Practical Implementations of Program Obfuscators for Point Functions

Giovanni Di Crescenzo
Lisa Bahler, Brian Coan
Applied Communication Sciences
Basking Ridge, NJ, 07920, USA
Email: gdicrescenzo@appcomsci.com
Email: lbahler@appcomsci.com
Email: bcoan@appcomsci.com

Yuriy Polyakov, Kurt Rohloff
New Jersey Institute of Technology
Newark, NJ, 07102, USA
Email: polyakov@njit.edu
Email: rohloff@njit.edu

David B. Cousins
Raytheon BBN Technologies
Middletown, RI, 02842, USA
Email: dcousins@bbn.com

***Abstract*—Point function obfuscators have recently been shown to be the first examples of program obfuscators provable under hardness assumptions commonly used in cryptography. This is remarkable, in light of early results in this area, showing impossibility of a single obfuscation solution for all programs. Point functions can be seen as functions that return 1 if the input value is equal to a secret value stored in the program, and 0 otherwise. In this paper, we select representative point function obfuscators from the literature, state their theoretical guarantees, and report on their (slightly) optimized implementations. We show that implementations of point function obfuscators, satisfying different obfuscation notions, can be used with practical performance guarantees. Notable results due to our design and implementation optimizations are very fast obfuscators based on group theory, and obfuscators based on lattice theory with running time below 10 seconds.**

I. Introduction

The problem of program obfuscation is recently attracting a significant amount of research in the cryptography literature, as general solutions to this problem seem to have great application potential. While early results in the area showed impossibility with respect to constructing a single obfuscator for all polynomial-time programs [1], most recent results show the possibility of constructing obfuscators for restricted families of functions, such as point functions (and extensions of them), under commonly used hardness assumptions. In the current research literature, there are a few theoretical definitions of program obfuscation (see, e.g., [1], [2]), and several constructions of point functions obfuscators based on commonly used hardness assumptions, with different performance and security features (see, e.g., [3], [4], [5], [2]).

In this paper, we first of all consider the theory-oriented definition of program obfuscators and specialize it to a practice-oriented version that is more suited for implementation, especially with respect to program obfuscators for a large class of functions, including point functions. Then, we consider 4 of the most used security notions for program obfuscators, based on concepts like simulation of the adversary's view or output, and indistinguishability between an obfuscation of a real or random secret. We sort out the intricate literature on this sub-area to select 5 interesting point function obfuscators from [3], [4], [5], [2], including (a) at least one satisfying each of these security notions; (b) at least one that is practically efficient and provably secure based on group-thery and no random oracles; (c) at least one based on a lattice-theory assumption, which is resistant to quantum computation attacks. We then report on their implementations, applying, wherever possible, both design and coding optimizations. Among the former type of optimizations, we replace the computations of certain values with different and more efficient computations of almost equally distributed values. In one case, we do not maintain a similar distribution, but we can still perform the optimization at the cost of a (much) stronger hardness assumption. Among the second type of optimizations, in group-theory obfuscators, we replace conventional modular exponentiation (often, the most expensive operations in group-theoretic cryptography) with modular exponentiation via pre-processing, combined with Montgomery multiplication; in lattice-theory obfuscators, we use parallelization and memory access techniques. Overall, we show that implementations of point function obfuscators, satisfying different obfuscation notions, can be used with practical performance guarantees. Notable results due to our design and implementation optimizations are, based on inexpensive computing resources:

1. a point function obfuscator based on a group-theory hardness assumption, requiring storage less than 1MB, and running time $< 0.0139s$ for the obfuscated program;
2. a point function obfuscator (in 2 variants) based on a lattice-theory hardness assumption, and satisfying:
   running time $< 2.6s$ and storage $<$ 3.1GB, or
   running time $< 8s$ and storage $\leq$ 68B,

where the latter variant is based on a strong hardness assumption (which is true if a pseudo-random generator behaves like a random oracles). Lessons learned include: (a) our definition of program obfuscators, based on parameter passing, can be more relevant to implementation than previous theoretical definitions, based on maps between circuits; (b) shifting computation to a pre-processing stage is a valid optimization paradigm for typical applications, where the obfuscator is run much less frequently than the obfuscated program.

II. Definitions and Preliminaries

We start this section by defining the family of point functions. We then recall the theory-oriented definition of program obfuscators, and modify it slightly into a practice-oriented definition that better fits a large class of obfuscator implementations (including ours for point functions).

Point functions. We consider *families of functions* as families of maps from a domain to a range, where maps may be parameterized by some values chosen according to some distribution on a parameter set. Let pF be a family of functions $f_{par} : Dom \to Ran$, where $Dom = \{0,1\}^n$, $Ran = \{0,1\}$, and each function is parameterized by value par from a parameter set $Par = \{0,1\}^n$, for some length parameter n. We say that pF is the *family of point functions* if on input $x \in Dom$, and *secret value* $y \in Par$, the point function f_{par} returns 1 if $x = y$ and 0 otherwise.

The theory-oriented definition. We say that a class of functions F admits an obfuscator Obf if Obf is an efficient algorithm that, on input a description of function $f \in F$ and/or a circuit C_f computing $f \in F$, returns an (obfuscated) circuit oC_f, such that the following two properties are satisfied:

1. (Almost exact functionality): For all f in F, and inputs x, it holds that $oC_f(x) = f(x)$, except possibly with very small probability.
2. (Polynomial slowdown): There exists a polynomial p such that for all f in F, the running time of oC_f is $\leq p(|C_f|)$, where $|C_f|$ denotes the size of circuit C_f.

