Cryptographic Program Obfuscation: Practical Solutions and Application-Driven Models

Giovanni Di Crescenzo

Abstract Program obfuscation is about modifying source or machine code into functionally equivalent code that is hard to understand to a human or some other program. Early obfuscation techniques included heuristic non-cryptographic code transformations, many of which however, have been found to be ineffective against sufficiently motivated adversaries. The recent area of cryptographic program obfuscation targets the design and implementation of program obfuscators that are provably secure under a widely accepted intractability assumption, following the standard of modern cryptography solutions. In this chapter we provide a brief summary of the state of the art in cryptographic program obfuscation, focusing on two main aspects: first, there are many implementations of point function obfuscators, satisfying different obfuscation notions, and many of them can be used with practical performance guarantees; second, multiple application-driven obfuscation models and problems can be generated, where practical attack classes can be addressed by leveraging current implementations of point function obfuscators, as well as potential future practical implementations of special-purpose obfuscators.

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. Approved for Public Release, Distribution Unlimited.

G. D. Crescenzo (✉)
Perspecta Labs, Basking Ridge, NJ, USA
e-mail: gdicrescenzo@perspectalabs.com

M. Conti et al. (eds.), *Versatile Cybersecurity*, Advances in Information Security 72, https://doi.org/10.1007/978-3-319-97643-3_5

1 Introduction

Program obfuscation is about modifying source or machine code into functionally equivalent code that is however hard to understand to a human or some other program. Until about 20 years ago, studies in program obfuscation were motivated mainly by the intention to protect software intellectual property from reverse-engineering attacks. Obfuscation techniques included heuristic code transformations performed by a human or by an obfuscator program, some of them building on transformations similar to those applied during compilation (see, e.g., [15] for a taxonomy of code transformation techniques). Many such techniques, however, have been found to be ineffective against a sufficiently motivated adversary, eventually being capable of developing automated deobfuscation techniques and thus reverse-engineering of the program (see, e.g., [35]).

In the past 20 years or so, the problem of program obfuscation has been attracting a significant amount of research in the modern cryptography literature, as 'cryptographic program obfuscation' might remove the heuristic aspect from obfuscation techniques. Following the success of modern cryptography, where multi-party computation protocols can be designed and proved secure under widely accepted computational intractability assumptions, cryptographic program obfuscation aspires at designing obfuscated programs whose obfuscation enjoys similar provability guarantees. Actually early results in the area conveyed somewhat mixed messages: on the positive side, in [25] it was showed that cryptographic program obfuscation could solve a long-standing open problem in cryptography; while on the negative side, results in [4] implied the very likely impossibility of constructing a single obfuscator for all polynomial-time programs. This still left open the following two main possibilities: (1) constructing a program obfuscator for all polynomial-time programs (i.e., a general-purpose obfuscator) with respect to a less general notion of obfuscation security (and thus, a less general class of attacks); and (2) constructing a program obfuscator for each specific polynomial-time program (a special-purpose obfuscator) with respect to a general notion of obfuscation security.

The line of research (1) has seen much excitement since several other uses of general-purpose obfuscation with respect to this weaker notion of obfuscation were presented (see, e.g., [3] and references therein), potentially solving several long-standing open problems, including some rather surprising ones (e.g., transforming any public-key encryption scheme into a private-key encryption ones). Many candidate general-purpose obfuscator constructions have been proposed based on heuristic constructions of an advanced mathematical objects, called multilinear forms, or approximate versions of them; unfortunately, as of today, many of these candidates have been broken and the future of this research direction has been questioned.

The line of research (2) has actually shown some encouraging progresses, in that recent results show the possibility of constructing obfuscators for restricted families of functions, such as secret verification (aka point) functions, and a few isolated extensions of them, under commonly used, and widely accepted, hardness

assumptions. 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. This line of research has many more chances of being relevant to real-life applications. For starters, commonly used protocols for password-based authentication have been reformulated as instances of point-function obfuscation (the password playing the role of the secret point to be matched). More specifically, the current research literature contains a few theoretical definitions of program obfuscation for point functions (see, e.g., [4, 5]), several constructions of program obfuscators for point functions based on commonly used hardness assumptions, with different performance and security features (see, e.g., [5, 11, 28, 36], as well as several other contributions cited in these cited papers), and implementation efforts showing the practicality of some of these results (see, e.g., [1, 18]). Some of the most relevant results are, in turn, based on other cryptographic primitives (such as deterministic encryption, lossy trapdoor functions, etc.), which have been previously studied in other sequences of papers, even though following the paths of these relationships and understanding the full applicability of these results is a non-trival task for the casual cryptography or security reader. Further specific programs for which special-purpose obfuscators have been proposed include hyperplane membership [13], short-distance matching [19], proxy re-encryption [27], and wildcard-based matching [10]. For none of these latter obfuscators, however, practical implementations have been shown yet.

This chapter can be divided into two conceptual parts, where we show:

1. practical implementations of point function obfuscators, provably secure under widely used intractability assumptions and in theory-oriented models and definitions of cryptographic program obfuscation, and
2. guidelines to generate application-oriented models and definitions of cryptographic program obfuscations, addressing more practical classes of attacks.

