuscator is *correct* for circuit C if for all $\mathbf{x} \in \{0,1\}^n$, it holds that

$$\mathsf{Evaluate}(\mathsf{Obfuscate}(\lambda, C), \mathbf{x}) = C(\mathbf{x}).$$

Definition 2.7 (Efficiency)**.** A program obfuscator is *efficient* for circuit C if there exists polynomial $p(\cdot)$ such that $|\mathsf{Obfuscate}(\lambda, C)| < p(\lambda)$.

Regarding security, the two key notions are *virtual black-box (VBB) obfuscation* and *indistinguishability obfuscation*. While all of our constructions are secure for the weaker indistinguishability notion, we heuristically assume that they provide VBB security.[2]

Definition 2.8. A program obfuscator is an *indistinguishability obfuscator* for circuit class $\{\mathcal{C}_\lambda\}$ if for all $\lambda \in \mathbb{N}$ and for every pair of circuits $C_0, C_1 \in C_\lambda$ such that $|C_0| = |C_1|$ and $C_0(\mathbf{x}) = C_1(\mathbf{x})$ for all inputs $\mathbf{x}$, then

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

Definition 2.9. A program obfuscator 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

$$\Pr[\mathcal{A}(\mathsf{Obfuscate}(\lambda, C)) = 1] \approx \Pr[\mathcal{S}^C(\lambda) = 1],$$

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

3 Overview of Existing Techniques

In this section, we briefly describe the high-level idea behind circuit obfuscation (§3.1) and compare it with the approach of obfuscation from constant-degree mmaps (§3.2).

[2] Although VBB obfuscation is impossible in general [17], there is no attack that we are aware of that breaks indistinguishability obfuscators when viewed (heuristically) as VBB obfuscators for the functions we consider.

3.1 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. 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 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 will be in order to generate encodings with the appropriate noise tolerance.

3.2 Comparison with Obfuscation from Constant-degree Multilinear Maps

Starting with the work of Lin [38], a recent and concurrent line of work has looked at building obfuscation from *constant-degree* mmaps [42, 10, 39, 41]. All of these approaches utilize the "bootstrapping" approach for building obfuscation from (succinct) functional encryption (FE) [18, 9]; 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 [18], who build obfuscation from succinct *public-key* FE (the approaches of Ananth and Jain [9] and Lin et al. [40] 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 [47]. The best current FE construction uses trilinear maps [41], and thus the circuit representations would have to include such 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 [45]). However, efficiency improvements to either the bootstrapping step or the underlying FE primitive could change this, and are interesting open problems.

4 Circuit Obfuscators

In this section we review the three existing circuit obfuscators in the literature: the Zimmerman obfuscator [49], denoted by Zim, the Applebaum-Brakerski obfuscator [12], denoted by AB, and the Lin obfuscator [38], denoted by Lin.

4.1 Obfuscation using Zim

In Zim, each "public" input bit x_i is encoded under symbol $X_{i,b}$; namely, the obfuscation contains 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 one 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.

	AB/Lin		Zim/LZ		MO	
Circuit	# Enc	κ	# Enc	κ	# Enc	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 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.

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 has completely disjoint indices.

Computing κ. Let n be the number of inputs to the circuit C, and let $\deg(\cdot)$ 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

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

Results. See Table 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× improvement.

4.2 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 [38] 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 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.

4.3 Obfuscation using Lin

In a breakthrough result, Lin showed how to construct obfuscation from *constant-degree* mmaps [38]. 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 [32], which constructs a PRF directly from a PRG. While the particular details for now are not important (see §8 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:

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

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 $16 = 4 \cdot \log_2(16) = 2 \cdot \log_2(256)$).

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

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 $x \in \{0,1\}^{\ell n}$. Then, to use e to select a sub-string, compute

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

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 1. However, for certain functions, using Σ-vectors can have a huge improvement. In Table 2 we show the effect on κ when using Σ-vectors for the GGM PRF (cf. §8.2). 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.

Basing Lin on Zim. As Lin builds on AB, which has much worse multilinearity than Zim, this leaves open the question whether the techniques in Lin could be applied directly to Zim. In Appendix B, we present the "Linnerman" obfuscator, denoted by LZ, which does exactly this. And we indeed realize the expected improvement. For example, obfuscating the GGM PRF with 16 "real" input bits and a 128-bit key results in $\kappa_{\mathsf{Lin}} = 8{,}163$ versus $\kappa_{\mathsf{LZ}} = 797$, a ten-fold improvement (cf. Table 2). Looking ahead, however, we find that our MIFE-based construction presented in §5 results in even *better* multilinearity than our "Linnerman" construction.

5 Multi-input Functional Encryption from Circuits

In this section we present our new construction for (single-key) multi-input functional encryption (MIFE). We begin with some high-level intuition before describing the scheme in detail in §5.1. In §5.2 we present a security proof, in §5.3 we describe some optimizations, and in §5.4 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. [19], 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 [49].