A practice-oriented definition. In practice, it can be unnecessarily complex to implement an obfuscator taking as input a circuit that computes function f, and returns as output another (obfuscated) circuit. Therefore, we perform syntax changes to obtain a definition involving simpler algorithms, from the point of view of implementation, and semantically-equivalent for a large class of function families, including point functions. Specifically, we view an obfuscator as a pair of efficient algorithms: an *obfuscation generator* $genO$ and an *obfuscation evaluator* $evalO$, with the following syntax. On input function parameters $fpar$, including a description of function $f \in F$, $genO$ returns generator output $gpar$. On input a description of function $f \in F$, generator output $gpar$, and evaluator input x, $evalO$ returns evaluator output y. The pair of algorithms $(genO, evalO)$ satisfies the following two properties:

1. (Almost exact functionality): For any f in F, with function parameters $fpar$, and any input x, the equality $y = f(x)$ holds with probability $1-\delta$, for some negligible (or very small) δ, where y is generated by the following probabilistic steps:
   $gpar \leftarrow genO(fpar)$,
   $y \leftarrow evalO(gpar, x)$.
2. (Polynomial slowdown): There exists a polynomial p such that for all f in F, the running time of $evalO$ is $\leq p(|f|)$, where $|f|$ denotes the size of the (smallest) boolean circuit computing f.

Security notions. Obfuscators (in both the theory-oriented and practice-oriented definition) can satisfy any one of the following different obfuscation security notions (which have to be valid for all inputs, all efficient adversary algorithms, and except with negligible probability):

1. *adversary view black-box simulation* [1]: the adversary's view of the (executable) program $evalO(gpar, \cdot)$ can be produced by an efficient algorithm, called the simulator, with black-box access to function f;
2. *adversary output black-box simulation* [1]: the adversary's output bit, on input the (executable) program $evalO(gpar, \cdot)$, can be guessed by an efficient algorithm, called the simulator, with black-box access to function f;
3. *strong indistinguishability* [2]: the adversary's output, on input an obfuscation of the program computing function f, is indistinguishable from the adversary's output, on input an obfuscation of the program computing a random function from family F.
4. *real-vs-random indistinguishability* [2]: the adversary cannot distinguish an obfuscation of the program computing function f from an obfuscation of the program computing a random function from family F.

It is not hard to see that an obfuscator satisfying notion 1 also satisfies notions 2,3,4. Moreover, in [2] it was proved that, for the family of point functions, an obfuscator satisfying notion 3 also satisfies notion 4, and that the converse may not hold.

Known Point Function Obfuscators. We summarize some bibliography on point function obfuscators. A first obfuscator, satisfying adversary view black-box simulation, was given in [3], under the random oracle assumption. A previous result of [4], although formulated as a oracle hashing scheme, can be restated as an obfuscator satisfying strong indistinguishability under the Decisional Diffie Hellman assumption. The obfuscator in [5] satisfies adversary output black-box simulation under the existence of a strong type of one-way permutations. Finally, more obfuscators were given in [2], and one of these, based on any deterministic encryption scheme, satisfies real-vs-random indistinguishability, and happens to have several instantiations. This is due to the fact that deterministic encryption schemes can be built using lattices [6] or lossy trapdoor functions [7], and the latter have been built using any one of many group-theoretic assumptions (see, e.g., [8]). In the next 5 sections, we report on our implementation of (optimized versions of) the obfuscators in [3], [4], [5], one obtained combining [2], [7], [8], and two variants of one obtained combining [2], [6].

III. An Obfuscator from Cryptographic Hashing

We describe a first obfuscator, denoted as (genO_1,evalO_1), for the family of point functions, based on collision-resistant hashing, modeled in the security analysis as random oracles.

Informal description: This construction is based on a technique often used to store passwords in certain operating systems, which has recently been re-interpreted as an obfuscation of the password verification algorithm. Informally, it goes as follows. The obfuscation generator first concatenates the secret value with a sufficiently-long random string, then applies a cryptographic hash function on this concatenated value, and finally returns the computed hash tag. The obfuscation evaluator does essentially the same computations on the input point (instead of the secret value), and returns 1 if the computed hash tag is equal to the hash tag returned by the obfuscation generator or 0 otherwise. A formal description follows.

Formal description: Let $|$ denote string concatenation, and let H denote a collision-resistant hash function (i.e., a function mapping an arbitrary-length input string to a fixed-length output string, such that it is hard for any efficient adversary to find two preimages of the same function output). A formal description of (genO$_1$,evalO$_1$) follows.

Input to genO$_1$: security parameters $1^n, 1^{\ell_0}$, length parameter 1^ℓ, secret value $z \in \{0,1\}^\ell$,

Instructions for genO$_1$:

1) Uniformly and independently choose $r \in \{0,1\}^{\ell_0}$
2) Compute $v = H(r|z)$, where $v \in \{0,1\}^n$
3) Set $gpar = (r, v)$ and return: $gpar$.

Input to evalO$_1$: security parameter 1^n, length parameter 1^ℓ, $r \in \{0,1\}^{\ell_0}$ and $v \in \{0,1\}^n$, input value $x \in \{0,1\}^\ell$

Instructions for evalO$_1$:

1) compute $v' = H(r|x)$, where $v' \in \{0,1\}^n$
2) if $v' = v$ return 1 else return 0

Theoretical result. Assuming H behaves like a random oracle, (genO$_1$,evalO$_1$) is an obfuscator of the family of point functions, satisfying the adversary view black-box simulation notion. In [3], it was first stated that if H behaves like a random oracle, the value $H(z)$ is a (not composable) obfuscation of secret value z. The known technique of concatenating z with a sufficiently long random string r before hashing makes the scheme composable (i.e., secure even if executed many times, on input related secret strings).

Parameter and primitive settings. Parameter ℓ can be set as needed in the specific application. Parameter n can be set as ≥ 256, to guarantee security against generic "birthday-type" collision attacks; our implementation sets it $= 512$. Parameter ℓ_0 is also set as $= 512$. H can be any cryptographic hash function that is believed to be secure enough in light of a significant amount of cryptanalysis efforts; thus, including SHA2 and SHA3. Our implementation uses SHA512, which is SHA2 when set it to return $n = 512$ bits as output.

