Implementing Cryptographic Program Obfuscation

Daniel Apon[*] Yan Huang[†] Jonathan Katz[*] Alex J. Malozemoff[*]

Abstract

Program obfuscation is the process of making a program "unintelligible" without changing the program's underlying input/output behavior. Although there is a long line of work on heuristic techniques for obfuscation, such approaches do not provide any cryptographic guarantee on their effectiveness. A recent result by Garg et al. (FOCS 2013), however, shows that cryptographic program obfuscation is indeed possible based on a new primitive called a *graded encoding scheme*.

In this work, we present the first implementation of such an obfuscator. We describe several challenges and optimizations we made along the way, present a detailed evaluation of our implementation, and discuss research problems that need to be addressed before such obfuscators can be used in practice.

1 Introduction

The goal of *program obfuscation* is to make programs "unintelligible" while preserving their original behavior. Formally, an obfuscator $\mathcal{O}$ is a compiler taking as input a program f (expressed, say, in C code or as a Boolean circuit) and outputting an obfuscated program $f' = \mathcal{O}(f)$ such that $f'(x) = f(x)$ for all inputs x. Intuitively, the security requirement is that the obfuscated code f' reveals nothing about the internal structure of the original program f other than what can be inferred from its input/output behavior and (a bound on) its size/running time.

For decades, program obfuscation has been suggested for, e.g., protecting intellectual property [24] or avoiding static or dynamic code analysis [23, 29, 31]. Generally, researchers have approached the problem by applying a series of static program transformations, hoping to obtain an incomprehensible (but equivalent) version of the original program. Unfortunately, such approaches do not offer any cryptographic guarantee of security. However, the recent landmark result of Garg et al. [17] demonstrates that cryptographic program obfuscation can be achieved, assuming a *graded encoding scheme*. Since then, many papers have been written improving various aspects of the original construction [3, 5, 11, 25] and utilizing the technique for many interesting applications [10, 16, 20, 26].

The construction of Garg et al. [17] satisfies a weaker notion of obfuscation than the *virtual black-box obfuscation* notion described intuitively above (which is known to be impossible to achieve in general [6]). This weaker notion, called *indistinguishability obfuscation*, ensures that for any two equivalent programs f and g (i.e., programs such that $f(x) = g(x)$ for all x), an obfuscation of f is indistinguishable (in a cryptographic sense) from an obfuscation of g. While weaker than virtual black-box obfuscation, indistinguishability obfuscation is surprisingly powerful. Applications

[*]University of Maryland. {dapon,jkatz,amaloz}@cs.umd.edu
[†]Indiana University. Work done while at University of Maryland. yh33@indiana.edu

of indistinguishability obfuscation include deniable encryption [26], efficient traitor tracing [10], two-round multiparty computation with succinct messages [16], full domain hash without a random oracle [20], and more. In addition, many obfuscation constructions have been shown to be virtual black-box obfuscators in a generic model [3, 5, 11], and it appears that such constructions may indeed be virtual black-box obfuscators for particular classes of functions, such as *evasive* functions [4].

Yet despite the fast growing body of literature on cryptographic program obfuscation, the idea has thus far remained entirely theoretical. It was not even clear whether these new constructions are even close to practically feasible.

Contributions. This work investigates the practicality of cryptographic program obfuscation through implementation and experimentation.

- We implement a full program-obfuscation toolchain for the first time, based on schemes from the literature [5, 11] and using the formula-to-branching-program construction of Ananth et al. [3]; see Section 2 and Section 3. Our code is publicly available for the benefit of the research community.[1]
- We investigate the *practicality* of program obfuscation through experiments; see Section 4. We find that program obfuscation is still far from being deployable, with the most complex functionality we are able to obfuscate being a 16-bit point function (i.e., a function that outputs 1 if the input matches some secret value, and 0 otherwise).
- We discuss the implications of our work and the bottlenecks that researchers should focus on to make obfuscation more practical; see Section 5.

2 Overview

In this section, we provide a self-contained overview of the techniques underlying cryptographic program obfuscation and our implementation. Our obfuscator $\mathcal{O}$ takes as input a program represented as a *Boolean formula*. (We only implement "basic" obfuscation and not the "bootstrapping" step which requires obfuscating an FHE logic implemented purely by Boolean *formulas* to obfuscate generic Boolean *circuits*. Note that any circuit can be expressed as a formula; although this results in exponential blowup asymptotically, for moderately-sized circuits converting the circuit to a formula and using basic obfuscation should be more efficient than applying bootstrapping to the original circuit.) A Boolean formula is a binary tree where each vertex is labeled with some Boolean gate type — in our implementation, AND, NOT, or XOR — and each leaf is labeled with some input bit x_i for $i \in \{1, \dots, n\}$; the root of the tree corresponds to the (1-bit) output.[2] The *depth* d of a formula is the number of vertices on the longest path from any leaf to the root, and the *size* s of a formula is the number of vertices in the entire tree.

Given a Boolean formula f, our obfuscator proceeds through a sequence of transformations (see Figure 1) before producing the final obfuscated program $\mathcal{O}(f)$. We give a general outline of these steps before giving further details:

[1] https://github.com/amaloz/obfuscation

[2] In order to obfuscate a function with k output bits one would need to construct k Boolean formulas, with the ith formula computing the ith output bit.

$$\begin{tikzcd} f \arrow[r, "\text{[7] or [28]}", "\text{Section 2.1}"'] & BP_f \arrow[r, "\text{convert}", "\text{Section 2.2}"'] & MBP_f \arrow[r, "\text{randomize}", "\text{Section 2.3}"'] & \widetilde{MBP}_f \arrow[r, "\text{encode}", "\text{Section 2.4}"'] & \mathcal{O}(f) \arrow[r, "\text{evaluate on } x", "\text{Section 2.5}"'] & f(x) \end{tikzcd}$$

Figure 1: Workflow diagram for obfuscation. Starting from a Boolean formula f, we first convert f to a *branching program* BP_f using one of two methods. We then convert this to a *matrix branching program* MBP_f. Next, MBP_f is randomized to give $\widetilde{MBP}_f$. Finally, each matrix element in $\widetilde{MBP}_f$ is encoded using a graded encoding scheme. The output of this procedure is an obfuscation $\mathcal{O}(f)$ of f. Any party given $\mathcal{O}(f)$ can evaluate it on any input x of their choice to compute $\mathcal{O}(f)(x) = f(x)$.

1. The Boolean formula f is converted into a functionally equivalent *branching program* BP_f, written as a directed, acyclic graph (Section 2.1.1). (Branching programs are defined below.)

2. The branching program BP_f is mapped to a *matrix branching program* MBP_f, which consists of a sequence of pairs of matrices (Section 2.2).

3. The matrix branching program MBP_f is *randomized* to give $\widetilde{MBP}_f$ (Section 2.3). Intuitively, this step (in combination with the next step) prevents an attacker from mixing-and-matching different pieces of the computation together.