Program obfuscators provide the capability to evaluate the (obfuscated) function on any possible input. Thus, existing obfuscators (both for branching programs and circuits) 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 §3 and §4, 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 $m = 1$ 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 $\hat{w}_i$ values are multiplied in to reach the zero-testing level and then subtracted, 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 [26] 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. §8) requires 13,111 slots, well above the number of inputs of these two circuits.

5.1 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 $\deg(\mathbf{x}^{(i)})$ denote the maximum degree of the bits of the ith MIFE slot across all output bits, and let $\deg(\mathbf{x}_o^{(i)})$ denote the maximum degree of the bits of the ith MIFE slot for output bit $o\in[m]$. Our 1SK-MIFE construction works as follows:

$\underline{\mathsf{1SK\text{-}MIFE.Setup}(1^\lambda, C)}$:

1. Define top-level index set
$$\mathcal{U} := Z\prod_{i\in[n]} W^{(i)}(X^{(i)})^{\deg(\mathbf{x}^{(i)})}.$$

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

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

4. For $i\in[n]$, generate the following encoding:
$$\hat{u}_i := [1,\underbrace{1,\ldots,1}_{n}]_{X^{(i)}}.$$

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

$$\hat{z}_o := [\delta_o, \underbrace{1, \ldots, 1}_{n}]_{Z \prod_{i \in [n]} W^{(i)} (X^{(i)})^{\deg(\mathbf{x}^{(i)}) - \deg(\mathbf{x}_o^{(i)})}},$$

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

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

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

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

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

1. Parse ek as $\left(\mathsf{pp}, C, \{\hat{z}_o\}_{o \in [m]}, \{\hat{u}_i\}_{i \in [n]}, \hat{C}^*\right)$.

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 Zimmerman [49, §3]). 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

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

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

5.2 Proof of Security

Theorem 5.1. *The construction in §5.1 is correct according to Definition 2.3.*

Proof. Correctness follows directly by inspection. ■

Theorem 5.2. *The construction in §5.1 is secure according to Definition 2.4.*

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 $\mathsf{Expt}^{\mathsf{1SK\text{-}MIFE}}_{C,Q,0}(\mathcal{A})$ and $\mathsf{Expt}^{\mathsf{1SK\text{-}MIFE}}_{C,Q,1}(\mathcal{A})$ 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

$$\left(\prod_{j\in[d_i]} (\hat{x}_j^{(i)})^{m_i} (\hat{u}_j^{(i)})^{\deg(\hat{x}_j^{(i)})-m_i}\right) \hat{z} \tag{2}$$

   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 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 [49, 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 $j \in [2, \ldots, n+1]$. 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 $C^\dagger := C(1, \ldots, 1, \{\alpha_j\}_{j\in[d_i]}, 1, \ldots, 1)$ 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 $C(1, \ldots, 1, \{\hat{x}_j^{(i)}\}_{j\in[d_i]}, 1, \ldots, 1)\hat{z} - \hat{C}^* \hat{w}^{(i)} \cdot \underbrace{\star \cdots \star}$, 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.

However, 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. ■

5.3 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 $\hat{C}^* := [0, 1, \ldots, 1]_{Z \prod_{i \in [n+1]} (X^{(i)})^{\deg(\mathbf{x}^{(i)})}}$, we directly compute the combination of $\hat{C}^*$ and $\hat{w}_o^{(n+1)}$; that is, we replace $\hat{C}^*$ with $\hat{w}_o^{(n+1)} := [0, \underbrace{1, \ldots, 1}_{n}, C_o^\dagger]_{ZW^{(n+1)} \prod_{i \in [n+1]} (X^{(i)})^{\deg(\mathbf{x}^{(i)})}}$ 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:

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

5.4 Obfuscation from MIFE

Our MIFE construction immediately gives us an obfuscation scheme using the transformation of Goldwasser et al. [33]: 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 MO. Note that we can also directly support Σ-vectors by encrypting each $\sigma \in \Sigma$ for each MIFE slot.

$\mathsf{MO.Obfuscate}(1^\lambda, C)$:

1. Compute $(\mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{1SK\text{-}MIFE.Setup}(1^\lambda, C)$.
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

$$\left(\mathsf{ek}, \{\mathsf{ct}_{i,0}, \mathsf{ct}_{i,1}\}_{i \in [n]}\right).$$

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

1. Parse Obf as $\left(\mathsf{ek}, \{\mathsf{ct}_{i,0}, \mathsf{ct}_{i,1}\}_{i \in [n]}\right)$.
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 MO 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 1 for some examples; roughly, MO requires up to $2\times$ fewer encodings than all existing approaches, which directly impacts the obfuscation time and size.

Results. See Table 1 and Table 2 for results. We can see that MO is better than all existing schemes, both in terms of κ values as well as the number of encodings needed.