Performance analysis. We used a Dell 2950 processor (Intel(R) Xeon(R) 8 cores: CPU E5405 @ 2.00GHz, 16GB RAM), without parallelism. Performance numbers for our implementation of (genO$_1$, evalO$_1$) are summarized below.

Table 1: Performance of scheme (genO$_1$, evalO$_1$).

Input length ℓ	Time(genO$_1$)	Time(evalO$_1$)
2048	.0004 s	.0002 s
16384	.0004 s	.0002 s
131072	.0005 s	.0003 s
1048576	.0012 s	.0011 s

Not surprisingly, this obfuscator is extremely efficient, even without optimizations. It is also useful as a comparison standard to evaluate the efficiency of other techniques. Still, constructions provably secure without the random oracle assumption are much more desirable, when possible. Even though practitioners sometimes rely on this assumption, we recall that in [9], it was proved that the random oracle assumption is in general unreliable, in the following sense: there are specific cryptographic schemes secure assuming a random oracle exists, but which become almost certainly insecure for any instantiation of the random oracle from a polynomial-time computable family of functions.

IV. An Obfuscator based on Decisional DH

In this section we describe an obfuscator, denoted as (genO$_2$,evalO$_2$), for the family of point functions, based on the Decisional Diffie-Hellman (DH) assumption. We first briefly recall this assumption and the notions of faster computation of modular exponentiation via preprocessing, and then describe the obfuscator and its properties.

Decisional DH assumption: Let p and q be primes such that $p = 2q + 1$ and $|q| = n + 1$, and let g be a generator of the q-order subgroup G_q of $\mathbb{Z}_p$. The Decisional DH problem asks to efficiently distinguish, given p, q, g, a triple $(g^a \bmod p, g^b \bmod p, g^{ab} \bmod p)$ from a triple $(g^a \bmod p, g^b \bmod p, g^c \bmod p)$, for uniformly and independently chosen elements a, b, c from $\mathbb{Z}_q$. The Decisional DH assumption says that no efficient algorithm can distinguish these two distributions, except with negligible probability.

Modular exponentiation with preprocessing: A pair of algorithms (ModExpPreproc, ModExpCompute) is used to denote a scheme for faster computation of modular exponentiation, using preprocessing, defined as follows. On input a base u and a modulus p, the algorithm ModExpPreproc computes some auxiliary information $aux_{u,p}$. On input a base u, a modulus p, an exponent d, and auxiliary information $aux_{u,p}$, the algorithm ModExpCompute computes a value v, such that $v = u^d \bmod p$. Here, the goal is to use auxiliary information $aux_{u,p}$ to compute v faster than using a standard modular exponentiation algorithm. A survey of such methods was given in [10]. Some of these methods, and in particular the ones selected here, reduce exponentiation to an arbitrary exponent to a sequence of multiplications of simpler and pre-computed exponentiations to specific exponents. We further optimized one of this method by performing Montgomery modular multiplications.

Informal description: The basic idea of the scheme is as in [4]: first, the obfuscation generator computes a first value as a random power of generator g, a second value as an exponentiation of the first value to the secret value, and returns both values; then, the obfuscation evaluator exponentiates the first value to the input point (instead of the secret value), and returns 1 if the computed group element is equal to the second value or 0 otherwise. We extend this basic idea by replacing one modular exponentiation with a random subgroup value computable using only one modular multiplication in the chosen group, and by computing all other exponentiations by carefully distributing the technique of exponentiation with preprocessing between the obfuscation generator and evaluator. We now give a formal description of ($\mathrm{genO_2}$,$\mathrm{evalO_2}$).

Input to genO$_2$: length parameter 1^n, secret value $z \in \{0,1\}^n$

Instructions for genO$_2$:

```
1) Randomly choose primes $p, q$ such that $p = 2q+1$, $|q| = n+1$
2) Rand. choose generator $g$ of $q$-order subgroup $G_q$ of $\mathbb{Z}_p$
3) Randomly choose $u \in G_q$ and $r \in \{0,\ldots,q-1\}$;
4) Compute $(aux_{u,p}) = \mathrm{ModExpPreproc}(u,p)$
5) Consider $z$ as an element of $G_q$
6) Compute $v = \mathrm{ModExpCompute}(u,p,z,aux_{u,p})$
7) Return: $(aux_{u,p},(u,v))$.
```

Input to evalO$_2$: security parameter 1^n, input value $x \in \{0,1\}^n$ and the output from *genO*, containing auxiliary information $aux_{u,p}$ for faster computation of exponentiation modulo p in base u, and pair (u,v).

Instructions for evalO$_2$:

```
1) Consider $x$ as an element of $G_q$
2) Compute $v' = \mathrm{ModExpCompute}(u,p,x,aux_{u,p})$
3) If $v' = v$ then return: 1 else return: 0.
```

Theoretical result. Under the Decisional DH assumption, ($\mathrm{genO_2}$,$\mathrm{evalO_2}$) is an obfuscator of the family of point functions with (almost) uniformly distributed secret values, according to strong indistinguishability obfuscation notion of [2] (which generalizes the oracle hashing secrecy from [4]). This follows by a generalization of the proof from [4] that the basic version of this construction is an oracle hashing scheme for random secret inputs under the Decisional DH assumption.

Parameter and primitive setting. Parameter n can be set as $=$ 2048, to guarantee security against known discrete logarithm finding algorithms. In algorithm $initO_2$, to perform the generation of prime p, along with prime q, and of generator g for the q-order subgroup G_q of $\mathbb{Z}_p$, we used procedures from the OpenSSL library. The scheme (ModExpPreproc, ModExpCompute) can be any pair of algorithms from [10]. In one such schemes, algorithm ModExpPreproc precomputes exponentiations modulo p in the same base u and for specific exponents (e.g., powers of 2 and combinations of thems). Later, based on these pre-computed values, algorithm ModExpCompute computes exponentiations modulo p in the same base u and for an arbitrary exponent, as a suitable sequence of multiplications modulo p.