4. The randomized matrix branching program $\widetilde{MBP}_f$ is then encoded, matrix element by matrix element, using a graded encoding scheme (Section 2.4). This step can be viewed, naively, as encrypting MBP_f using a scheme that enables homomorphic evaluation of MBP_f, and is the only step that relies on a cryptographic hardness assumption. It has the effect of hiding from the attacker all details of the matrix branching program as well any information about the computation being performed (other than the final output).

We now describe each of these steps in more detail. In order to illustrate the concepts, we use the running example of obfuscating a single AND gate, i.e., the Boolean formula $f(x_1, x_2) = x_1 \wedge x_2$.

Notation. We let λ denote the security parameter. That is, an obfuscation with security parameter λ is designed to bound the probability of successful attacks by $2^{-\lambda}$. We let $[k] = \{1, \dots, k\}$, use bold letters to denote vectors, and use capital letters to denote matrices.

2.1 Branching Programs

Let $f : \{0,1\}^n \to \{0,1\}$ be some Boolean formula, and let $x \in \{0,1\}^n$ be the input with x_i denoting the ith bit of x. A *branching program* is a directed acyclic graph whose vertices are partitioned into disjoint layers, defined as follows:

Definition 1 *A* ***branching program*** *of width w and length m is a directed acyclic graph with s vertices, such that the following constraints hold:*

- *Each vertex is either a* source *vertex, an* internal *vertex or a* final *vertex.*
- *Each non-final vertex is labeled with some $\ell \in \{`x_1', \dots, `x_n'\}$ and has out-degree two, with one of the outgoing edges labeled '0' and the other '1'.*

- *There are two final vertices; both have out-degree zero and one is labeled '0' and the other '1'.*
- *There is a unique source vertex with in-degree zero.*
- *The vertices are partitioned into m layers L_j, where $|L_j| \leq w$. All vertices in a layer have the same label 'x_i'.*
- *Edges starting in layer L_j end in some layer $L_{j'}$ with $j' > j$.*
- *Both final vertices are in the same layer, L_m.*

Note that the notation 'x_i' stresses it is the *symbol*, rather than the bit value represented by the symbol, which is associated with a vertex.

We can evaluate a branching program as follows. On input $x = x_1 \cdots x_n$, start from the source node in the branching program and traverse the edges corresponding to the Boolean value of the current node's label. For example, suppose the source node has label 'x_j'. Then, on an input x where $x_j = 1$, we would take the outgoing edge labeled '1'. The 0/1 label of the final vertex is the output of the program.

2.1.1 From Formulas to Branching Programs

At the time of writing, there were two approaches for compiling Boolean formulas into branching programs: using Barrington's theorem [7] (as done in the work of Garg et al. [17], among others), or using the approach of Sauerhoff et al. [28] (as proposed in the work of Ananth et al. [3]).[3] Barrington's theorem shows how to convert any Boolean formula of depth d into a branching program of width 5 and length at most 4^d, whereas the approach of Sauerhoff et al. converts formulas of size s into branching programs of width at most $2s + 4$ and length at most s. However, the approach of Sauerhoff et al. is more efficient both asymptotically [3] and in practice, and thus our implementation focuses on this approach.

The transformation of Sauerhoff et al. from formulas to branching programs is inductive. The base case is a "trivial" branching program—one for each input wire—consisting of only three vertices: the source node, the reject node, and the accept node. There is a directed edge labeled '1' from source to accept, and a directed edge labeled '0' from source to reject.

We then proceed inductively. Given a branching program BP_f for some formula f, we can construct the branching program for the formula $\neg f$ by swapping the labels of the accept and reject nodes in BP_f.

Given two branching programs BP_f and BP_g for the formulas f and g, we can "AND" the two formulas, producing the branching program $BP_{f \wedge g}$, as follows: First, we merge the accept node of BP_f with the source node of BP_g. Second, we merge the reject node of BP_f with the reject node of BP_g. Finally, we let the source node of BP_f be the source node of $BP_{f \wedge g}$, and let the accept and reject nodes of BP_g be the accept and reject nodes, respectively, of $BP_{f \wedge g}$. (Figure 2 shows the branching program obtained by applying this approach to a single AND gate.)

Lastly, given two branching programs BP_f and BP_g for the formulas f and g, we can "XOR" the two formulas, producing the branching program $BP_{f \oplus g}$ as follows: Without loss of generality, assume BP_g is the smaller of the two branching programs (otherwise we can swap the order of f and g). We first duplicate BP_g and use the NOT-transformation to produce $BP_{\neg g}$. Next, we merge the accept node of BP_f with the source node of $BP_{\neg g}$ and merge the reject node of BP_f with the

[3]Recently, two new approaches [27, 32] to constructing obfuscators present asymptotic improvements to the approaches discussed in this work. In addition, later versions of the work of Ananth et al. [3] propose an alternative approach than that of Sauerhoff et al. We leave implementations of these new approaches as future work.

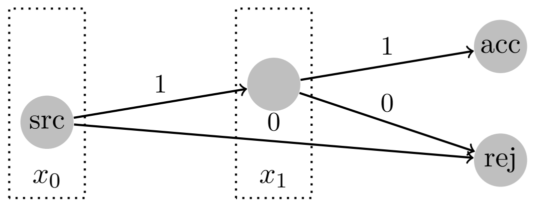

Figure 2: Branching program for an AND gate. To evaluate on input x_0x_1, look at each layer of the graph (denoted by the dotted rectangles). If the layer corresponds to the ith input bit, then remove all outgoing edges from that layer that are not labeled by the value x_i. The output is 1 iff there is a path from src to acc in the resulting graph.

source node of BP_g. Finally, we merge the accept nodes (and, respectively, the reject nodes) of BP_g and $BP_{\neg g}$. The branching program $BP_{f \oplus g}$ has the same source node as BP_f and the same accept and reject nodes as BP_g (equivalently, $BP_{\neg g}$).

We now prove that given a branching program constructed as above, then for all inputs x there exists at most one path from the source to the accept node.

Lemma 1 *Let BP_f be a branching program constructed using the above procedure. Then for all x, there exists at most one path from the source to the accept node in BP_f.*

Proof. We prove the result by induction. Clearly, this holds for the base case (namely, the "trivial" branching program described above). Likewise, it is easy to see that given BP_f, the construction $BP_{\neg f}$ also maintains the invariant.

Now let BP_f and BP_g be two branching programs. We first show that the invariant is maintained when we construct $BP_{f \wedge g}$. The resulting branching program merges the accept node of BP_f with the source node of BP_g; this step maintains the invariant as by induction there exists a single path to the accept node in BP_f. Likewise, the resulting branching program merges the reject node of BP_f with the reject node of BP_g; as this does not affect the accept node the invariant is maintained.