6 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 2.1). 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 for secure computation either minimize the number of gates [35, 43] or the depth [22]. As an early attempt, we took a similar approach using off-the-shelf tools: taking a Cryptol[3] function as input, we used the Software Analysis Workbench [27] to generate circuits composed of AND and NOT gates, optimized them using either ABC [21], a tool for synthesizing and optimizing binary sequential logic circuits, or Yosys [48], 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 §6.1 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 §6.2 we show how well the optimizations perform on a circuit that computes a single round of AES.

[3]Cryptol is a domain-specific language designed for writing programs over streams of bits [37].

6.1 Circuit Optimizations

In this section we describe each of our circuit optimization approaches: using off-the-shelf optimizers (§6.1.1), using a DSL (§6.1.2), encoding lookup tables (§6.1.3), folding constants (§6.1.4), and circuit flattening (§6.1.5).

6.1.1 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 [46] to use ABC [21] and Yosys [48], 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.

6.1.2 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.

6.1.3 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 $\left[\begin{array}{cccc} 0 & 1 & 1 & 0 \end{array}\right]$, 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.

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 3: Comparison of DSL optimizations. 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.

6.1.4 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 $\text{NOT}(x)$. Such gates also occur if we fix input bits to a constant.

6.1.5 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.

6.2 Optimization Results

See Table 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.

7 Implementation

We implemented the MIFE scheme described in §5 and all of the obfuscators discussed in §4, §5, and Appendix B. 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 [26] 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 [26]. Unfortunately, the CLT mmap is prone to several attacks. Most relevant to our constructions, Coron et al. [24] 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. [25] 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 [28] has been used to eliminate all known attacks [30], 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.

8 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. §8.1) and the Goldreich, Goldwasser, Micali (GGM) PRF (cf. §8.2). We explored a number of other PRF constructions, including MiMC [8] and the PRF of Applebaum and Raykov [14]. 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 §5.

8.1 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 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. §3.2).

8.2 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 [32] 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[4] 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 [38] used a variant of the GGM PRF in her construction of obfuscation from constant-degree mmaps (cf. §3.2). 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. §4.3) 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

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

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 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.

8.2.1 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 [31]: 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 [31] suggested choosing P at random. O'Donnell and Witmer [44] 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 [13] 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 for the results.

We found that all three predicates have roughly the same multilinearity, with `xor-and` and `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: $\mathrm{XOR}(x, y) := x + y - 2xy$, thus requiring one multiplication and three linear operations, whereas $\mathrm{AND}(x, y) := xy$, 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. [15], 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.

8.2.2 Implementing the GGM PRF

We implemented the GGM PRF within our DSL for both "standard" (boolean) inputs and Σ-vector inputs; see Table 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 (1)).

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: 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.

8.2.3 Performance Results

See Table 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[5]. 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. [36] obfuscated an 80-bit point function, and Halevi et al. [34] obfuscated a 100-state non-deterministic 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 $|\Sigma| = 256$ (resulting in one application of the PRG and thus $\kappa = 7$) versus $|\Sigma| = 16$ (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 $|\Sigma| = 256$. As an example, for key length 128 we need 67,872 encodings for

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

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

Table 5: Multilinearity values for the GGM PRF obfuscated using MO 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.

n	$\|\Sigma\|$	# PRGs	k	# Gates	# Enc	κ	Obf time	Obf size	Obf RAM	Eval time	Eval RAM
8	256	1	32	23,710	67,680	7	10 h	121 GB	268 GB	4.4 m	143 GB
8	256	1	64	27,692	67,744	7	10 h	121 GB	270 GB	4.7 m	155 GB
8	256	1	128	29,889	67,872	7	10 h	121 GB	268 GB	4.9 m	152 GB
8	16	2	32	8,580	872	33	127 m	11 GB	44 GB	11 m	15 GB
8	16	2	64	16,138	936	33	127 m	12 GB	41 GB	17 m	18 GB
8	16	2	128	31,431	1,064	33	130 m	13 GB	49 GB	29 m	21 GB
12	64	2	32	32,998	9,840	33	3.7 h	119 GB	267 GB	42 m	125 GB
12	64	2	64	62,227	9,904	33	3.7 h	120 GB	270 GB	67 m	128 GB
12	64	2	128	121,798	10,032	33	3.7 h	122 GB	274 GB	—	—

Table 6: Obfuscation details for various GGM PRF circuits. 'n' denotes both the number of "real" input bits (i.e., $n = \#\mathsf{PRGs} \cdot \log_2(|\Sigma|)$) and the number of output bits of the circuit; '$|\Sigma|$' 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; '# Enc' denotes the number of mmap encodings required for obfuscation; '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.

$|\Sigma| = 256$ versus 1,064 encodings for $|\Sigma| = 16$. 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 $|\Sigma| = 256$: 4.9 minutes versus 29 minutes using our previous example. This presents an interesting tradeoff of obfuscation time/size versus evaluation time.