In the first part of the chapter (Sects. 2, 3, 4, 5, 6, 7, and 8) we start by considering theory-oriented models and definitions of program obfuscators from the literature, 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 including 4 of the most used security notions for cryptographic program obfuscators, capturing the following theoretical classes of attacks:

1. learning some information on the obfuscated program significantly better than by just evaluating a black box computing the same program;
2. learning the output of a predicate on input the obfuscated program significantly better than by just evaluating a black box computing the same program;
3. distinguishing the output of a predicate on input the obfuscated program from the output of a predicate on input an obfuscation of a random program within a given class;
4. distinguishing the obfuscated program from an obfuscation of a random program within a given class.

We sort out the intricate literature on this sub-area to select some interesting point function obfuscators from [1, 5, 11, 18, 28, 36], 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-theory 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 our implementations of these obfuscators from [1, 18], showing their practical performance, in terms of runtime and storage of the obfuscated program. These implementations apply, wherever possible, a small amount of both design and coding optimizations. Among the former type of optimizations, the computations of certain values are replaced with different and more efficient computations of almost equally distributed values. In one case, a similar distribution is maintained only at the cost of a (much) stronger hardness assumption. Among the second type of optimizations, in group-theory obfuscators, conventional modular exponentiation (often, the most expensive operations in group-theoretic cryptography) is replaced with modular exponentiation via pre-processing, combined with Montgomery multiplication; in lattice-theory obfuscators, probabilistic testing techniques can be used to reduce both storage and runtime.

Overall, our conclusion in this chapter's first part is that implementations of point function obfuscators, provably satisfying different obfuscation notions under widely accepted intractability assumptions, can be used with practical performance (i.e., runtime and storage) guarantees.

In the second part of our chapter (Sect. 10), we present application-driven models for cryptographic program obfuscation identifying research problems in this area as a tuple of points, each point in the tuple being taken from a different 3-dimensional space. We consider a first 3-dimensional space on problem models (with dimensions on program representation models, input models and participant models), a second 3-dimensional space on security requirements (with dimensions on adversary resources, adversary attacks and adversary goals), and a third space on performance requirements (with dimensions on runtime, memory use and storage of the obfuscated program). Definitions of security requirements are driven by (a) adversary goals and security notions based on distinguishing and computing over obfuscated programs; (b) adversary resources such as chosen program inputs and associated outputs, inspection of the program's code, and eavesdropping program inputs and associated outputs; and, most importantly, (c) practical attack classes such as:

1. the adversary making remote calls to the obfuscated program;
2. the adversary stealing or being leaked the obfuscated program and being able to run it in a different computing environment;
3. the adversary intruding in the same computing environment where the obfuscated program resides, observing while it is being run in that environment as well as being able to inspect and run the program.

Variants of these attacks are discussed in various models, where the adversary may target general or secret-based programs, taking low-entropy or high-entropy inputs, in a 2-party or 3-party model. The resulting application-driven research problems

enhance the applicability of program obfuscation solutions. As an example for that, we show an obfuscation in the 3-party model for simple function families with low-entropy secrets that does protect the secret (instead, any obfuscation in the 2-party model would not adequately protect the secret against learning or black-box attacks).

Overall, our conclusion in this chapter's second part is that current implementations of point function obfuscators (as well as potential future implementations of special-purpose obfuscators) may soon be leveraged to address practical attack classes with practical performance guarantees.

2 Theory-Oriented Modeling of Cryptographic Program Obfuscation

This section introduces and refines a number of definitions and facts related to the literature's theory-oriented modeling of cryptographic program obfuscation. First, it starts with some basic notations and definitions (in Sect. 2.1); then, it provides functionality, efficiency and security requirements of program obfuscators for point functions (in Sect. 2.2); finally, it quickly recalls known constructions of program obfuscators for point functions (in Sect. 2.3).

2.1 Basic Notations and Definitions

Let $a|b$ denote the concatenation of a and b, and let symbol Z_q denotes the set of integers $\{0, \ldots, q-1\}$.

If S is a set, an element of S^n is an n-component vector with components in S, and an element of $S^{m,n}$ is an m-row, n-column matrix with entries in S.

The expression $y \leftarrow T$ denotes the probabilistic process of uniformly and independently choosing y from set T. The expression $y \leftarrow A(x_1, x_2, \ldots)$ denotes the (possibly probabilistic) process of running algorithm A on input $x_1, x_2, \ldots$ and any necessary random coins, and obtaining y as output. A probability distribution D is also written as $D = \{p_1; p_2; \ldots; p_n : v\}$ to denote the distribution of v after the ordered execution of probabilistic processes $p_1, \ldots, p_n$.

2.2 Modeling Cryptographic Program Obfuscation

The original definition from [4] of cryptographic program obfuscators contained 3 main requirements that can be briefly stated as follows: (program functionality) the obfuscated program behaves like the original program; (polynomial slowdown)

the obfuscated program is only polynomially slower than the original program; (virtual black-box obfuscation) the obfuscated program does not leak more to an adversary than access to a black box computing the original program. After recalling a formal version of this definition, the rest of this subsection gives a refined definition of cryptographic program obfuscators, obtained by syntax changes to the original definition and by allowing for some very small error probability of incorrect program output, even when the program input is adversarially chosen after seeing the obfuscated program. The resulting definition is simpler to deal with, from both theory and implementation purposes, and is semantically equivalent for a large class of function families, including point functions. Finally, various security notions are reviewed, including and beyond the original virtual black-box obfuscation (here renamed 'adversary output black-box simulation').