The proof that the invariant is maintained when we construct $BP_{f \oplus g}$ is similar to the above case, and thus omitted. ∎

2.2 Matrix Branching Programs

The next step is to compile the graph-based branching program BP_f into a functionally equivalent matrix branching program MBP_f, which can be evaluated by iterated matrix multiplication. We utilize the notion of a matrix branching program, defined as follows:

Definition 2 *A **matrix branching program** of width w and length m for n-bit inputs is given by a tuple*

$$MBP_f = \left(\mathsf{inp}, \mathbf{s}, (B_{j,0}, B_{j,1})_{j \in [m]}, \mathbf{t}\right)$$

where $B_{j,b} \in \{0,1\}^{w \times w}$, for $j \in [m]$ and $b \in \{0,1\}$, $\mathsf{inp} : [m] \to [n]$ is a function mapping layer indices j to input-bit indices i, $\mathbf{s} := (1, 0, \ldots, 0)$, and $\mathbf{t} := (0, \ldots, 0, 1)^T$. The output of MBP_f on input $x \in \{0,1\}^n$ is defined as:

$$MBP_f(x) = \begin{cases} 1 & \text{if } \mathbf{s} \cdot \left(\prod_{j=1}^{m} B_{j, x_{\mathsf{inp}(j)}}\right) \cdot \mathbf{t} \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\left(\underbrace{\begin{bmatrix}1\\0\\0\\0\end{bmatrix}}_{s}, \underbrace{\left(\begin{bmatrix}1&0&1&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}, \begin{bmatrix}1&1&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \right)}_{x_0}, \underbrace{\left(\begin{bmatrix}1&0&0&0\\0&1&1&0\\0&0&1&0\\0&0&0&1\end{bmatrix}, \begin{bmatrix}1&0&0&0\\0&1&0&1\\0&0&1&0\\0&0&0&1\end{bmatrix} \right)}_{x_1}, \underbrace{\begin{bmatrix}0\\0\\0\\1\end{bmatrix}}_{t} \right)$$

Figure 3: Matrix branching program for an AND gate. To evaluate on input x_0x_1, proceed as follows. If $x_i = 0$, choose the first matrix; otherwise choose the second matrix. Multiply the matrices; if the $[1, 4]$ entry of the resulting matrix is 1 then the output is 1; otherwise, the output is 0.

Note that the output of a matrix branching program is the entry in the first row and last column of the iterated matrix product, and the vectors $\mathbf{s}$ and $\mathbf{t}$ simply select this entry as the output.

To convert a branching program to a matrix branching program, we proceed as follows. For each layer L_j of BP_f, we produce two w-by-w 0/1-valued matrices $B_{j,0}, B_{j,1}$. The matrix $B_{j,b}$ is the adjacency matrix composed of edges leaving layer L_j with label b. Additionally, we set the diagonal entries to 1 (i.e., we include self-loops in the adjacency matrix), which is needed since as an optimization we allow layers in our branching program construction to have differing numbers of nodes. (In Figure 3 we depict the matrix branching program generated from the branching program from Figure 2.)

We now prove that when using the above procedure to convert branching programs to their matrix equivalents, it in fact holds that $\mathbf{s} \cdot \left(\prod_{j=1}^{m} B_{j,x_{\mathsf{inp}(j)}}\right) \cdot \mathbf{t} \in \{0, 1\}$, which saves us from running the randBP' step as detailed by Ananth et al. [3].

Lemma 2 *Let MBP_f be a matrix branching program converted from a branching program BP_f using the above procedure. Then for all inputs x, it holds that $\mathbf{s} \cdot \left(\prod_{j=1}^{m} B_{j,x_{\mathsf{inp}(j)}}\right) \cdot \mathbf{t} \in \{0, 1\}$.*

Proof. This follows directly from combining Lemma 1 with the proof of Lemma 2 from the work of Ananth et al. [3]. Namely, in the latter proof, we can utilize the fact that there exists at most one path between any two vertices in the graph of BP_f to conclude that the resulting matrix computation contains only 0/1 values. ■

2.2.1 Oblivious Branching Programs

Depending on the setting, it may also be necessary to ensure that the inp function does not leak information about the underlying formula f. (This is needed in order to satisfy the formal definition of obfuscation. But in settings where some information about f is known anyway—e.g., when f is known to be a point function, and only the hidden point must remain secret—this step may not be needed.) This can be done by fixing inp independently of f, setting inp to cycle through each of the input bits $x_1, x_2, \ldots, x_n$ in order, m times. "Dummy" layers are filled in with a pair of identity matrices, thus preserving correctness of the iterated matrix multiplication.

Note, however, that when using the approach of Sauerhoff et al., the width w is a function of the *type* of gates used. For example, a standalone XOR gate under Sauerhoff et al.'s approach equates to a width 5 matrix branching program whereas a standalone AND gate equates to a width 4 matrix branching program [28]. Thus, we need to pad the graph with "dummy" vertices *before* transforming the graph into a matrix branching program to prevent the attacker from learning information about what gate is being computed by simply looking at the width of the matrix branching program.

2.3 Randomized Matrix Branching Programs

The first step toward hiding the underlying computation is to *randomize* the matrices [21]. This randomization ensures that an attacker cannot "mix-and-match" steps of the computation once the matrices have been encoded under the graded encoding scheme.

Let g be a prime, and define the procedure $\mathsf{rand}_g(MBP_f)$ as follows:

1. Uniformly sample $m+1$ invertible matrices from $\mathbb{Z}_g^{w \times w}$, denoted $R_0, \ldots, R_m$.
2. Let $\widetilde{B}_{j,b} := R_{j-1} B_{j,b} R_j^{-1}$ for all $j \in [m], b \in \{0,1\}$.
3. Compute two "bookend" vectors $\widetilde{\mathbf{s}} := (1, 0, \ldots, 0) \cdot R_0^{-1}$ and $\widetilde{\mathbf{t}} := R_m \cdot (0, \ldots, 0, 1)^T$.
4. Output $\widetilde{MBP}_{f,g} = \left(\mathsf{inp}, \widetilde{\mathbf{s}}, \left(\widetilde{B}_{j,0}, \widetilde{B}_{j,1}\right)_{j \in [m]}, \widetilde{\mathbf{t}}\right)$.

Indeed, we have that for all inputs $x \in \{0,1\}^n$,

$$\widetilde{MBP}_{f,g}(x) = \widetilde{\mathbf{s}} \cdot \left(\prod_{j=1}^{m} \widetilde{B}_{j,x_{\mathsf{inp}(j)}}\right) \cdot \widetilde{\mathbf{t}} = \mathbf{s} \cdot R_0^{-1} \cdot \left(\prod_{j=1}^{m} R_{j-1} B_{j,b} R_j^{-1}\right) \cdot R_m \cdot \mathbf{t}$$
$$= \mathbf{s} \cdot \left(\prod_{j=1}^{m} B_{j,x_{\mathsf{inp}(j)}}\right) \cdot \mathbf{t} = MBP_f(x),$$

where matrix multiplication arithmetic is performed modulo g. For notational convenience, we drop the g subscript and write $\widetilde{MBP}_f$ when the modulus is understood.