9 Conclusion

In this work, 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.

Acknowledgments

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

References

[1] Arxan — Obfuscation, 2017. `https://www.arxan.com/technology/obfuscation/`.

[2] Arxan — White-box cryptography, 2017. `https://www.arxan.com/technology/white-box-cryptography/`.

[3] CHES 2017 capture the flag challenge — The WhibOx contest, 2017. `http://whibox.cr.yp.to/`.

[4] CryptoExperts — White-box cryptography, 2017. `https://www.cryptoexperts.com/technologies/white-box/`.

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

[6] Gemalto — White-box cryptography, 2017. `https://sentinel.gemalto.com/software-monetization/white-box-cryptography`.

[7] Semantic Designs — Thicket^{TM} family of source code obfuscators, 2017. `http://www.semdesigns.com/Products/Obfuscators/`.

[8] 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.

[9] 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.

[10] 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, May 2017.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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`.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In Kaisa Nyberg and Howard M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270. Springer, Heidelberg, August 2003.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, 1986.

[33] 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.

[34] Shai Halevi, Tzipora Halevi, Victor Shoup, and Noah Stephens-Davidowitz. Implementing BP-obfuscation using graph-induced encoding. In *ACM CCS 17*, 2017.

[35] 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.

[36] 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.

[37] Jeffrey R. Lewis and Brad Martin. Cryptol: High assurance, retargetable crypto development and validation. In *IEEE MILCOM'03*, volume 2, pages 820–825, 2003.

[38] 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.

[39] Huijia Lin. Indistinguishability obfuscation from SXDH on 5-linear maps and locality-5 PRGs. In *CRYPTO 2017*, 2017.

[40] 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.

[41] Huijia Lin and Stefano Tessaro. Indistinguishability obfuscation from trilinear maps and block-wise local PRGs. In *CRYPTO 2017*, 2017.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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 II*, volume 10211 of *LNCS*, pages 399–424. Springer, Heidelberg, May 2017.

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

[49] 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.

A Circuit Information

In Table 7, we list the circuits we consider in this work, 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.

B The Linnerman Obfuscator

In this section we introduce the "Linnerman" obfuscator, denoted by LZ, which combines Zim and Lin. Lin is based off AB, which as noted in §4 results in very large values for κ. We thus adapt Lin to be based off Zim, resulting in much smaller values for κ. Our approach is similar to that taken by Lin [38] (cf. §4.3) by adapting Zim to support Σ-vectors. In what follows, we give the high-level intuition of our construction; full details are in §B.1.

Supporting Σ-vectors. Recall from §4.3 that a Σ-vector is a symbol represented in unary as follows: if $s \in \Sigma$ represents the "ith" symbol, we encode s as a ℓ-length vector (where $\ell = |\Sigma|$) of all zeros except the ith, which is set to one. The main challenge is thus enforcing that an evaluator inputs s "as a whole", and cannot mix-and-match bits from s with bits from some other Σ-vector s'. For example, if $s := 0b01$ and $s' := 0b10$, we must enforce that the evaluator cannot input $s'' := 0b11$.

We do this by assigning an index set $S_{k,s}$ to each Σ-vector input $k \in [c]$ and each Σ-vector $s \in \Sigma$. Then, for each $j \in [\ell]$, we place a random value $\alpha_{k,j}$ in the check slot. For example, the input encoding of $s := 0b01$ for the kth input is $\{[0, \alpha_{k,1}]_{S_{k,s}}, [1, \alpha_{k,2}]_{S_{k,s}}\}$; likewise, the encoding of $s' := 0b10$ for the kth input is $\{[1, \alpha_{k,1}]_{S_{k,s'}}, [0, \alpha_{k,2}]_{S_{k,s'}}\}$. Note that $\alpha_{k,j}$ is common across these two encodings. Now, suppose the evaluator wants to input (invalid) symbol 0b11. It can try to use $\{[1, \alpha_{k,1}]_{S_{k,s'}}, [1, \alpha_{k,2}]_{S_{k,s}}\}$; however, these are each encoded under different index sets, and thus the evaluator will be unable to produce a valid encoding at the top-level. Likewise, the check elements $\alpha_{k,j}$ enforce that the evaluator cannot mix-and-match encoded elements within an encoding of a single Σ-vector. For example, $\{[1, \alpha_{k,2}]_{S_{k,s}}, [1, \alpha_{k,2}]_{S_{k,s}}\}$ would fail due to the top-level check slot not canceling out.

B.1 Construction

Let $C : \Sigma^c \times \{0,1\}^m \to \{0,1\}^\gamma$. Let $q = |\Sigma|$ and let ℓ be the bitlength of elements in Σ. Let $S_{k,\neq s} := \prod_{s' \neq s \in \Sigma} S_{k,s'}$, and $S_{k,\Sigma} := \prod_{s \in \Sigma} S_{k,s}$. Let $\deg(x_o^k)$ denote the degree of x^k for output bit $o \in [\gamma]$. We construct our obfuscator as follows:

$\mathsf{LZ.Obfuscate}(1^\lambda, C, \mathbf{y})$:

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 7: 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 MO.