Performance analysis. We used the same processor as for the previously described obfuscator. Schemes used to perform modular exponentiation with preprocessing exhibit tradeoffs between stored data (containing pre-computed exponentiations modulo specific exponents) and running time of the algorithm (taking as input an arbitrary exponent). As an example tradeoff setting, by keeping stored data less than 1MB, our implementation achieved:

1. runtime of .0734s for $\mathrm{genO_2}$, including .0465s for precomputation related to exponentiation and .0269s for the rest of the obfuscation;
2. runtime of .0139s for $\mathrm{evalO_2}$.

V. An Obfuscator based on Discrete Logarithms

In this section we present an obfuscator, denoted as ($\mathrm{genO_3}$,$\mathrm{evalO_3}$), for the family of point functions, based on the Discrete Logarithm assumption. First, we briefly recall this assumption, and then describe the obfuscator and its properties.

Discrete Logarithm assumption: Let p, q be primes such that $p = 2q+1$, $|q| = n+1$, and let g be a generator of the group $\mathbb{Z}_p^*$. The Discrete Logarithm problem asks to compute x, given p, g, y such that $y = g^x \bmod p$, for a random $x \in \{0,\ldots,p-1\}$. The Discrete Logarithm assumption says that no efficient algorithm can compute x with more than negligible, in n, probability. For any $x \in \{0,\ldots,p-1\}$, the function $\mathrm{MostSigBit}(x)$ returns 0 if $1 \le x \le (p-1)/2$ and 1 if $(p-1)/2 < x \le p-1$. As for the obfuscator from Section IV, we use scheme (ModExpPreproc, ModExpCompute) for faster computation of modular exponentiation.

Informal and formal description: The starting idea of this scheme is as in [5]. The obfuscation generator works in $3n$ iterations, and computes at each iteration the output of a one-way permutation on input the output from the previous iteration, and a hard-core bit associated with the current evaluation. The input in the first iteration is the secret value z. At the end of all iterations, it returns the $3n$ hard-core bits. The obfuscation evaluator performs the same computation of $3n$ hard-core bits, using as input in the first iteration the input value x. At the end, it returns 1 if the computed hard-core bits are equal to those returned by the obfuscation generator or 0 otherwise. We instantiate this basic idea by setting the one-way permutation as exponentiation modulo a prime p (which is often conjectured to be a one-way permutation over $\mathbb{Z}_p^*$), and by setting the hard-core bit as the most significant bit of the discrete logarithm exponent. We then compute all modular exponentiations by carefully distributing the technique of modular exponentiation with preprocessing between the obfuscation generator and evaluator, similarly as done for our obfuscator in Section IV.

We now give a formal description of ($\mathrm{genO_3}$,$\mathrm{evalO_3}$).

Input to genO$_3$: length parameter 1^n, secret value $z \in \{0,1\}^n$.

Instructions for genO$_3$:

```
1) Randomly choose prime $p \in \{0,1\}^{n+1}$
2) Randomly choose a generator $g$ of $\mathbb{Z}_p^*$
3) Compute $aux_{g,p} = \mathrm{ModExpPreproc}(g,p)$
4) Consider $z$ as an element of $\mathbb{Z}_p^*$ and set $w_1 = z$
5) For $i = 1,\ldots,3n$,
       compute $w_{i+1} = \mathrm{ModExpCompute}(g,p,w_i,aux_{g,p})$
       compute $v_i = \mathrm{MostSigBit}(w_{i+1})$
6) Set $v = (v_1|\cdots|v_{3n})$
7) Return: $(aux_{g,p},v)$.
```

Input to evalO$_3$: security parameter 1^n, input value $x \in \{0,1\}^\ell$ and the output from *genO*, containing auxiliary information $aux_{g,p}$ for faster computation of exponentiation modulo p in base g, and $3n$-bit vector v.

Instructions for evalO$_3$:

```
1) Consider $x$ as an element of $\mathbb{Z}_p^*$ and set $w'_1 = x$
2) For $i = 1, \ldots, 3n$,
       compute $w'_{i+1} = \text{ModExpCompute}(g, p, w'_i, aux_{g,p})$
       compute $v'_i = \text{MostSigBit}(w'_{i+1})$
3) Set $v' = (v'_1|\cdots|v'_{3n})$
4) If $v' = v$ then return 1 else return: 0.
```

Theoretical results. Under the Discrete Logarithm assumption, (genO$_3$,evalO$_3$) is an obfuscator of the family of point functions, according to (a weak version of) the adversary output black-box simulation notion [1]. This follows by combining the following: (1) the proof in [5] that the generalized construction is an obfuscator under a strong one-way permutation assumption; (2) an instantiation of the strong one-way permutation using exponentiation modulo a large prime, based on the Discrete Logarithm assumption; (3) an instantiation of the hard-core predicate for the one-way permutation using the most significant bit, based on the Discrete Logarithm assumption and a result from [11].

Parameter and primitive setting. To guarantee security against known discrete logarithm finding algorithms, we set $n = 2048$. In algorithm $initO_3$, to perform the generation of prime p and generator g for $\mathbb{Z}_p^*$, we used procedures from the OpenSSL library. The scheme (ModExpPreproc, ModExpCompute) can be any scheme from [10].

Performance analysis. We used the same processor as for the previously described obfuscators. Performance numbers for our implementation of (genO$_3$, evalO$_3$), when $n = 2048$, are summarized below.

Table 2: Performance of scheme (genO$_3$, evalO$_3$).