2.4 Graded Encoding Schemes

While the prior randomization step ensures that a valid sequence of matrices must be multiplied in order, we still need to ensure that (1) only valid sequences of matrices give meaningful output, and (2) the matrix values themselves are hidden—or more generally, that inspecting the obfuscated program does not leak distinguishing information about the underlying implementation. Concretely, observe that randomized matrix branching programs can suffer from attacks that try to compute something other than the original function $f(x)$. For example, if two layers L_j and $L_{j'}$ of $\widetilde{MBP}_f$ are labeled with the same input bit x_i, it is possible to include $\widetilde{B}_{j,0}$ and $\widetilde{B}_{j',1}$ in the product and obtain some output where the random R_j terms still cancel out. This equates to computing a function f' that is different from f, which enables the attacker to do things that are impossible merely from access to f.

To thwart such attacks, we need to use some cryptographic mechanism to hide the entries of the matrices and to prevent an attacker from using inconsistent matrices for some input bit. The cryptographic primitive used is a *graded encoding scheme* [15].

In a graded encoding scheme, producing an encoding of a plaintext message is a private method that can only be performed by a party that knows the scheme's secret parameters. But, similar to fully homomorphic encryption (FHE), any party, given an encoding u of message m and an encoding u' of a message m', can add $u + u'$ to get a new encoding of the message $m + m'$. Similarly, multiplying encodings $u \cdot u'$ gives a new encoding of the message $m \cdot m'$.

Unlike FHE, however, each encoding u is defined relative to some subset S of a universe $[Z]$. These subsets *restrict* the way in which encodings may be added and multiplied. In particular, two encodings may only be added if they are encoded relative to the *same* set S, and their sum is encoded relative to S as well. Moreover, two encodings may only be multiplied if they are encoded relative to two *disjoint* sets S and S', and their product is encoded relative to the *union* of S and S'.

Finally, and again in contrast to FHE, there is a public "zero-test parameter" p_{zt} that is used to test if an encoding u encodes the plaintext 0. However, zero-testing only works on encodings relative to the *entire universe* $[Z]$.

Definition 3 *A* ***graded encoding scheme*** *is a tuple of probabilistic polynomial time algorithms* (InstGen*,* Enc*,* Add*,* Mult*,* isZero) *that behave as follows:*

- ***Instance Generation:*** $(\mathsf{sp}, \mathsf{pp}) \leftarrow \mathsf{InstGen}(1^\lambda, 1^\kappa)$*.*
  InstGen *takes as input the security parameter λ and multilinearity parameter κ, and outputs secret parameters* sp *and public parameters* pp*. The secret parameters* sp *contain an integer Z with $\kappa \leq Z \leq 2\kappa$, primes $g_1, \ldots, g_N$, where $N = poly(\lambda, \kappa)$, and a collection of sets $\{E_S^m : m \in \mathbb{Z}_{g_1} \times \cdots \times \mathbb{Z}_{g_N}, S \subseteq [Z]\}$. We view E_S^m as the set of possible encodings of the value m with respect to the set S. The public parameters* pp *enable the public operations on encodings described below.*

- ***Encoding:*** $u \leftarrow \mathsf{Enc}(\mathsf{sp}, m, S)$*.*
  Enc *takes as input the secret parameters* sp*, a plaintext value $m \in \mathbb{Z}_{g_1} \times \cdots \times \mathbb{Z}_{g_N}$, and a set $S \subseteq [Z]$, and outputs a randomized encoding of m with respect to the set S, denoted $u \in E_S^m$.*

- ***Addition:*** $u \leftarrow \mathsf{Add}(\mathsf{pp}, u, u')$*.*
  Add *takes as input the public parameters* pp *and encodings $u \in E_S^m, u' \in E_{S'}^{m'}$, and outputs an encoding $u \in E_S^{m+m'}$ if $S = S'$, and $\perp$ otherwise.*

- ***Multiplication:*** $u \leftarrow \mathsf{Mult}(\mathsf{pp}, u, u')$*.*
  Mult *takes as input the public parameters* pp *and encodings $u \in E_S^m, u' \in E_{S'}^{m'}$, and outputs an encoding $u \in E_{S \cup S'}^{m \cdot m'}$ if $S \cap S' = \emptyset$, and $\perp$ otherwise.*

- ***Zero Test:*** $b \leftarrow \mathsf{isZero}(\mathsf{pp}, u)$*.*
  isZero *takes as input the public parameters* pp *and an encoding u and outputs 1 if $u \in E_{[Z]}^{\mathbf{0}}$ (i.e., u is an encoding of **0** with respect to the entire universe $[Z]$), and 0 otherwise.*

We defer a discussion of security until later, but informally, a graded encoding scheme is secure if (1) the only way an evaluator can produce an encoding u^* of the all-zeros plaintext with respect to the entire universe $[Z]$ is by combining *given* encodings $\{u_1, u_2, \ldots\}$ by addition and multiplication, and (2) only such encodings u^* pass the zero-test.

2.4.1 Choosing a Set System

It remains to describe the system of sets S under which each matrix element of $\widetilde{MBP_f}$ is encoded in order to produce $\mathcal{O}(f)$. Intuitively, we want to leverage the fact that zero-testing requires an element encoded with respect to the *entire* universe $[Z]$ to force the evaluator of $\mathcal{O}(f)$ to commit to a fixed setting of each input bit $x_i \in \{0,1\}$. In particular, we ensure that only combinations of matrices in $\widetilde{MBP}_f$ that are consistent with some well-defined input x produce an encoding that forms an exact set cover of $[Z]$. Thus, we associate each matrix of $\widetilde{MBP}_f$ with a distinct subset $S \subset [Z]$. This means that for any fixed matrix of $\widetilde{MBP}_f$, we encode each of its individual elements using the same set S. The vectors $\mathbf{s}$ and $\mathbf{t}$ are each encoded using a distinct element in $[Z]$.

More concretely, we group the matrices of $\widetilde{MBP}_f$ according to their input bit label 'x_i'. For example, denote the matrices associated with x_1 as $\widetilde{B}_{j_1,0}, \widetilde{B}_{j_1,1}, \ldots, \widetilde{B}_{j_k,0}, \widetilde{B}_{j_k,1}$ (for some k). We assign sets S to these matrices as follows:

- We assign the matrices $\widetilde{B}_{j_1,0}, \widetilde{B}_{j_2,0}, \ldots, \widetilde{B}_{j_k,0}$ the sets $\{1,2\}, \{3,4\}, \ldots, \{2k-3, 2k-2\}, \{2k-1\}$ (in that order).
- We assign the matrices $\widetilde{B}_{j_1,1}, \widetilde{B}_{j_2,1}, \ldots, \widetilde{B}_{j_k,1}$ the sets $\{1\}$, $\{2,3\}$, $\ldots$, $\{2k-2, 2k-1\}$ (in that order).

We continue this process for each such group of matrices (associated with the remaining labels 'x_2', $\ldots$, 'x_n') using increased set-indices, until every matrix is assigned a distinct subset $S \subset [Z]$. This process allows us to determine the value of Z needed in our implementation, which we increase by two to account for encoding the vectors $\mathbf{s}$ and $\mathbf{t}$.

Note the attacker cannot use inconsistent values for a given variable since sets corresponding to the inconsistent values are not disjoint (as valid multiplications require disjoint sets). The attacker cannot run a zero-test on a partially evaluated program either, because the encoding of a partial result is always formed with respect to a strict subset of Z whereas zero-testing only works for encodings regarding the (full) set Z.