1. Construct top-level index-set

$$\mathcal{U} = Y^{\deg(\mathbf{y})} \prod_{k\in[c]} (S_{k,\Sigma})^{\deg(x^k)} Z_k W_k.$$

2. Compute $(\mathsf{pp}, \mathsf{sp}, p_{\mathrm{ev}}, p_{\mathrm{chk}}) \leftarrow \mathsf{Setup}(\mathcal{U}, 1^\lambda, 2)$.

3. For $k \in [c]$, $j \in [\ell]$, compute $\alpha_{k,j} \leftarrow \mathbb{Z}_{p_{\mathrm{ev}}}$.
   For $k \in [c]$, $s \in \Sigma$, $o \in [\gamma]$, compute $\gamma_{k,s,o} \leftarrow \mathbb{Z}_{p_{\mathrm{ev}}}$, $\delta_{k,s,o} \leftarrow \mathbb{Z}_{p_{\mathrm{chk}}}$.
   For $i \in [m]$, compute $\beta_i \leftarrow \mathbb{Z}_{p_{\mathrm{chk}}}$.

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

$$C^*_o := C((\alpha_{1,j})_{j\in[\ell]}, \ldots, (\alpha_{c,j})_{j\in[\ell]}, \beta_1, \ldots, \beta_m)_o \in \mathbb{Z}_{p_{\mathrm{chk}}}.$$

5. For $k \in [c]$, $s \in \Sigma$, $j \in [\ell]$, generate the following encoded elements:

$$\hat{s}_{k,s,j} := [s_j, \alpha_{k,j}]_{S_{k,s}} \qquad \hat{u}_{k,s} := [1, 1]_{S_{k,s}}.$$

   For $k = 1$, $s \in \Sigma$, $o \in [\gamma]$, generate the following encoded elements:

$$\hat{z}_{1,s,o} := [\delta_{1,s,o}, \gamma_{1,s,o}]_{Y^{\deg(\mathbf{y})-\deg(y_o)}(S_{k,s})^{\deg(x^k)-\deg(x^k_o)}(S_{k,r\neq s})^{\deg(x^k)} Z_k W_k}.$$

   For $k \in \{2, \ldots, c\}$, $s \in \Sigma$, $o \in [\gamma]$, generate the following encoded elements:

$$\hat{z}_{k,s,o} := [\delta_{k,s,o}, \gamma_{k,s,o}]_{(S_{k,s})^{\deg(x^k)-\deg(x^k_o)}(S_{k,r\neq s})^{\deg(x^k)} Z_k W_k}$$

   For $k \in [c]$, $s \in \Sigma$, $o \in [\gamma]$, generate the following encoded elements:

$$\hat{w}_{k,s,o} := [0, \gamma_{k,s,o}]_{W_k}.$$

   For $i \in [m]$, generate the following encoded elements:

$$\hat{y}_i := [y_i, \beta_i]_Y.$$

   For $o \in [\gamma]$, generate the following encoded elements:

$$\hat{C}^*_o := [0, C^*_o]_{Y^{\deg(\mathbf{y})} \prod_{k\in[c]} (S_{k,\Sigma})^{\deg(x^k)} Z_k}.$$

   Finally, generate the following encoded element:

$$\hat{v} := [1, 1]_Y$$

6. Output the following as the obfuscated program:

$$\left(\mathsf{pp}, \{\hat{s}_{k,s,j}\}_{k\in[c],s\in\Sigma,j\in[\ell]}, \{\hat{u}_{k,s}\}_{k\in[c],s\in\Sigma}, \{\hat{z}_{k,s,o}, \hat{w}_{k,s,o}\}_{k\in[c],s\in\Sigma,o\in[\gamma]}, \{\hat{y}_i\}_{i\in[m]}, \hat{v}, \{\hat{C}^*_o\}_{o\in[\gamma]}\right)$$

LZ.Evaluate(Obf, $\mathbf{x}$):

1. For $o \in [\gamma]$, use Add and Mult (as is done in Zimmerman's construction [49, §3]) to compute

$$\hat{C}_o := [C(s_1, \ldots, s_\ell, y_1, \ldots, y_m)_o, C((\alpha_{1,j})_{j\in[\ell]}, \ldots, (\alpha_{c,j})_{j\in[\ell]}, \beta_1, \ldots, \beta_m)_o]_{Y^{\deg(\mathbf{y})} \prod_{k\in[c]} (S_{k,\Sigma})^{\deg(x^k)}}.$$

2. For $o \in [\gamma]$, use Add and Mult to compute

$$\hat{z}_o := \hat{C}_o \prod_{k\in[c]} \hat{z}_{k,\mathsf{sym}(k),o} - \hat{C}^*_o \prod_{k\in[c]} \hat{w}_{k,\mathsf{sym}(k),o},$$

   where $\mathsf{sym}(k)$ returns the symbol $s \in \Sigma$ that corresponds to input k.