Storage (in Bytes) ℓ	Time(genO$_3$)	Time(evalO$_3$)
4160	110.5831 s	110.6973 s
648208	60.5575 s	60.9090 s
969776	45.5783 s	46.2278 s
2414944	28.7626 s	28.5533 s
20509168	19.8536 s	15.7760 s
65872016	29.4014 s	13.1885 s
120233632	44.9399 s	12.4013 s
219817984	76.4558 s	12.0895 s
406573616	141.5508 s	12.4832 s

Some of the theoretically desirable design features of this obfuscator (e.g., the provability of a strong obfuscation property under general hardness assumptions, and the production of only one output bit per execution of the underlying cryptographic primitive) certainly did not help achieving high performance. Still, this is the only point function obfuscator that satisfies this important obfuscation notion without a random oracle assumption, and thus it was of interest to optimize it. Using less than 0.22GB storage, we achieved just above 12s of evaluation time (2nd to last line in Table 2), which is almost one order of magnitude faster than the unoptimized evaluation time (first line in Table 2).

VI. An Obfuscator from Decisional Residuosity

In this section we present an obfuscator, denoted as (genO$_4$,evalO$_4$), for the family of point functions, based on the Decisional Residuosity (DR) assumption. We first briefly recall this assumption, and then describe the obfuscator and its properties.

DR assumption: Let p, q be ℓ-bit primes and let $N = pq$. The DR (modulo N^2) problem asks to efficiently distinguish, given N, a random value in $\mathbb{Z}_{n^2}^*$ from a random n-th residue in $\mathbb{Z}_{n^2}^*$ (i.e., a value $y = x^N \bmod N^2$, for some random $x \in \mathbb{Z}_{n^2}^*$). The DR assumption says that no efficient algorithm can distinguish the two distributions, except with negligible probability.

Informal description: The starting idea of this scheme combines results in [2], [7], where a point function obfuscator is constructed from any deterministic encryption [2], and the latter is constructed from any pairwise-independent hash function and lossy trapdoor function [7]. Finally, we use the construction of a lossy trapdoor function from [8], in turn based on Damgaard-Jurik's cryptosystem [12] (a variant of Paillier's cryptosystem [13]). The resulting obfuscation evaluator only performs two modular exponentiations, and we can compute one of them using preprocessing, similarly as done in Section IV.

Formal description: For any x, let $minH(x)$ denote the min entropy of string x; that is, x is sampled from a distribution that returns no value with probability $> 2^{-t}$. We now give a formal description of (genO$_4$,evalO$_4$).

Input to genO$_4$: security parameter 1^n, length parameter 1^ℓ, accuracy parameter ϵ, secret value $z \in \{0,1\}^\ell$, and min-entropy parameter t, such that $minH(z) \geq t \geq n + 2\epsilon$, and $\ell = (n-2)s + n/2 - 1$, for some integer $s \geq 1$.

Instructions for genO$_4$:

```
1) Randomly choose primes $p, q$ such that $|p| = |q| = n/2$
2) Set $N = pq$
3) Randomly choose $r \in \mathbb{Z}_N^*$
4) Set $c = (1+N)r^{N^s} \bmod N^{s+1}$
5) Write $z$ as $(u_0, u_1)$, where $u_0 \in \mathbb{Z}_{N^s}$ and $u_1 \in \mathbb{Z}_N^*$
6) Randomly choose pairwise indep. hash function $piH : \mathbb{Z}_{n^s} \times \mathbb{Z}_n^* \rightarrow \mathbb{Z}_{n^s} \times \mathbb{Z}_n^*$
7) Set $(v_0, v_1) = piH(u_0, u_1)$, where $v_0 \in \mathbb{Z}_{N^s}$ and $v_1 \in \mathbb{Z}_N^*$
8) Set $aux_{c,N^{s+1}} = \text{ModExpPreproc}(c, N^{s+1})$
9) Set $w_0 = \text{ModExpCompute}(c, N^s, v_0, aux_{c,N^{s+1}})$
10) Set $w = w_0(v_1)^{N^s} \bmod N^{s+1}$
11) Return: $(t, piH, \epsilon, c, N, s, w)$
```

Input to evalO$_4$: security parameter 1^n, length parameter 1^ℓ, input value $x \in \{0,1\}^\ell$ and the output from $genO_4$, containing min-entropy parameter t, pairwise independent hash function piH, accuracy parameter ϵ, auxiliary information $aux_{c,N^{s+1}}$ for faster computation of exponentiation modulo N^{s+1} in base c, value $c \in \mathbb{Z}_{N^{s+1}}$, integer N, integer s, and value $w \in \mathbb{Z}_{N^{s+1}}$.

Instructions for evalO$_4$:

```
1) Write $z$ as $(u'_0, u'_1)$, where $u_0 \in \mathbb{Z}_{N^s}$ and $u_1 \in \mathbb{Z}_N^*$
2) Set $(v'_0, v'_1) = piH_2(u'_0, u'_1)$, where $v'_0 \in \mathbb{Z}_{N^s}$ and $v'_1 \in \mathbb{Z}_N^*$
3) Set $w'_0 = \text{ModExpCompute}(c, N^{s+1}, v'_0, aux_{c,N^{s+1}})$
```

4) Set $w' = w'_0(v'_1)^{N^s} \bmod N^{s+1}$
5) If $w' = w$ then return 1 else return 0.

Theoretical properties. Under the Decisional Residuosity (modulo N^{s+1}) assumption, the pair (genO$_4$,evalO$_4$) is an obfuscator for the family of point functions, according to the real-vs-random obfuscation indistinguishability definition of [2], and where the point has min entropy at least $n + 2\epsilon$. This is obtained by combining the following: (1) the proof in [2] that an obfuscator based on any deterministic encryption scheme satisfies the real-vs-random indistinguishability obfuscation notion; (2) the result in [7] saying that a deterministic encryption scheme can be obtained by applying a pairwise-independent hash function to the input, and then a lossy trapdoor function to its output; (3) the construction in [8] of a lossy trapdoor function based on Damgaard-Jurik's cryptosystem [12] (a variant of Paillier's cryptosystem [13]). The pairwise-independent hash function is used to apply the Leftover Hash Lemma from [14].