The original definition. We say that the family of functions F admits an obfuscator Obf if Obf is an efficient (possibly probabilistic) 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 refined 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. We then generalize this definition to allow for some small error probability of incorrect program output, even when the program input is adversarially chosen after seeing the obfuscated program. Specifically, we view an obfuscator as a pair of efficient algorithms: an *obfuscation generator oGen* and an *obfuscation evaluator oEval*, with the following syntax. On input function parameters $fpar$, including a description of function $f \in F$, $oGen$ returns generator output $gpar$. On input a description of function $f \in F$, generator output $gpar$, and evaluator input x, $oEval$ returns evaluator output y. The pair of algorithms $(oGen, oEval)$ satisfies the following two properties:

1. (Almost exact functionality): For any f in F, with function parameters $fpar$, and any algorithm A, the equality $y = f(x)$ holds with probability $1 - \delta$, for some very small value δ, where y is generated by the following probabilistic steps:

   1. $gpar \leftarrow oGen(fpar)$,
   2. $x \leftarrow A(gpar)$
   3. $y \leftarrow oEval(gpar, x)$.

2. (Polynomial slowdown): There exists a polynomial p such that for all f in F, the running time of $oEval$ is $\leq p(|f|)$, where $|f|$ denotes the size of the (smallest) boolean circuit computing f.

Security notions. Obfuscators (in both the original and refined definition) can satisfy any one of the following different obfuscation security notions (which have to be valid for all functions input to the obfuscation generator chosen according to their specified distribution, for all efficient adversary algorithms, and except possibly with very small probability):

1. *adversary view black-box simulation* [4]: The adversary can read, and thus execute, the evaluator program $oEval(gpar, \cdot)$. Informally speaking, this notion says that no efficient adversary with these capabilities learns any more information than what it can learn by evaluating a black box program that computes function f. A bit more formally, for any efficient adversary with these capabilities, there exists an efficient algorithm, called the simulator, with black-box access to function f, that produces an output indistinguishable from the evaluator program $oEval(gpar, \cdot)$.
2. *adversary output black-box simulation* [4]: The adversary can read, and thus execute, the evaluator program $oEval(gpar, \cdot)$ and is constrained to return a bit at the end of its computation. Informally speaking, this notion says that for any efficient adversary with these capabilities, the adversary's output bit (indicating, for instance, whether the obfuscated program satisfies a certain property or not) could have been produced after evaluating a black box program that computes function f. A bit more formally, the adversary's output bit can be guessed by an efficient algorithm, called the simulator, with black-box access to function f.
3. *real-vs-random indistinguishability* [5]: The adversary can read, and thus execute, an evaluator program $oEval(rr, \cdot)$ which is randomly chosen between the evaluator program obtained after an obfuscation of the program computing function f and the evaluator program obtained after an obfuscation of the program computing a function randomly chosen from F. The adversary is returning a bit at the end of its computation. Informally speaking, this notion says that at the end of its computation, the adversary cannot distinguish the two cases: an obfuscation of the program computing function f from an obfuscation of the program computing a random function from family F.
4. *strong indistinguishability* [5]: As in the real-vs-random indistinguishability, the adversary can read, and thus execute, an evaluator program $oEval(rr, \cdot)$ which is randomly chosen between the evaluator program obtained after an obfuscation of the program computing function f and the evaluator program obtained after an obfuscation of the program computing a function randomly chosen from F. Informally speaking, this notion says that at the end of its computation, no efficient distinguisher algorithm can distinguish the adversary's output in the two cases: an obfuscation of the program computing function f from an obfuscation of the program computing a random function from family F.

All these notions intuitively capture important properties that a program obfuscator should have, and for any two of these notions, their equivalence is either false or unknown. It is not hard to see that an obfuscator satisfying notion 1 also satisfies notions 2, 3, and 4. Moreover, in [5], it was proved that, for the family of point functions, an obfuscator satisfying notion 4 also satisfies notion 3, and that the converse may not hold.

2.3 Cryptographic Point Function Obfuscation

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* $s \in Par$, the point function f_{par} returns 1 if $x = s$ and 0 otherwise.

In an obfuscator for the family of point functions, the following holds: the obfuscation generator algorithm $oGen$ takes as input the secret value s; the almost exact functionality property implies that, except with very small probability, the evaluator's output is equal to 1 if $x = s$ and 0 otherwise; and each of the security notions implies a different type of obfuscation of secret value s.

We now summarize a sample of known constructions of point function obfuscators. A first obfuscator, satisfying adversary view black-box simulation, was given in [28], under the random oracle assumption. Previous results, although formulated as different cryptographic primitives, might be restated as point function obfuscators satisfying strong indistinguishability under the Decisional Diffie Hellman assumption [11] or under the existence of claw-free permutations [12]. The obfuscator in [36] satisfies (a weakened version of) adversary output black-box simulation under the existence of a strong type of one-way permutations. Finally, more obfuscators were given in [5], 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 hard problems on lattices [37] or lossy trapdoor functions [7], and the latter have been built using any one of many group-theoretic assumptions (see, e.g., [21]).