2.5 Executing Obfuscated Programs

Putting everything together, an evaluator can use the Add and Mult procedures of the graded encoding scheme to evaluate the iterated matrix multiplication of an underlying, randomized matrix branching program $\widetilde{MBP}_f$ on some input x, retrieving $\mathsf{Enc}(\mathbf{s} \cdot \prod_{j=1}^{m} B_{j,x_{\mathsf{inp}(j)}} \cdot \mathbf{t}) = u$. The evaluator can then use isZero to zero-test this result; namely, if $\mathsf{isZero}(\mathsf{pp}, u) = 1$ then $f(x) = 0$, otherwise $f(x) = 1$.

3 Implementation

We have implemented an obfuscator using both Barrington's theorem [5] and the approach of Sauerhoff et al. [3] for converting formulas to branching programs. Our implementation is written in a combination of Python and C, using Sage [30] for algebraic operations, the GNU Multiple Precision Arithmetic Library [1] for efficient large number computations, and OpenMP [2] for parallelization. We program all the computationally expensive steps in C, and thus our use of Python does not significantly affect the overall performance.[4]

As shown in Figure 1, our tool takes as input a Boolean formula f and produces as output its obfuscated form $\mathcal{O}(f)$, which can then be evaluated by anyone who receives a copy of it. In

[4]For example, when obfuscating and evaluating $f(x_0, x_1) = x_0 \wedge x_1$ with security parameter 128 we find that 98.5% of the execution time is spent executing C code. This percentage only increases as the size of the Boolean formula increases.

the rest of this section, we provide details about our implementation and discuss some challenges encountered during implementation along with the various optimizations we made to address them.

3.1 Implementing Graded Encodings

We utilize the graded encoding scheme of Coron et al. [14] (the "CLT scheme"), which has better space efficiency than the scheme based on ideal lattices [15]. The CLT graded encoding scheme is instantiated as follows. (We discuss the concrete values of all parameters in Section 3.1.2.)

- InstGen: Let Z be the desired size of the set system (cf. Section 2.4.1), and N a value that depends on Z and the security parameter λ (see Section 3.1.2). First, we generate N (large) secret primes $\{p_i\}_{i=1}^N$ and compute their product $q = \prod_{i=1}^N p_i$. In addition, we generate N (small) secret primes $\{g_i\}_{i=1}^N$, N random integers $\{h_i\}_{i=1}^N$, and Z random values $z_1, \ldots, z_Z \in \mathbb{Z}_q$.

  Next, we construct a zero-test parameter, defined as the integer[5]

$$p_{zt} = \sum_{i=1}^N h_i \cdot \left(\prod_{j=1}^Z z_j \cdot g_i^{-1} \bmod p_i \right) \cdot \prod_{i' \neq i} p_{i'} \bmod q.$$

  The secret parameters are $\left(\{z_i\}_{i=1}^Z, \{g_i\}_{i=1}^N, \{p_i\}_{i=1}^N\right)$; the public parameters are (p_{zt}, q).

- Enc: An encoding of a message $m = (m_1, \ldots, m_N) \in \mathbb{Z}_{g_1} \times \cdots \times \mathbb{Z}_{g_N}$ relative to the set $S \subseteq [Z]$ is a value $u \in \mathbb{Z}_q$ such that for all $1 \leq i \leq N$,

$$u \equiv \frac{r_i \cdot g_i + m_i}{\prod_{j \in S} z_j} \pmod{p_i} \tag{1}$$

  for (small, random) integers r_i.

- Add: Given $u, u' \in \mathbb{Z}_q$ with

$$\forall i: \; u \equiv \frac{r_i \cdot g_i + m_i}{\prod_{j \in S} z_j} \pmod{p_i}, \quad u' \equiv \frac{r'_i \cdot g_i + m'_i}{\prod_{j \in S} z_j} \pmod{p_i},$$

  it is easy to see that

$$\forall i: \; u + u' \equiv \frac{(r_i + r'_i) \cdot g_i + (m_i + m'_i)}{\prod_{j \in S} z_j} \pmod{p_i},$$

  which lies in $E_S^{m+m'}$ assuming $(r_i + r'_i) \cdot g_i + (m_i + m'_i) < p_i$ for all i.

- Mult: Let S and S' be sets such that $S \cap S' = \emptyset$. Given $u, u' \in \mathbb{Z}_q$ with

$$\forall i: \; u \equiv \frac{r_i \cdot g_i + m_i}{\prod_{j \in S} z_j} \pmod{p_i}, \quad u' \equiv \frac{r'_i \cdot g_i + m'_i}{\prod_{j \in S'} z_j} \pmod{p_i}$$

[5] Note that here we are utilizing the heuristic as presented by Coron et al. [14, §6] of using a single zero-testing element p_{zt} rather than a vector of elements.

it is easy to see that

$$\forall i:\ u \cdot u' \equiv \frac{(r_i r_i' + r_i m_i' + r_i' m_i) \cdot g_i + m_i m_i'}{\prod_{j \in S \cup S'} z_j} \pmod{p_i},$$

which lies in $E_{S \cup S'}^{m \cdot m'}$ assuming $(r_i r_i' + r_i m_i' + r_i' m_i) \cdot g_i + m_i m_i' < p_i$ for all i.

- isZero: Using p_{zt}, we zero-test an element u encoded with respect to $[Z]$ by first computing

$$\omega := p_{zt} \cdot u \pmod{q}$$

where

$$\begin{aligned} p_{zt} \cdot u = &\left(\sum_{i=1}^{N} h_i \cdot \left(\prod_{j=1}^{Z} z_j \cdot g_i^{-1} \bmod p_i \right) \cdot \prod_{i' \neq i} p_{i'} \right) \\ &\cdot \left(\sum_{i=1}^{N} (r_i \cdot g_i + m_i) \cdot \left(\prod_{j=1}^{Z} z_j^{-1} \bmod p_i \right) \cdot \left(\prod_{i' \neq i} p_{i'} (p_{i'}^{-1} \bmod p_i) \right) \right) \pmod{q} \\ = &\sum_{i=1}^{N} h_i \cdot \left(\prod_{j=1}^{Z} z_j \cdot g_i^{-1} \bmod p_i \right) \cdot (r_i \cdot g_i + m_i) \cdot \left(\prod_{j=1}^{Z} z_j^{-1} \bmod p_i \right) \cdot \prod_{i' \neq i} p_{i'} \pmod{q} \\ = &\sum_{i=1}^{N} h_i \cdot \left(r_i + m_i \cdot (g_i^{-1} \bmod p_i) \right) \cdot \prod_{i' \neq i} p_{i'} \pmod{q}. \end{aligned}$$

Then, we test whether $\omega \in \mathbb{Z}$ is "small" compared to q or not. (Concretely, ω is small if $|\omega| < q \cdot 2^{-\nu-\lambda-2}$.) In particular, ω is small if $m_i = 0$ for all $i \in [N]$, as otherwise for some i, the relatively large $g_i^{-1} \bmod p_i$ term does not vanish in the final expression.