Parameter and primitive setting. Parameter s can be set depending on what ℓ is needed in the specific application, and our implementation only requires an essentially unrestricted $\ell < 2^{31}$. Parameter ϵ can be set as 128, to guarantee that the statistical distance between the distribution of piH's output and a uniformly distributed string of the same length, is $\leq 2^{-128}$. Parameter t can be set as $t = n + 2\epsilon$. For the generation of $n/2$-bit primes p, q, we used procedures from the OpenSSL library. Function piH can be any pairwise-independent hash function, including the 1-degree polynomial over $GF(2^\ell)$ [15].

Performance analysis. We used the same processor as for the previously described obfuscators. As an example setting balancing a storage-computation tradeoff, by keeping less than 2.4MB stored data, our implementation achieved runtime of .1317s for genO$_4$ and of .1005s for evalO$_4$.

VII. An Obfuscator based on the LWR problem

In this section we present two variants of an obfuscator, denoted as (genO$_{5,i}$,evalO$_{5,i}$), for $i = 0, 1$, for the family of point functions, based on a recently introduced problem on lattices, called Learning With Rounding (LWR). We first briefly recall this problem and its related assumption, and then present the obfuscator and its properties.

Learning With Rounding assumption. Let A^T denote the transpose of matrix or vector A. Let p, q be primes, and, for any vector $v = (v_1, \ldots, v_m)$, let $\lfloor v \rceil_p$ denote the vector whose i-th element is the closest integer to $(q/p)v_i$, for $i = 1, \ldots, m$. Let $\mathbb{Z}_q^{n,m}$ denote the set of $n \times m$-matrix with elements in $\{0, \ldots, q-1\}$, and let $\mathbb{Z}_q^n = \mathbb{Z}_q^{n,1}$, for any positive integers n, m. Consider the following two distributions:

1. $D_1 = \{A \leftarrow \mathbb{Z}_q^{n,m}; s \leftarrow \mathbb{Z}_q^n; b = \lfloor A^T s \rceil_p : (A, b)\}$
2. $D_1 = \{A \leftarrow \mathbb{Z}_q^{n,m}; b \leftarrow \mathbb{Z}_p^m : (A, b)\}$

The LWR problem asks to efficiently distinguish, whether a sample (A, b) came from D_0 or D_1. The LWR assumption says that the distributions D_0 and D_1 are indistinguishable to any efficient algorithm, except with negligible probability. In [16] it is conjectured that in light of known algorithmic attacks, the LWR assumption seems to hold if $q/p \geq \sqrt{n}$ is an integer and p is polynomial in n.

A. Description of a First Variant

Informal Description. First, we use the obfuscator from any deterministic encryption scheme, as described in [2], and then instantiate the deterministic encryption scheme with the one from [6], based on the Lattice with Rounding assumption. We performed two design optimizations to this construction: first, the key generation for the deterministic encryption algorithm only generates the public key, and not the secret key, since the latter is never used by the obfuscator; second, we generate a uniformly distributed public key, instead of the one returned by the scheme in [6], in turn based on lattice key generation approaches from [17]. The latter simplification is possible since the distribution of the public key was proved in [17] to be statistically indistinguishable from uniform.

Formal description: Let $\cdot$ denote matrix/vector product $\bmod q$. We now give a formal description of (genO$_{5,0}$,evalO$_{5,0}$).

Input to genO$_{5,0}$: dimension parameters $1^n, 1^m$, domain parameters $t, 1^q$, factor parameter δ, rounding prime p, and secret vector $z \in \{0, \ldots, t-1\}^n$, such that $t \leq q$.

Instructions for genO$_{5,0}$:

1) Randomly choose M from $\mathbb{Z}_q^{m,n}$
2) Compute vector $u = M \cdot z$
3) Compute rounded vector $v = \lfloor u \rceil_p$
4) Return: (M, v)

Input to evalO$_{5,0}$: dimension parameters $1^n, 1^m$, domain parameters $t, 1^q$, factor parameter δ, rounding parameter p, input vector $x \in \{0, \ldots, t-1\}^n$, such that $t \leq q$, and the output from $genO_3$, containing $M \in \mathbb{Z}_q^{m,n}$, and $v \in \mathbb{Z}_q^m$.

Instructions for evalO$_{5,0}$:

1) Compute vector $u' = M \cdot x$
2) Compute rounded vector $v' = \lfloor u' \rceil_p$
3) If $v' = v$ then return 1 else return: 0.

Theoretical results. Assuming the hardness of the LWR problem, (genO$_{5,0}$,evalO$_{5,0}$) is an obfuscator for the family of point functions, according to the real-vs-random obfuscation indistinguishability definition of [2]. This is obtained by combining the following: (1) the proof in [2] that an obfuscator based on deterministic encryption satisfies the real-vs-random indistinguishability obfuscation notion; (2) the result in [6] saying that a deterministic encryption scheme can be obtained assuming the hardness of the LWR problem.

Primitive setting. For primality testing, we used a procedure from the Palisade lattice crypto library. The 64-bit Mersenne Twister was used for pseudo-random generation of the matrix M, the seed being generated by a call to /dev/random.

B. Description of a Second Variant

Informal Description. This scheme is a variant of the previous scheme, trying to first minimize stored data, and then computation time. The main difference is in the timing of the generation of the matrix M returned by the obfuscator. In the previous variant, M was pseudo-randomly chosen, returned by the obfuscator, and taken as input by the evaluator. In this variant, M is pseudo-randomly generated by both obfuscator and evaluator, using the same short random seed, which is returned by the obfuscator to the evaluator. On one hand, this considerably reduces storage as the obfuscator outputs a short seed instead of a huge matrix M. On the other hand, in the security analysis, we are forced to make a quite strong assumption; namely, that the LWR problem does not become significantly simpler when the seed s generating M is known. One final modification to further reduce storage is that the obfuscator stores $H(v)$ instead of v, where H is a collision-resistant hash function, and the evaluator will do an analogue computation before checking for equality.