In Sects. 3, 4, 5, 6, 7, and 8, we review somewhat improved designs of these obfuscators from [1, 18] and in Sect. 9 we compare their security and performance properties.

3 A Point Function Obfuscator from Cryptographic Hashing

The first obfuscator (from [18, 28]), denoted as (oGen_1, oEval_1), for the family of point functions, is based on collision-resistant hashing, modeled in the security analysis as random oracles.

Informal description: This well-known 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 more formal description follows.

Formal description: 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). Scheme (oGen_1, oEval_1) goes as follows.

Input to oGen_1: security parameters 1^n, 1^{ℓ_0}, length parameter 1^{ℓ}, secret value $z \in \{0,1\}^{\ell}$,

Instructions for oGen_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 oEval_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 oEval_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, (oGen_1, oEval_1) is an obfuscator of the family of point functions, satisfying the adversary view black-box simulation notion. In [28], 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 $\geq$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.

4 A Point Function Obfuscator Based on Decisional DH

The second point function obfuscator (from [11, 18]) we describe, denoted as $(\text{oGen}_2, \text{oEval}_2)$, is 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 $|q| = n + 1$. The pair $(Z_p^*, \cdot)$, where $Z_p^* = \{1, \ldots, p-1\}$ and $\cdot$ denotes product modulo p, is a group and has a q-order subgroup, denoted as G_q. Let g denote a generator of G_q. Efficient algorithms are known to randomly choose primes p, q of this form, and a generator for G_q. The *Decisional DH problem* over G_q asks to efficiently distinguish, given p, q, g, the following two tuples:

1. $(p, q, g, g^a \bmod p, g^b \bmod p, g^{ab} \bmod p)$, and
2. $(p, q, g, g^a \bmod p, g^b \bmod p, g^c \bmod p)$,

for uniformly and independently chosen elements a, b, c from Z_q. The *Decisional DH assumption* over G_q says that no efficient algorithm can distinguish these two distributions, except with very small probability. The *Discrete Logarithm problem* over G_q asks to efficiently compute, given p, q, g, and an element $h \in G_q$, the exponent $x \in Z_q$ such that $g^x = h \bmod p$. The *Discrete Logarithm assumption* over G_q says that no efficient algorithm can solve the Discrete Logarithm problem, except with very small probability. The Decisional DH assumption implies the Discrete Logarithm assumption. Even if the converse is known not to hold in some other groups, no polynomial-time algorithm is known to solve the Decisional DH problem in subgroup G_q. A survey of the Decisional DH problem can be found in [8].

Modular exponentiation with preprocessing: The pair of algorithms (ModExpPreproc, ModExpCompute) denotes a scheme for faster computation of modular exponentiation, using preprocessing, and 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, such as the textbook square-and-multiply algorithm. A survey of such faster methods was given in [24]. Some of these methods reduce exponentiation to an arbitrary exponent to a sequence of multiplications of simpler and pre-computed exponentiations to specific exponents. In the implementation described here, one of these methods is further optimized by efficient variants of modular multiplications (i.e., performing Montgomery modular multiplications [9]).

Informal description: 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. This basic idea is extended 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. A formal description of ($oGen_2$, $oEval_2$) follows.
Input to $oGen_2$: length parameter 1^n, secret value $z \in \{0, 1\}^n$
Instructions for $oGen_2$:

1. Randomly choose primes p, q such that $p = 2q + 1, |q| = n + 1$
2. Randomly choose generator g of q-order subgroup G_q of Z_p^*
3. Randomly choose $u \in G_q$
4. Compute $(aux_{u,p}) = \text{ModExpPreproc}(u, p)$
5. Consider z as an element of G_q
6. Compute $v = \text{ModExpCompute}(u, p, z, aux_{u,p})$
7. Return: $(aux_{u,p}, (u, v))$.

Input to $oEval_2$: security parameter 1^n, input value $x \in \{0, 1\}^n$ and the output from $oGen$, containing auxiliary information $aux_{u,p}$ for faster computation of exponentiation modulo p in base u, and pair (u, v).
Instructions for $oEval_2$:

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

Theoretical result. Under the Decisional DH assumption, ($oGen_2$, $oEval_2$) is an obfuscator of the family of point functions with (almost) uniformly distributed secret values, according to strong indistinguishability obfuscation notion of [5] (which generalizes the oracle hashing secrecy from [11]). This follows by a generalization of the proof from [11] 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 Z_p, we used procedures from the OpenSSL library. The scheme (ModExpPreproc, ModExpCompute) can be any pair of algorithms from [24]. 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 them). 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.

5 A Point Function Obfuscator Based on Discrete Logarithms

The third obfuscator (from [18, 36]), denoted as ($oGen_3$, $oEval_3$), for the family of point functions, is based on the Discrete Logarithm assumption. First, we briefly recall this assumption, and then describe the obfuscator and its properties.