3.1.1 Using the CLT Graded Encoding Scheme

We make use of the CLT graded encoding scheme as follows:

1. We first construct a (non-randomized) matrix branching program MBP_f. Using this we can derive the multilinearity κ of the graded encoding scheme, and thus run InstGen to generate the public and secret parameters.

2. Using the g_is generated in the previous step, we can now construct a randomized matrix branching program $\widetilde{MBP}_{f,g_1}$ corresponding to the first "slot" of the plaintext space.

3. Finally, we encode the matrices element by element, where each element is the vector corresponding to the same-position matrix element in the randomized matrix branching program.

4. To evaluate on an input x, we multiply together the matrices (as specified by x and inp) along with the bookend vectors, and zero-test the resulting value.[6]

[6]An earlier version of our implementation did not compute multiplications modulo q as specified in the Mult procedure above, and also naively computed $\mathbf{s} \cdot \left(\prod_{j=1}^{m} B_{j,x_{\mathsf{inp}(j)}} \right) \cdot \mathbf{t}$ instead of the much more efficient $(((\mathbf{s} \cdot B_{1,x_{\mathsf{inp}(1)}}) B_{2,x_{\mathsf{inp}(2)}}) \cdots B_{m,x_{\mathsf{inp}(m)}}) \cdot \mathbf{t}$. We thank Daniel J. Bernstein, Andreas Hülsing, Tanja Lange, and Ruben Niederhagen for bringing these optimizations to our attention [8].

κ	α	β	ρ	ρ_f	η	ν	N
10	52	52	52	1060	1276	115	7273
14	52	52	52	1484	1700	115	9690
18	52	52	52	1908	2124	115	12107

Table 4: Concrete parameter settings for various settings of κ. The security parameter λ is fixed at 52. Note that these are the parameters for the obfuscations presented in Table 5.

3.1.2 Setting Parameters

The parameters given by Coron et al. [14] were chosen to support their particular use-case of multiparty key exchange, which requires a public mechanism for *re-randomizing* encodings. However, in the case of program obfuscation, there is no need to re-randomize. This allows us to optimize the parameters of the scheme to gain improved efficiency. That is, since Add and Mult give correct results as long as the growth of the noise terms r_i does not cause any of the respective numerators of an encoding to exceed the moduli p_i, we only need to set the size of the p_i sufficiently large to ensure we can perform all the operations required to evaluate the obfuscated program.

Similarly, we can bound ρ_f, the maximum size in bits of the noise terms r_i, at the maximum depth of multiplication (i.e., when κ initial encodings have been multiplied together) as $\rho_f = \kappa(2\lambda + 2)$. Using the parameters specified by Coron et al. would have given us $\rho_f \gg \kappa(5\lambda + 2)$. Reducing the size of ρ_f greatly improves efficiency, as it affects the size η of the primes p_i, which subsequently affects the vector dimension. Both of these parameters appear to be the main efficiency bottlenecks.

For completeness, we list all the parameters we use for the CLT scheme, and the reasoning behind our choices. Recall that λ denotes the security parameter. These settings are designed in order to achieve 2^λ security against known attacks.

- α (the bit-size of the primes g_i): λ, to prevent a brute-force attack on the plaintext space.
- β (the bit-size of the h_i values used to construct p_{zt}): λ, due to a GCD-based attack [14, §5.2].[7]
- ρ (the bit-size of the random r_i values): λ, to prevent a brute-force attack on the noise [12, 14].
- η (the bit-size of the primes p_i): $\rho_f + \alpha + 2\beta + \lambda + 8$, due to [14, Lemma 3]. Recall that ρ_f is a bound on the noise after performing all the necessary multiplications.
- ν (for zero-testing; ω is "small" if the ν most-significant bits of ω are all zeroes): $\alpha + \beta + 11$, due to [14, Lemma 3].
- N (the vector dimension of the plaintext space): $\eta \log_2 \lambda$, to prevent an orthogonal lattice reduction attack [14, §5.1].

Table 4 presents concrete numbers for various settings of κ with a fixed security parameter of 52.

3.2 Security of Our Implementation

To date, the security of general-purpose obfuscators has been argued in three general ways:

1. By showing the virtual black-box (VBB) property in an ideal model [5, 11],

[7] Recent work [22] demonstrates an attack on the zero-test parameter which necessitates increasing β to 4λ. However, that attack only applies to the full CLT scheme where there is a vector of zero-test parameters available to the attacker. As we (heuristically) only publish a single zero-test parameter, the presented attack does not apply in our setting.

#	CLT param gen					BP rand	Bookends enc		Layer enc		Obf time
	p_i/g_i	CRT	z_i	p_{zt}	total		avg	total	avg	total	
8	168.55	38.88	6.57	58.21	272.22	3.52	12.66	25.44	88.21	705.68	1007.40
12	428.94	76.85	12.88	116.73	635.42	11.88	30.16	60.45	346.25	4154.98	4863.87
16	944.25	164.68	24.65	212.89	1346.50	28.25	61.27	122.71	1359.19	21747.12	23246.24

Table 5: Microbenchmarks for point function obfuscation, with security parameter $\lambda = 52$. All timings are in seconds. '#' denotes the input size of the point function. 'p_i/g_i' denotes the time to generate random p_is and g_is, as well as compute $q = \prod p_i$. 'CRT' denotes the time to compute CRT coefficients for the p_i's, which is used when encoding. Likewise, 'z_i' and 'p_{zt}' denote the time required to generate z_is and the zero test element p_{zt}. 'BP rand' denotes the time required to randomize λ branching programs. (This is the only computationally expensive step not implemented in C.) 'Bookend enc' denotes the time to construct and encode the bookend vectors. 'Layer enc' denotes the time to encode each layer of the randomized matrix branching program. 'Obf time' denotes the *total* time to obfuscate (which includes the CLT parameter generation time and encoding times). See Table 4 for the concrete CLT parameters for these obfuscations.

#	**Obfuscation**			**Evaluation**	
	Time	Size	RAM	Time	RAM
8	1007.40	1.75	8.95	57.85	0.51
12	4863.87	9.07	21.09	213.13	1.10
16	23246.24	31.13	41.20	655.40	2.58

Table 6: Time, RAM usage, and size for obfuscating and evaluating different sized point functions. '#' denotes the input size of the point function. 'Time' denotes the running time (in seconds) to obfuscate/evaluate a given point function. 'Size' denotes the size (in GB) of the resulting obfuscation. 'RAM' denotes the maximum amount of memory (in GB) used during obfuscation/evaluation. See Table 4 for the concrete CLT parameters for these obfuscations.

2. By an exponential number of "semantically-secure graded encoding" assumptions [9, 25], or

3. By a single assumption on multilinear groups (and CLT encodings' quality as an instantiation of multilinear groups) plus complexity leveraging [19].