Formal description: Let H be a collision-resistant hash function. We now give a formal description of (genO$_{5,1}$,evalO$_{5,1}$).

Input to genO$_{5,1}$: dimension parameters $1^n, 1^m$, domain parameters $t, 1^q$, factor parameter δ, rounding parameter p, and secret vector $z \in \{0,1\}^{\{}0,\dots,t-1\}^n$, such that $t \le q$.

Instructions for genO$_{5,1}$:

1) Pseudo-randomly choose M from $\mathbb{Z}_q^{m,n}$ starting from a random seed s
2) Compute vector $u = M \cdot z$
3) Compute rounded vector $v = \lfloor u \rceil_p$
4) Compute tag $w = H(v)$
5) Return: (s, w)

Input to evalO$_{5,1}$: dimension parameters $1^n, 1^m$, domain parameters $t, 1^q$, factor parameter δ, rounding parameter p, input vector $x \in \{0,\dots,t-1\}^n$, such that $t \le q$, and the output from $genO_{5,1}$, containing seed s and $w \in \{0,1\}^\ell$.

Instructions for evalO$_{5,1}$:

1) Pseudo-randomly generate $M' \in \mathbb{Z}_q^{m,n}$ using seed s
2) Compute vector $u' = M' \cdot x$
3) Compute rounded vector $v' = \lfloor u' \rceil_p$
4) Compute tag $w' = H(v')$
5) If $w' = w$ then return 1 else return: 0.

Theoretical results. Assuming the hardness of the LWR problem even when M is pseudo-randomly generated with a publicly available seed, and that H is a collision-intractable hash function, (genO$_{5,1}$,evalO$_{5,1}$) is an obfuscator for the family of point functions, according to the real-vs-random obfuscation indistinguishability definition of [2]. This is obtained by combining the proof of the result for (genO$_{5,1}$,evalO$_{5,1}$) with the modified assumption on the LWR problem and the assumption on H. We caution the reader that the only intuition we have as to why the modified LWR assumption might be true or false is the following: we can prove that if the pseudo-random generator acts like a random oracle, then this assumption is true, based on the original LWR assumption. However, we already discussed for our first obfuscator why random oracle assumptions may not be reliable.

Primitive setting. In addition to the primitives as in the previous variant, we set H as SHA512.

C. Analysis of Both Variants

Parameter setting. The deterministic encryption scheme in [6] sets all parameters we need as a function of the dimension n and a parameter δ. We determine settings for n, δ so to approximately minimize other parameters, as well as performance metrics, while subject to the following two constraints:

1. $n >= \log(q/\sigma) * 33.1$, for $\sigma = 5$, and
2. q/p is an integer $\ge \sqrt{n}$.

Constraint 1 is based on analysis in [18], which provides a lower bound on n, guaranteeing that the strongest known attacks to the related and much more studied Learning with Error (LWE) problem, and also applicable to LWR, are as successful as breaking a 128-bit cryptographic primitive. Constraint 2 is based on a conjecture in [16], saying that, in light of the strongest known attacks, LWR seems a hard problem as long as $q/p \ge \sqrt{n}$ is an integer and p is polynomial in n. (Should this conjecture appear too optimist in the future, an alternative constraint on parameters could be based on analogue conjectures for the LWE problem, combined with more recent research that provides provable reductions between LWE and LWR that are more efficient than that in [16].) The resulting settings are: $n = 1336$, $\delta = 0.521$, $t = 1$, $m = 285707$ (the dimension of the ciphertext), $p = 170396512836$ and $q = 6304670974932$, where $q/p = 37$. Interestingly, these settings facilitate storing the elements of the encryption matrix and ciphertext vector into variables of unsigned long long type, thus avoiding special large integer operation libraries.

Practical Implementation issues: loop ordering and threading. The matrix-vector multiplication involves nested for loops, one looping over the rows and one over the columns, where there are many more rows than columns. The loop order makes little difference to performance for either the obfuscation generator or the evaluator. However, with an eye to adding parallelization to the code, doing the loops in row-column order allows a whole ciphertext element to be calculated in entirety with each iteration of the outer loop. Thus, a thread is solely responsible for calculating a particular element of the ciphertext vector.

In the parallel version of the first variant, different threads divide the work of the outer loop over the rows, each thread writing to their own file. The relative order of the files is not known, and within any file, all that is known is that the matrix row numbers are increasing, but there can be skips. For this reason, when the matrix is written out it needs to be self-describing, with the data for each row preceded by its row id. When the evaluator later reads the matrix data in, it can reconstitute the same matrix that the obfuscation generator had used. In the parallel version of the second variant, both the obfuscation generator and evaluator use the same seed as input to a pseudo-random number generator, to generate the matrix elements. Because each row is handled completely by one thread, then if there is a seed per row used to seed a random number generator on a row-by-row basis, then the evaluator will be able to generate the same matrix as the obfuscation generator in the face of threads. For this reason, the initial seed that the obfuscation generator creates and stores in a file is used to create a family of seeds in a deterministic way for all the rows of the matrix. These row-specific seeds are used to seed the random number generators for each row.

Performance analysis: storage. In the first variant, the obfuscation generator stores the matrix, in one or more files in a known folder. Assuming 8-byte matrix elements, and a 285707 x 1336 matrix, this yields 3.05 GB of data. Further, to accommodate threading, 285707 4-byte row ids are also stored; this is relatively small, though, and the rounded result is still 3.05 GB. In this variant, it was elected to store the complete ciphertext vector. This is 285707 x 8 bytes, or 2.29 MB. In the second variant, the obfuscation generator stores, in one file, only the 4-byte seed and the 64-byte SHA512 hash of the cipher vector, for a total of 68 bytes.