3. Output the bitstring resulting from running $\mathsf{ZeroTest}(\hat{z}_o)$ for $o \in [\gamma]$.

B.2 Proof of Security

Theorem B.1. *The construction in Appendix B.1 achieves indistinguishability obfuscation for* NC^1 *in the noisy composite-order multilinear map generic model.*

Proof. For simplicity, we assume that there is only a single output bit (i.e., $o = 1$), and let $\hat{C}^* := \hat{C}^*_1$. We can generalize the proof to multiple output bits using a similar approach as is done by Zimmerman [49, Remark 3.18].

Our proof follows the same framework as Zimmerman's proof. We begin by proving a "structural lemma" on the index sets.

Lemma B.2. *Let $\mathcal{A}$ be an efficient adversary in the composite-order multilinear map generic model, and let z be a formal polynomial produced by $\mathcal{A}$ at the top-level index-set $\mathcal{U}$. Then any monomial t occurring in the formal expansion of z has one of the following two forms:*

1. *For values $x^1, \ldots, x^c \in \Sigma$ and for constant $a \in \mathbb{Z}$:*

$$t = a\hat{C}^* \left(\prod_{k\in[c]} \hat{w}_{k,x^k} \right)$$

2. *For values $x^1, \ldots, x^c \in \Sigma$ and for monomial function h:*

$$t = h\Big((\hat{s}_{1,x^1,j}, \hat{u}_{1,x^1,j})_{j\in[\ell]}, \ldots, (\hat{s}_{c,x^c,j}, \hat{u}_{c,x^c,j})_{j\in[\ell]}, (\hat{y}_i)_{i\in[m]}, \hat{v} \Big) \left(\prod_{k\in[c]} \hat{z}_{k,x^k} \right).$$

Proof. Consider monomial t with index-set $\mathcal{U}$. Note that t must contain encodings with the Z_k indices, and thus must contain either encoding $\hat{C}^*$, or encodings $\hat{z}_{k,s}$.

1. Suppose t contains $\hat{C}^*$. Then the only missing indices are the W_ks, which are contained in the encodings $\hat{w}_{k,s}$. Thus, t must contain one of $(\hat{w}_{k,s})_{s\in\Sigma}$ for each $k \in [c]$. For $k \in [c]$, define $x^k = s$ such that t contains $\hat{w}_{k,s}$. Thus, form (1) is satisfied.

2. Suppose t does not contain $\hat{C}^*$. Then it must contain one of $(\hat{z}_{k,s})_{s\in\Sigma}$ for each $k \in [c]$. For $k \in [c]$, define $x^k = s$ such that t contains $\hat{z}_{k,s}$. As $\hat{z}_{k,x^k}$ contains indices $(S_{k,\neq x^k})^{\deg(x^k)}$, we conclude that t cannot contain encodings $\hat{s}_{k,s,j}$ and $\hat{u}_{k,s,j}$ for $s \neq x^k$ and $j \in [\sigma]$. Thus, form (2) is satisfied. ■

We use the notion of an *input profile* used by Barak et al. [16] and Zimmerman [49], and refer to those works for its formal definition.

Lemma B.3. *Let $\mathcal{A}$ be an efficient adversary in the composite-order multilinear map generic model. For all valid toplevel polynomials z generated by $\mathcal{A}$ in the above construction, (1) the input profile $\mathsf{prof}(z)$ is a set of strings in $\{0,1\}^n$, of which none are partial, and (2) $\mathsf{prof}(z)$ can be computed (inefficiently) given z.*

Proof. Part (1) follows from Lemma B.2. Part (2) follows from the definition of prof: we can directly apply the algorithm for computing $\mathsf{prof}(z)$ given z. ■

We now utilize these lemmas to prove Theorem B.1. Let $\mathcal{A}$ be an efficient adversary. We construct a simulator $\mathcal{S}$ as follows. On input circuit $C : \Sigma^c \times \{0,1\}^m \to \{0,1\}$, $\mathcal{S}$ runs $\mathsf{Obfuscate}(1^\lambda, C, 0^m)$, using the generic multilinear map oracle $\mathcal{M}$ for the Setup and Encode operations, and forwards the output obfuscated program to $\mathcal{A}$. The simulator $\mathcal{S}$ forwards all queries made by $\mathcal{A}$ to Add and Mult to $\mathcal{M}$. On a $\mathsf{ZeroTest}$ query on some handle h, $\mathcal{S}$ proceeds as follows. If, based on the internals of $\mathcal{M}$, h is not a formal polynomial at index set $\mathcal{U}$, then $\mathcal{S}$ returns $\perp$. Otherwise, it follows the algorithm in Figure B.1. When $\mathcal{A}$ terminates, $\mathcal{S}$ forwards the output of $\mathcal{A}$ to the distinguisher.