An ideal model proof of VBB security for our scheme would follow that of Barak et al. [5], except that we do not use dual-input branching programs in our implementation (as an efficiency optimization). Consequently, we do not know how to directly prove Claim 5 in their work [5, §6], namely that, to use their terminology, profiles of all single-input elements are complete. Nonetheless, we observe that an ideal model proof of VBB security for our scheme should go through in a straightforward way assuming the Bounded Speedup Hypothesis [11].

4 Evaluation

To evaluate our implementation, we obfuscated point functions of various sizes. We ran the code[8] on a 64-core machine using the Intel Xeon Processor E7-8830 and one terabyte of RAM, and we thank David Archer and Getty Ritter of Galois, Inc., for providing access to the machine. We limited the number of cores used to 32 and used non-oblivious branching programs.

[8]We used the code from commit 0e5ecab449d55a87c8aaadb99c1dbd45a0ba496a.

We obfuscated point functions taking 8, 12, and 16 bits of input, using security parameter 52 throughout.[9] These functions produce branching programs of widths 10, 14, and 18, respectively. Due to the time it took to obfuscate and evaluate each function, we only conducted a single run for each. Table 5 breaks down the cost of each step in the obfuscation process, Table 6 breaks down the cost (in time and RAM usage) of obfuscating and evaluating each function.

Note that the majority of time is spent encoding elements; the CLT parameter generation takes only 4–10% of the overall running time. Also note the large amount of RAM required to obfuscate. A large portion (90%+) of this is to compute and store the CRT coefficients which are needed to make encoding efficient. Finally, note that evaluation is much faster (between 17× and 35×) than obfuscation, and in addition uses much less RAM. This is "promising" in the sense that evaluation is more important "in practice" than obfuscation, as obfuscation can be thought of as a one-time operation, whereas parties can evaluate the obfuscated circuit any number of times. (Of course, evaluation is still too costly for nearly all realistic applications.)

5 Discussion and Conclusion

In this work, we provide the first implementation of cryptographic program obfuscation. We discuss both challenges encountered and optimizations made over the course of our development, and present a detailed evaluation of the performance of such obfuscators. Although we show that obfuscation is still far from practical, we are still able to obfuscate some "meaningful" programs. We hope that the availability of an obfuscation prototype will spur further work in both making obfuscation more practical as well as understanding the underlying security primitives better.

The evaluation results from Section 4 show that work still needs to be done before program obfuscation is usable in practice. In fact, the most complex function we obfuscate with meaningful security is a 16-bit point function, which contains just 15 AND gates. Even such a simple function requires about 9 hours to obfuscate and results in an obfuscation of 31.1 GB. Perhaps more importantly (since the obfuscation time is a one-time cost), evaluating the 16-bit point obfuscation on a single input takes around 3.3 hours. However, it is important to note that the fact that we can produce *any* "useful" obfuscations at all is surprising. Also, both obfuscation and evaluation are embarrassingly parallel and thus would run significantly faster with more cores (the largest machine we experimented on had 32 cores). Hopefully our implementation will help spur further research into making obfuscation more practical.

Regarding the main bottlenecks in efficiency, the most immediate one is the efficiency of the underlying graded encoding scheme, and in particular, the size η of the primes and the vector dimension N. Recall that κ represents the multilinearity of our graded encoded scheme, and grows linear in the length of the matrix branching program. The size of η, which affects both the encoding time and overall size of the obfuscation, is roughly $O(\kappa\lambda)$. Likewise, N, which also greatly affects the encoding time and size, is roughly $O(\eta \log \lambda) = O(\kappa\lambda \log \lambda)$. Unfortunately, these parameters need to be set as such due to known attacks [14]. A possible research direction would be to analyze the practical impact of these attacks, and determine whether these values can be reduced in any way without impacting the overall security.

Ideally, we would even prefer to have a mechanism for graded encoding schemes that functions akin to bootstrapping or modulus switching for fully homomorphic encryption, allowing the evaluator to control the growth of the various system parameters mid-computation while retaining

[9] We chose 52 simply because it is the smallest "reasonable" security parameter to experiment with.

security. Unfortunately, all known noise management techniques for fully homomorphic encryption implicitly rely on the plaintext space being public, while in the case of graded encoding schemes, the parameters that define the plaintext space — the primes g_i — must remain secret in order to hide the encoded values.

Another major bottleneck is the use of branching programs, which restrict us to obfuscating *single-bit Boolean formulas*. This (1) precludes computing more than $n - 1$ gates in one circuit, and (2) requires obfuscating the computation of every single output bit independently. While branching programs generated using Barrington's approach suffer from exponential growth with the circuit depth, the work of Ananth et al. [3] is a very important step in reducing the overhead due to branching programs. More recently, Zimmerman [32] devised a construction which avoids branching programs altogether; investigating the practical impact of this approach is left as future work.

Finally, to aid in the *cryptanalysis* of program obfuscation and the underlying cryptographic tool of graded encoding schemes, we have released a "challenge" to the community: we have published an obfuscated point function with a hidden secret[10]; the challenge is to determine the secret.[11] We hope this (and any future) challenge will inspire cryptanalysts to focus on attacking the cryptographic constructs used in program obfuscation, both to improve confidence in the constructions themselves as well as to better understand the concrete parameters needed for the underlying graded encoding scheme in order to achieve real-world security.

Acknowledgments

The authors thank Prabhanjan Ananth for identifying that Definition 2 from v1.0 stated that $MBP_f(x) \in \{0, 1\}$ without a proof. We also thank Daniel J. Bernstein, Andreas Hülsing, Tanja Lange, and Ruben Niederhagen for solving our first challenge, and in the process improving the running time of our evaluation procedure. Finally, we thank David Archer and Getty Ritter of Galois, Inc., for providing access to the machine used to run our evaluation experiments.

This work was supported in part by NSF awards #1111599 and #1223623. Work of Alex J. Malozemoff was conducted with Government support through the National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFG 168a, awarded by DoD, Air Force Office of Scientific Research.

[10]Note that there are much more efficient ways to obfuscate point functions (such as releasing the SHA hash of the point). The point here is that the construction presented in this work is "generic" (i.e., works for arbitrary polynomial-size circuits), and point functions are just one example of a function which can be obfuscated, and unfortunately the most "interesting" function we can currently obfuscate in a reasonable amount of time.

[11]See `https://www.dropbox.com/s/85d03o0ny3b1c0c/point-14.circ.obf.60.zip`, which contains an obfuscation of a 14-bit point function using security parameter 60. **Note:** The obfuscation has been broken! We thank Daniel J. Bernstein, Andreas Hülsing, Tanja Lange, and Ruben Niederhagen for finding the hidden point [8]. They found the point by greatly speeding up the evaluation procedure, among other optimizations, and then doing an enumeration over all 2^{14} possible inputs, reusing intermediate calculations to speed up the search. Using these techniques, they were able to learn the point in around 17.5 minutes using a 22 machine cluster [8, Table 6.4]. See `https://amaloz.github.io/obfuscation` for the release of any new challenges.

References

[1] The GNU Multiple Precision Arithmetic Library. https://gmplib.org. Accessed on May 11 2014.

[2] The OpenMP® API specification for parallel programming. https://openmp.org. Accessed on May 11 2014.