Performance analysis: running time. We used a 2-core x86_64 machine, with 2.5 CPU GHz and 4988.71 BogoMIPS, and an 8-core x86_64 machine, with 2 CPU GHz and 3990.05 BogoMIPS, with varying numbers of threads enabled. For any given setting, 10 runs of a program were made and the results averaged. Average times are reported here for both variants of these point function obfuscators, with the thread count set to 1 and 8, the number of cores.

Table 3: Performance of scheme ($\mathrm{genO}_{5,0}$, $\mathrm{evalO}_{5,0}$).

Time($\mathrm{genO}_{5,0}$)	Time($\mathrm{evalO}_{5,0}$)
37.65 s; 8 cores; 1 thread	4.44 s; 8 cores; 1 thread
47.85 s; 2 cores; 2 threads	6.89 s; 2 cores; 2 threads
46.79 s; 2 cores; 1 thread	2.54 s; 2 cores; 1 thread

Table 4: Performance of scheme ($\mathrm{genO}_{5,1}$, $\mathrm{evalO}_{5,1}$).

Time($\mathrm{genO}_{5,1}$)	Time($\mathrm{evalO}_{5,1}$)
9.42 s; 8 cores; 8 threads	9.54 s; 8 cores; 8 threads
7.71 s; 2 cores; 2 threads	7.59 s; 2 cores; 2 threads
12.19 s; 2 cores; 1 thread	12.13 s; 2 cores; 1 thread

Using a single thread is preferable for the first variant, but using a number of threads equal to the number of cores is optimal for the second variant.

VIII. Conclusion

While early results in the area of provable program obfuscation showed impossibility of a single obfuscation solution for all programs, more recent research has generated several constructions of program obfuscators for point functions. In this paper, we carefully selected representative point function obfuscators from the literature, applied a number of design and implementation optimizations, stated their theoretical guarantees, and reported on their (slightly) optimized implementations. A basic conclusion is that implementations of point function obfuscators, satisfying different obfuscation notions, can be used with practical performance guarantees and without need for specialized computing resources. We also observed that a definition of program obfuscators, based on parameter passing, can be more relevant to implementation than the known theoretical definition, based on maps between circuits. Finally, we showed that shifting computation to a preprocessing stage is a valid optimization paradigm for typical applications of program obfuscators.

Acknowledgment

This work was supported by the Defense Advanced Research Projects Agency (DARPA) via U.S. Army Research Office (ARO), contract number W911NF-15-C-0233. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, ARO or the U.S. Government.

References

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, “On the (im)possibility of obfuscating programs,” in *Proc. of CRYPTO 2001*, 2001, pp. 1–18.

[2] M. Bellare and I. Stepanovs, “Point-function obfuscation: A framework and generic constructions,” in *Proc. of TCC 2016-A2*, 2016, pp. 565–594.

[3] B. Lynn, M. Prabhakaran, and A. Sahai, “Positive results and techniques for obfuscation,” in *Proc. of EUROCRYPT 2004*, 2004, pp. 20–39.

[4] R. Canetti, “Towards realizing random oracles: Hash functions that hide all partial information,” in *Proc. of CRYPTO 97*, 1997, pp. 455–469.

[5] H. Wee, “On obfuscating point functions,” in *Proc. of 37th ACM STOC 2005*, 2005, pp. 523–532.

[6] X. Xie, R. Xue, and R. Zhang, “Deterministic public key encryption and identity-based encryption from lattices in the auxiliary-input setting,” in *Proc. of SCN 2012*, 2012, pp. 1–18.

[7] A. Boldyreva, S. Fehr, and A. O’Neill, “On notions of security for deterministic encryption, and efficient constructions without random oracles,” in *Proc. of CRYPTO 2008*, 2008, pp. 335–359.

[8] D. M. Freeman, O. Goldreich, E. Kiltz, A. Rosen, and G. Segev, “More constructions of lossy and correlation-secure trapdoor functions,” in *Proc. of PKC 2010*, 2010, pp. 279–295.

[9] R. Canetti, O. Goldreich, and S. Halevi, “The random oracle methodology, revisited (preliminary version),” in *Proc. of 30th ACM STOC*, 1998, pp. 209–218.

[10] D. M. Gordon, “A survey of fast exponentiation methods,” *J. Algorithms*, vol. 27, no. 1, pp. 129–146, 1998.

[11] M. Blum and S. Micali, “How to generate cryptographically strong sequences of pseudo random bits,” in *Proc. of 23rd IEEE FOCS 1982*, 1982, pp. 112–117.

[12] I. Damgård and M. Jurik, “A generalisation, a simplification and some applications of paillier’s probabilistic public-key system,” in *Proc. of PKC 2001*, 2001, pp. 119–136.

[13] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *Proc. of EUROCRYPT ’99*, 1999, pp. 223–238.

[14] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby, “A pseudorandom generator from any one-way function,” *SIAM J. Comput.*, vol. 28, no. 4, pp. 1364–1396, 1999.

[15] L. Carter and M. N. Wegman, “Universal classes of hash functions,” *J. Comput. Syst. Sci.*, vol. 18, no. 2, pp. 143–154, 1979. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(79)90044-8

[16] A. Banerjee, C. Peikert, and A. Rosen, “Pseudorandom functions and lattices,” in *Proc. of EUROCRYPT 2012*, 2012, pp. 719–737.

[17] D. Micciancio and C. Peikert, “Trapdoors for lattices: Simpler, tighter, faster, smaller,” in *Proc. of EUROCRYPT 2012*, 2012, pp. 700–718.

[18] C. Gentry, S. Halevi, and N. P. Smart, “Homomorphic evaluation of the AES circuit,” in *Proc. of CRYPTO 2012 (see also updated version on eprint)*, 2012, pp. 850–867.