Discrete Logarithm assumption over Z_p^*: Let p be an $(n+2)$-bit prime, and let g be a generator of the group Z_p^*. The *Discrete Logarithm problem over Z_p^** 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 over Z_p^** 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 MostSigBit(x) returns 0 if $1 \leq x \leq (p-1)/2$ and 1 if $(p-1)/2 < x \leq p-1$. As for the obfuscator from Sect. 4, we use scheme (ModExpPreproc, ModExpCompute) for faster computation of modular exponentiation.

Informal and formal description: The starting idea of this scheme is as in [36], using two main tools: a one-way permutation (i.e., a permutation that can be efficiently computed but is conjectured to be hard to invert when computed on a random input); and a hard-core predicate for this one-way permutation (i.e., a predicate function that returns a single hard-core bit, is efficiently computable from the input to the one-way permutation and is hard to guess given only the output of the one-way permutation). 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. This basic idea is instantiated by setting the one-way permutation as exponentiation modulo a prime p (which is often conjectured to be a one-way permutation over Z_p^*), and by setting the hard-core bit as the most significant bit of the discrete logarithm exponent (which has been proved to be a hard-core bit for exponentiation modulo p, under the same conjecture). Then, all modular exponentiations are computed by carefully distributing the technique of modular exponentiation with preprocessing between the obfuscation generator and evaluator, similarly as done for our obfuscator in Sect. 4.

A formal description of ($oGen_3$, $oEval_3$) follows.

Input to $oGen_3$: length parameter 1^n, secret value $z \in \{0,1\}^n$.

Instructions for $oGen_3$:

1. Randomly choose prime $p \in \{0,1\}^{n+1}$
2. Randomly choose a generator g of Z_p^*
3. Compute $aux_{g,p} = \text{ModExpPreproc}(g, p)$
4. Consider z as an element of Z_p^* and set $w_1 = z$
5. For $i = 1, \ldots, 3n$,

compute $w_{i+1} = \text{ModExpCompute}(g, p, w_i, aux_{g,p})$
compute $v_i = \text{MostSigBit}(w_{i+1})$

6. Set $v = (v_1 | \cdots | v_{3n})$
7. Return: $(aux_{g,p}, v)$.

Input to $oEval_3$: security parameter 1^n, input value $x \in \{0,1\}^\ell$ and the output from *oGen*, containing auxiliary information $aux_{g,p}$ for faster computation of exponentiation modulo p in base g, and $3n$-bit vector v.
Instructions for $oEval_3$:

1. Consider x as an element of 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, (oGen_3, oEval_3) is an obfuscator of the family of point functions, according to (a weak version of) the adversary output black-box simulation notion [4]. This follows by combining the following: (1) the proof in [36] 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 [6].
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 Z_p^*, we used procedures from the OpenSSL library. The scheme (ModExpPreproc, ModExpCompute) can be any scheme from [24].

6 A Point Function Obfuscator from Decisional Residuosity

This section presents an obfuscator from [5, 7, 17, 18, 21, 31], denoted as (oGen_4, oEval_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 $Z_{n^2}^*$ from a random n-th residue in $Z_{n^2}^*$ (i.e., a value $y = x^N \bmod N^2$, for some random $x \in Z_{n^2}^*$). 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 [5, 7], where a point function obfuscator is constructed from any deterministic encryption [5], and the latter is constructed from any pairwise-independent hash function and lossy trapdoor function [7]. Finally, the construction of a lossy trapdoor function from [21] is used, in turn based on the public-key cryptosystem from [17] (a variant of the one in [31]). The resulting obfuscation evaluator only performs two modular exponentiations, and one of them can be computed using preprocessing, similarly as done in Sect. 4.

Formal description: For any x, let $t = minH(x)$ 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 (oGen_4, oEval_4).

Input to $oGen_4$: 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 $oGen_4$:

1. Randomly choose primes p, q such that $|p| = |q| = n/2$
2. Set $N = pq$
3. Randomly choose $r \in Z_N^*$
4. Set $c = (1+N)r^{N^s} \bmod N^{s+1}$
5. Write z as (u_0, u_1), where $u_0 \in Z_{N^s}$ and $u_1 \in Z_N^*$
6. Randomly choose pairwise independent hash function $piH : Z_{n^s} \times Z_n^* \to Z_{n^s} \times Z_n^*$
7. Set $(v_0, v_1) = piH(u_0, u_1)$, where $v_0 \in Z_{N^s}$, $v_1 \in Z_N^*$
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 $oEval_4$: security parameter 1^n, length parameter 1^ℓ, input value $x \in \{0,1\}^\ell$ and $oGen_4$'s output, 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 $c \in Z_{N^{s+1}}$, integer N, integer s, and value $w \in Z_{N^{s+1}}$.

Instructions for $oEval_4$:

1. Write z as (u_0', u_1'), where $u_0 \in Z_{N^s}$ and $u_1 \in Z_N^*$
2. Set $(v_0', v_1') = piH(u_0', u_1')$, where $v_0' \in Z_{N^s}$, $v_1' \in Z_N^*$
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 (oGen_4, oEval_4) is an obfuscator for the family of point functions, according to the real-vs-random obfuscation indistinguishability definition of [5], and where the point has min entropy at least $n + 2\epsilon$. This is obtained by combining

the following: (1) the proof in [5] 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 [21] of a lossy trapdoor function based on Damgaard-Jurik's cryptosystem [17] (a variant of Paillier's cryptosystem [31]). The pairwise-independent hash function is used to apply the Leftover Hash Lemma from [26].