Correctness follows using the same argument as used by Zimmerman [49]. Thus, we need to show that for any valid zero-test query z, the response made by $\mathcal{S}$ is indistinguishable from that made in the real world.

By Lemma B.2, any zero-test query z must be of the form $z = \sum_{\mathbf{x}\in\mathsf{prof}(z)} f_{\mathbf{x}}$, where:

$$f_{\mathbf{x}} = h_{\mathbf{x}}\Big((\hat{s}_{1,x^1,j}, \hat{u}_{1,x^1,j})_{j\in[\ell]}, \ldots, (\hat{s}_{c,x^c,j}, \hat{u}_{c,x^c,j})_{j\in[\ell]}, (\hat{y}_i)_{i\in[m]}, \hat{v}\Big) \prod_{k\in[c]} \hat{z}_{k,x^k} + a_{\mathbf{x}}\hat{C}^* \prod_{k\in[c]} \hat{w}_{k,x^k}$$

for multivariant polynomial $h_{\mathbf{x}}$ and constant $a_{\mathbf{x}} \in \mathbb{Z}$. As the encodings $\hat{u}_{k,s,j}$ and $\hat{v}$ encode 1, they can be ignored, and thus we can consider the simplified expressions $f'_{\mathbf{x}}$ and $h'_{\mathbf{x}}$, where:

$$f'_{\mathbf{x}} = h'_{\mathbf{x}}\Big((\hat{s}_{1,x^1,j})_{j\in[\ell]}, \ldots, (\hat{s}_{c,x^c,j})_{j\in[\ell]}, (\hat{y}_i)_{i\in[m]}\Big) \prod_{k\in[c]} \hat{z}_{k,x^k} + a_{\mathbf{x}}\hat{C}^* \prod_{k\in[c]} \hat{w}_{k,x^k}.$$

Note that in the real world, the first component takes the form:

$$f'_{\mathbf{x}} = h'_{\mathbf{x}}\Big((\hat{s}_{1,x^1,j})_{j\in[\ell]}, \ldots, (\hat{s}_{c,x^c,j})_{j\in[\ell]}, (\hat{y}_i)_{i\in[m]}\Big) \prod_{k\in[c]} \delta_{k,x^k} \pmod{p_{\mathrm{ev}}}$$

and the second component takes the form:

$$f'_{\mathbf{x}} = (h'_{\mathbf{x}} + a_{\mathbf{x}}C)((\alpha_{1,j})_{j\in[\sigma]}, \ldots, (\alpha_{c,j})_{j\in[\sigma]}, \beta_1, \ldots, \beta_m) \prod_{k\in[c]} \gamma_{k,x^k} \pmod{p_{\mathrm{chk}}}.$$

Now consider the simulator's zero-test decision procedure. We do a case analysis.

On input formal polynomial z, proceed as follows:

1. Compute $\mathsf{prof}(z)$.
2. For each $\mathbf{x} := x^1 \cdots x^k \in \mathsf{prof}(z)$, proceed as follows.
   (a) Compute $z'_{\mathbf{x}} \in \mathbb{Z}[\hat{s}_{k,x^1,1}, \ldots, \hat{s}_{k,x^1,\sigma}, \ldots, \hat{s}_{k,x^c,1}, \ldots, \hat{s}_{k,x^c,\sigma}, \hat{y}_1, \ldots, \hat{y}_m]$ as z with its variables substituted as follows:

$$\hat{z}_{k,x^k}, \hat{w}_{k,x^k} \mapsto 1 \qquad \hat{z}_{k,\neq x^k}, \hat{w}_{k,\neq x^k} \mapsto 1$$
$$\hat{C}^* \mapsto C((\hat{s}_{1,x^1,j})_{j\in[\sigma]}, \cdots, (\hat{s}_{c,x^c,j})_{j\in[\sigma]}, \hat{y}_1, \cdots, \hat{y}_m) \qquad \hat{u}_{k,s,j}, \hat{v} \mapsto 1$$

   (b) Compute $z''_{\mathbf{x}} \in \mathbb{Z}$ as z with its variables substituted as follows:

$$\hat{C}^*, \hat{w}_{k,x^k} \mapsto 1 \qquad \hat{w}_{k,\neq x^k} \mapsto 0 \qquad \hat{s}_{k,s,j}, \hat{y}_i, \hat{z}_{k,s} \mapsto 0 \qquad \hat{u}_{k,s,j}, \hat{v} \mapsto 1$$

   Next, do the following:
   - If $z'_{\mathbf{x}} \not\equiv 0$, return "non-zero".
   - If $z'_{\mathbf{x}} \equiv 0$:
     - If $z''_{\mathbf{x}} = 0$, continue to next $\mathbf{x} \in \mathsf{prof}(z)$.
     - If $z''_{\mathbf{x}} \neq 0$, query the oracle on $C(\mathbf{x}, \mathbf{y})$.
       * If $C(\mathbf{x}, \mathbf{y}) \neq 0$, return "non-zero".
       * If $C(\mathbf{x}, \mathbf{y}) = 0$, continue to next $\mathbf{x} \in \mathsf{prof}(z)$.
3. Return "zero".