[3] P. Ananth, D. Gupta, Y. Ishai, and A. Sahai. Optimizing obfuscation: Avoiding Barrington's theorem. Cryptology ePrint Archive, Report 2014/222, Version 20140329:175740, 2014. https://eprint.iacr.org/2014/222.

[4] B. Barak, N. Bitansky, R. Canetti, Y. T. Kalai, O. Paneth, and A. Sahai. Obfuscation for evasive functions. In Y. Lindell, editor, *TCC 2014: 11th Theory of Cryptography Conference*, volume 8349 of *Lecture Notes in Computer Science*, pages 26–51, San Diego, CA, USA, Feb. 24–26, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/668.

[5] B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai. Protecting obfuscation against algebraic attacks. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 221–238, Copenhagen, Denmark, May 11–15, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/631.

[6] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18, Santa Barbara, CA, USA, Aug. 19–23, 2001. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2001/069.

[7] D. A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in $\mathbf{NC}^1$. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

[8] D. J. Bernstein, A. Hülsing, T. Lange, and R. Niederhagen. Bad directions in cryptographic hash functions. 2014.

[9] N. Bitansky, R. Canetti, Y. T. Kalai, and O. Paneth. On virtual grey box obfuscation for general circuits. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 108–125, Santa Barbara, CA, USA, Aug. 17–21, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2014/580.

[10] D. Boneh and M. Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 480–499, Santa Barbara, CA, USA, Aug. 17–21, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/642.

[11] Z. Brakerski and G. N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In Y. Lindell, editor, *TCC 2014: 11th Theory of Cryptography Conference*, volume 8349 of *Lecture Notes in Computer Science*, pages 1–25, San Diego, CA, USA, Feb. 24–26, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/563.

[12] Y. Chen and P. Q. Nguyen. Faster algorithms for approximate common divisors: Breaking fully-homomorphic-encryption challenges over the integers. In D. Pointcheval and T. Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 502–519, Cambridge, UK, Apr. 15–19, 2012. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2011/436.

[13] J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehlé. Cryptanalysis of the multilinear map over the integers. Cryptology ePrint Archive, Report 2014/906, 2014. https://eprint.iacr.org/2014/906.

[14] J.-S. Coron, T. Lepoint, and M. Tibouchi. Practical multilinear maps over the integers. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 476–493, Santa Barbara, CA, USA, Aug. 18–22, 2013. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/183.

[15] S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 1–17, Athens, Greece, May 26–30, 2013. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2012/610.

[16] S. Garg, C. Gentry, S. Halevi, and M. Raykova. Two-round secure MPC from indistinguishability obfuscation. In Y. Lindell, editor, *TCC 2014: 11th Theory of Cryptography Conference*, volume 8349 of *Lecture Notes in Computer Science*, pages 74–94, San Diego, CA, USA, Feb. 24–26, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/601.

[17] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual Symposium on Foundations of Computer Science*, pages 40–49, Berkeley, CA, USA, Oct. 26–29, 2013. IEEE Computer Society Press. Full version available at https://eprint.iacr.org/2013/601.

[18] C. Gentry, S. Gorbunov, and S. Halevi. Graded multilinear maps from lattices. Cryptology ePrint Archive, Report 2014/645, 2014. https://eprint.iacr.org/2014/645.

[19] C. Gentry, A. Lewko, A. Sahai, and B. Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. Cryptology ePrint Archive, Report 2014/309, 2014. http://eprint.iacr.org/2014/309.

[20] S. Hohenberger, A. Sahai, and B. Waters. Replacing a random oracle: Full domain hash from indistinguishability obfuscation. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages

201–220, Copenhagen, Denmark, May 11–15, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/509.

[21] J. Kilian. Founding cryptography on oblivious transfer. In *20th Annual ACM Symposium on Theory of Computing*, pages 20–31, Chicago, Illinois, USA, May 2–4, 1988. ACM Press.

[22] H. T. Lee and J. H. Seo. Security analysis of multilinear maps over the integers. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 224–240, Santa Barbara, CA, USA, Aug. 17–21, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2014/574.

[23] C. Linn and S. K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM CCS 03: 10th Conference on Computer and Communications Security*, pages 290–299, Washington D.C., USA, Oct. 27–30, 2003. ACM Press.

[24] X. Luo, P. Zhou, E. W. W. Chan, W. Lee, R. K. C. Chang, and R. Perdisci. HTTPOS: Sealing information leaks with browser-side obfuscation of encrypted flows. In *ISOC Network and Distributed System Security Symposium – NDSS 2011*, San Diego, California, USA, Feb. 6–9, 2011. The Internet Society.

[25] R. Pass, K. Seth, and S. Telang. Indistinguishability obfuscation from semantically-secure multilinear encodings. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 500–517, Santa Barbara, CA, USA, Aug. 17–21, 2014. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/781.

[26] A. Sahai and B. Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In D. B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 475–484, New York, NY, USA, May 31 – June 3, 2014. ACM Press. Full version available at https://eprint.iacr.org/2013/454.

[27] A. Sahai and M. Zhandry. Obfuscating low-rank matrix branching programs. Cryptology ePrint Archive, Report 2014/773, 2014. https://eprint.iacr.org/2014/773.

[28] M. Sauerhoff, I. Wegener, and R. Werchner. Relating branching program size and formula size over the full binary basis. In C. Meinel and S. Tison, editors, *STACS 99: 16th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 57–67, Trier, Germany, Mar. 4–6, 1999. Springer, Berlin, Germany.

[29] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *ISOC Network and Distributed System Security Symposium – NDSS 2008*, San Diego, California, USA, Feb. 10–13, 2008. The Internet Society.

[30] W. Stein and D. Joyner. SAGE: System for algebra and geometry experimentation. *ACM SIGSAM Bulletin*, 39(2):61–64, 2005.

[31] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In V. Atluri and C. Díaz, editors, *ESORICS 2011: 16th European Symposium on Research in Computer Security*, volume 6879 of *Lecture Notes in Computer Science*, pages 210–226, Leuven, Belgium, Sept. 12–14, 2011. Springer, Berlin, Germany.

[32] J. Zimmerman. How to obfuscate programs directly. Cryptology ePrint Archive, Report 2014/776, 2014. https://eprint.iacr.org/2014/776.

Changelog

- Version 2.0 (February 10, 2015):
  - Removed randomized matrix branching program figure.
  - Removed section on "attack" which leaks g_1 if one slot is filled. It appears that leaking g_1 does not break the security of obfuscation [13, Footnote 5].
  - Reworded first paragraph of Section 2.1.1 and removed footnote.
  - Removed comment about the lattice-based multilinear map [18] in Section 3.1, which has since been broken.
  - Added Lemma 1 to Section 2.1.1 and updated Section 2.2 by fixing Definition 2 and adding Lemma 2.
  - Added the fact that the challenge (cf. Section 5) has been broken.
  - Added footnote in Section 3.1.1 about the improved evaluation procedure.
  - Updated link to code.
  - Updated Acknowledgments.
- Version 1.0 (October 1, 2014): First release.