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 $GF(2^{\ell})$ [14], which we implemented using [34].

7 A Point Function Obfuscators Based on the LWR Problem

In this section we describe an obfuscator, denoted as $(\text{oGen}_5, \text{oEval}_5)$, for the family of point functions (with almost uniformly distributed secrets), using an assumption related to the LWR problem. The obfuscator is obtained in [1] by first combining results in [5, 30, 37] and then performing various design optimizations. We first briefly recall the definition of the LWR problem and its related assumptions, 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 $Z_q^{n,m}$ denote the set of $n \times m$-matrix with elements in $\{0, \ldots, q-1\}$, and let $Z_q^n = Z_q^{n,1}$, for any positive integers n, m. Consider the following two distributions:

1. $D_0 = \{A \leftarrow Z_q^{n,m}; s \leftarrow Z_q^n; b = \lfloor A^T s \rceil_p : (A, b)\}$
2. $D_1 = \{A \leftarrow Z_q^{n,m}; b \leftarrow Z_p^m : (A, b)\}$

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. The LWR assumption has been introduced in [2], as a variant of the LWE assumption, previously introduced in [33], and has been used in some cases to potentially improve the design of cryptographic primitives and protocols based on the LWE assumption. In [2] it is also 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. We also consider a modified LWR assumption, also called *public-seed*

LWR assumption, which assumes the hardness of the LWR problem when matrix A is pseudo-randomly generated with publicly known seed. The variant of this assumption based on LWE has been discussed in detail in [22], and similar conclusions can be reached in the LWR case. Specifically, the public-seed LWR assumption does not appear to be significantly stronger than the LWR assumption.
Informal Description. We start with the obfuscator from any deterministic encryption scheme, as described in [5]. Then, we instantiate the deterministic encryption scheme with the one from [37], based on the LWR assumption. Next, we use two design optimizations from [18]: 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 [37], in turn based on lattice key generation approaches from [30]. The latter simplification is possible since the distribution of the public key was proved in [30] to be statistically indistinguishable from uniform. Finally, we use three design optimizations from [1]: (1) both the obfuscation generator and the obfuscation evaluator use a pseudo-random (instead of random) matrix M with published seed as the public key, and (2) the obfuscation generator stores $H(b)$ instead of a target vector b, and the obfuscation evaluator uses $H(b')$ instead of a generated vector b' in its test checking equality between a generated and a target vector, where H denotes a collision-resistant hash function; (3) inspired by probabilistic testing techniques, we expect that it suffices to run the evaluator's equality test only on a randomly chosen subset of the matrix A's rows, of size much smaller than the original number of rows; then, since matrix A is pseudo-randomly generated, one might as well modify the obfuscator so that it only returns a much reduced number of rows. Optimizations (1) and (2) reduce storage, but slightly increase running time, while optimization (3) further reduces both storage and running time.
Formal description: Let H be a collision-resistant hash function. We now give a formal description of (oGen_5, oEval_5).
Input to $oGen_5$*:* dimension parameters 1^n, 1^m, domain parameter 1^q, factor parameter δ, rounding parameter p, statistical security parameter 1^λ, and secret vector $z \in \{0,1\}^n$.
Instructions for $oGen_5$*:*

1. Set $v = (n + \lambda)/\log q$
2. Pseudo-randomly choose M from $Z_q^{v,n}$ starting from a random seed s
3. Compute vector $u = M \cdot z$
4. Compute rounded vector $b = \lfloor u \rceil_p$
5. Compute tag $w = H(b)$
6. Return: (s, w)

Input to $oEval_5$: dimension parameters 1^n, 1^m, domain parameters t, 1^q, factor parameter δ, rounding parameter p, statistical security parameter 1^λ, input vector $x \in \{0,1\}^n$, and the output from $oGen_5$, containing seed s and $w \in \{0,1\}^\ell$.
Instructions for $oEval_5$:

1. Set $v = (n + \lambda)/\log q$
2. Pseudo-randomly generate $M' \in Z_q^{v,n}$ using seed s
3. Compute vector $u' = M' \cdot x$
4. Compute rounded vector $b' = \lfloor u' \rceil_p$
5. Compute tag $w' = H(b')$
6. If $w' = w$ then return 1 else return: 0.

Theoretical result. Under the public-seed LWR assumption, and using results from [5, 37], in [1] it is proved that (oGen_5, oEval_5) is an obfuscator for the family of point functions (with almost uniformly distributed secrets), according to the adversary view black-box simulation definition.
Parameter setting. Parameters for scheme (oGen_5, oEval_5) are set in [1] by slightly improving some constants in those recommended by [37]. Specifically, all parameters are set as a function of the dimension n and a parameter δ, and settings for n, δ are determined so to approximately minimize other parameters, including 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 $\geq \sqrt{n}$.