Figure B.1: Simulator's decision procedure to handle a valid $\mathsf{ZeroTest}$ query made by $\mathcal{A}$.

1. There exists some $\mathbf{x} \in \mathsf{prof}(z)$ such that $z'_{\mathbf{x}} (= h'_{\mathbf{x}} + a_{\mathbf{x}} C) \not\equiv 0$. This case captures the setting where the check slot results in non-zero. Let $\mathbf{x}^* = x^1, \ldots, x^c$ be the lexicographically first such $\mathbf{x}$. We need to show that with high probability

$$(h'_{\mathbf{x}^*} + a_{\mathbf{x}^*} C)((\alpha_{1,j})_{j\in[\sigma]}, \ldots, (\alpha_{c,j})_{j\in[\sigma]}, \beta_1, \ldots, \beta_m) \prod_{k\in[c]} \gamma_{k,x^k} \not\equiv 0 \pmod{p_{\mathrm{chk}}}.$$

   We first consider the expression $h'_{\mathbf{x}^*} + a_{\mathbf{x}^*} C$. Note that the total degree is bounded by 2^d Thus, we can apply the Schwartz-Zippel lemma to conclude that the probability $h'_{\mathbf{x}^*} + a_{\mathbf{x}^*} C$ evaluates to zero is negligible.

   Now consider the entire query z. From the above we know that the coefficient of $\prod_{k\in[c]} \gamma_{k,x^k}$ is non-zero with high probability, and thus we conclude that z is non-zero with high probability. Thus, the simulator's response of "non-zero" is correct (with all but negligible probability).

2. For every $\mathbf{x} \in \mathsf{prof}(z)$ we have $z'_{\mathbf{x}} \equiv 0$, but there exists at least one $\mathbf{x} \in \mathsf{prof}(z)$ such that $z''_{\mathbf{x}} \neq 0$ and $C(\mathbf{x}, \mathbf{y}) \neq 0$. This case captures the setting where the correct evaluation of C is non-zero.

Let $\mathbf{x}^*$ be the lexicographically first such $\mathbf{x}$. We need to show that

$$f'_{\mathbf{x}} = h'_{\mathbf{x}}(x^1, \ldots, x^c) \prod_{k \in [c]} \delta_{k,x^k} \not\equiv 0 \pmod{p_{\text{ev}}}.$$

We have that $h'_{\mathbf{x}^*} + a_{\mathbf{x}^*} C \equiv 0$ and thus $h'_{\mathbf{x}^*}(\mathbf{x}^*, \mathbf{y}) = -a_{\mathbf{x}^*} C(\mathbf{x}^*, \mathbf{y}) \neq 0$.

Now consider the entire query z. From the above we know that the coefficient of $\prod_{k \in [c]} \delta_{k,x^k}$ is non-zero with high probability, and thus we conclude that z is non-zero with high probability. Thus, the simulator's response of "non-zero" is correct.

3. For every $\mathbf{x} \in \mathsf{prof}(z)$ we have $z'_{\mathbf{x}} \equiv 0$ and either $z''_{\mathbf{x}} = 0$ or $C(\mathbf{x}, \mathbf{y}) = 0$. This case captures the setting where the correct evaluation of C is zero. We need to show that

$$f'_{\mathbf{x}} = h'_{\mathbf{x}}(x^1, \ldots, x^c) \prod_{k \in [c]} \delta_{k,x^k} \equiv 0 \pmod{p_{\text{ev}}}.$$

Since $h'_{\mathbf{x}} \equiv -a_{\mathbf{x}} C$ it holds that $h'_{\mathbf{x}}(\mathbf{x}, \mathbf{y}) = -a_{\mathbf{x}} C(\mathbf{x}, \mathbf{y})$, and since either $a_{\mathbf{x}} = 0$ or $C(\mathbf{x}, \mathbf{y}) = 0$ we conclude that $h'_{\mathbf{x}}(\mathbf{x}, \mathbf{y}) = 0$. Thus, the simulator's response of "zero" is correct. ■

Computing κ. Instead of providing a closed form formula for determining κ for a given circuit, we instead compute it directly using a dummy multilinear map. This gives an exact value for κ as opposed to a potential over-approximation, as we found for the formulas for Zim, AB, and Lin.

Changelog

- Version 1.0 (August 29, 2017): Full version of a paper published at ACM CCS 2017.