Constraint 1 is based on analysis in [23], which provides a lower bound on n, guaranteeing that the strongest known attacks to the 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 [2], saying that, in light of the strongest known attacks to LWE, and also applicable to LWR, the LWR problem seems to remain hard as long as $q/p \geq \sqrt{n}$ is an integer and p is polynomial in n. This set of parameters is then generated starting from $n = 1336$. The resulting settings are:

1. $n = 1336$,
2. $\delta = 0.521$,
3. $m = 285707$ (the dimension of the ciphertext),
4. $p = 170396512836$
5. $q = 6304670974932$, where $q/p = 37$, and
6. $v = \lceil (n + \lambda)/\log p \rceil = 40$, where $\lambda = 128$.

An alternative set of parameter settings can be generated starting with the larger value $n = 2048$, in case the above conjecture appears too optimist in the future, using analogue formulae to derive all other parameters from n, δ.

8 A Point Function Obfuscator Based on the LWE Problem

In this section we present an obfuscator from [1], denoted as (oGen_6, oEval_6), for the family of point functions (with almost uniformly distributed secrets), using an assumption related to the LWE problem. We first briefly recall the definition of the LWE problem and its related assumptions, and then present the obfuscator and its properties.

Learning With Error assumption. Let A^T denote the transpose of matrix or vector A, let q be a prime, let $\cdot$ denote matrix-vector product mod q, and let $+$ denote vector sum mod q. Let $G_{\mu,\sigma}$ denote the probability density of the Gaussian distribution with mean μ and standard deviation σ. For any set $S \subseteq Z$, let $dG_{S,\mu,\sigma}$ denote the probability density of the *discrete Gaussian distribution* with mean μ and standard deviation σ, with assigns to any $x \in S$ the probability $G_{\mu,\sigma}(x)/\sum_{z\in S} G_{\mu,\sigma}(z)$. We note that $dG_{Z_q,\mu,\sigma}$ can be efficiently sampled [20].
Now, consider the following two distributions:

1. $D_0 = \{A \leftarrow Z_q^{n,m}; s \leftarrow Z_q^n; e \leftarrow dG_{Z_q,0,2\sqrt{n}}; b = A^T \cdot s + e \ : \ (A, b)\}$
2. $D_1 = \{A \leftarrow Z_q^{n,m}; b \leftarrow Z_q^m \ : (A, b)\}$

The *LWE problem* asks to efficiently distinguish, whether a sample (A, b) came from D_0 or D_1. The *LWE assumption* states that the distributions D_0 and D_1 are indistinguishable to any efficient algorithm, except with negligible probability. The LWE assumption has been introduced in [33] and has been used to design various cryptographic primitives and protocols since then. The literature includes both research on attack efforts, and on its relationship to other well studied assumptions on lattices, such as bounded-distance decoding and shortest-vector finding. (See [29, 30, 32] for detailed bibliographies and problem overviews). We also consider a modified LWE assumption, also called *public-seed LWE assumption*, which assumes the hardness of the LWE problem when matrix A is pseudo-randomly generated with publicly known seed. This assumption has been discussed in detail in [22], where it is suggested that it might not be significantly stronger than the LWE assumption.
Informal Description. Although similar to the obfuscator in Sect. 7, the approach used by obfuscator (oGen_6, oEval_6) is not based on a deterministic encryption scheme and in fact is inherently probabilistic. On input an n-bit secret string z, the generator algorithm uses the LWE assumption to embed the secret into a random matrix A and a vector b computed as $A \cdot z + e$, for some short Gaussian error e. Note that by the LWE assumption, vector b is computationally indistinguishable from a random vector of the same structure. On input an n-bit string x, the evaluator algorithm computes vector b' as $A \cdot x$, and returns 1 if the vector $b' - b$ is short with respect to some norm (e.g., the L^1 norm), and 0 otherwise. As for the obfuscator in Sect. 7: (1) matrix A is pseudo-randomly generated by both generator and evaluator, using the same short random seed, which is returned as output by the generator and then taken as input by the evaluator; and (2) the generator only returns a much reduced number of rows for matrix A. We refer the reader to [1] for a formal description.
Theoretical results. Under the public-seed LWE assumption, in [1] it is proved that (oGen_6, oEval_6) is an obfuscator for the family of point functions (with almost uniformly distributed secrets), according to the adversary view black-box simulation definition.

9 Security and Performance Comparisons

Security comparisons. Table 1 contains the security notions satisfied by the presented obfuscators and the hardness assumptions under which the obfuscators satisfy these security notions.
Performance comparisons. Table 2 contains the generator runtime, evaluator runtime and storage complexity of the presented obfuscators. The implementation of the first 4 obfuscators was performed on a Dell 2950 processor (Intel(R) Xeon(R) 8 cores: CPU E5405 @ 2.00GHz, 16GB RAM), without parallelism. The implementation of the last 2 obfuscators was performed on an 8-core x8664 machine, with 2 CPU GHz and 3990.05 BogoMIPS. In all cases, both the secret and the input length were chosen as $= 2048$.

Remarks The 6 point function obfuscators can be mapped to uncomparable points in a multi-dimensional space based on the following attributes: evaluator runtime, storage, security notion, hardness assumption, as well as quantum-resistant security. While (oGen_1, oEval_1) has the slowest evaluator runtime, it also assumes that the hash function behaves like a random oracle (a very strong assumption that turned out to be false for some older hash functions). Obfuscators (oGen_i, oEval_i), for $i = 2, 3, 4$, are the only ones satisfying strong indistinguishability, adversary output black-box simulation, and real-vs-random indistinguishability, respectively. Obfuscators (oGen_j, oEval_j), for $j = 5, 6$, are the only ones satisfying a security notion under a quantum-resistant hardness assumption.

Table 1 Security notions and hardness assumptions

Obfuscator	Security notion	Hardness assumption
(oGen_1, oEval_1)	Adv view bb simulation	Random Oracle
(oGen_2, oEval_2)	Strong indistinguishability	Decisional DH
(oGen_3, oEval_3)	Adv output bb simulation	Discrete Log
(oGen_4, oEval_4)	Real-vs-random indistinguishability	Decisional Residuosity
(oGen_5, oEval_5)	Adv view bb simulation	Learning with Rounding
(oGen_6, oEval_6)	Adv view bb simulation	Learning with Errors

Table 2 Performance of the 6 point function obfuscators (oGen_i, oEval_i), for $i = 1, \ldots, 6$

Obfuscator	Generator runtime	Evaluator runtime	Storage
(oGen_1, oEval_1)	0.0004 s	0.0002 s	1 KB
(oGen_2, oEval_2)	0.0734 s	0.0139 s	1 MB
(oGen_3, oEval_3)	76.46 s	12.09 s	0.22 GB
(oGen_4, oEval_4)	0.1317 s	0.1005 s	2.4 MB
(oGen_5, oEval_5)	0.0178 s	0.144 s	$<$100 B
(oGen_6, oEval_6)	0.5580 s	0.3271 s	$<$250 B

10 Application-Driven Modeling of Cryptographic Program Obfuscation

Our treatment of application-driven modeling for cryptographic program obfuscation identifies research problems in this area as a tuple of points, each point in the tuple being taken from a different 3-dimensional spaces. In particular, we focus our dicussion on two 3-dimensional spaces that contain the seemingly most interesting problem variables. In a first 3-dimensional space, based on problem models, we consider dimensions on program representation model, input entropy model and participant model. In a second 3-dimensional space, based on security requirements, we consider dimensions on adversary resources, attacks and goals. Here, identified practical attack classes include remote calls, program theft, and system intrusion. One could consider yet another 3-dimensional space, based on performance requirements, with most interesting dimensions being running time, storage complexity, and memory usage of the evaluator program.

Most tuples generated using points in these 3-dimensional spaces have not been investigated in the literature or even posed as open problems. In the rest of this section, we describe all the mentioned dimensions and 3-dimensional spaces, and discuss which problems in these spaces have been studied in the literature or are currently open problems.

Program Representation. The formalism used to represent programs can be important to determine what features of a program need to be obfuscated or not. Certain program parameters, such as input length, might often be leaked to an adversary without compromising the secrecy desired in the application at hand (as is most typically the case with encryption). Most interestingly, programs can be parameterized by additional secret values or have sensitive logic or both.

In some applications, it might be of greater interest to obfuscate these secret values while it might be not a problem to reveal the program's logic. For instance, consider the family of point functions, defined as $pF = \{f_s \mid s \in \{0,1\}^n\}$, where s is a secret string, and $f_s : \{0,1\}^n \to \{0,1\}^n$ maps an input $x \in \{0,1\}^n$ to 1 if $x = s$ or to 0 otherwise. For such family, it is of interest to obfuscate secret s while allowing the capability of evaluating f_s, and it may be not important or of less interest to hide the logic (i.e., conditional, equality, etc.) used by function f_s. Applications captured by this program representation include password/passphrase verification, password managers, and, more generally, secret-based entity authentication.

In other applications, it might be of greater interest to obfuscate the sensitive logic than parameter values. For instance, consider the family of all polynomial-time functions over n-bit inputs, defined as $aF = \{f \mid Dom(f) = \{0,1\}^n\}$, where f can be any polynomial-time computable function. For such family, it might be of interest to obfuscate the function's logic (i.e., the structure and gates of the circuit computing f), while allowing the capability of evaluating f, and it may be not important or of less interest to hide parameters (i.e., the input length or any auxiliary input values) used by function f. Several applications related to protection of program logic and any related intellectual property are captured by this program representation.

Input Model. The amount of entropy in the program's input can be important to determine if the program can be securely obfuscated or not against adversaries with certain resources or interaction models. In particular, consider a program with a *low-entropy input* and an adversary capable of unlimited evaluations of this program on inputs of its choice. By evaluating the program on all candidate inputs, the adversary can efficiently determine the entire program's input/output behavior, regardless of whether the program is obfuscated or not. If the program's input has low entropy, this adversary can efficiently learn the program and thus bypass any obfuscation. Note that in the literature the definition of low-entropy input is often left unspecified or limited to an asymptotic statement (i.e., an input's entropy is low if it is at most logarithmic in the security parameter). On the other hand, the definition of *high-entropy input* is usually identified with the value of the security parameter (i.e., the entropy amount for which exhaustive search attacks are actually impractical). While it is true that there are many applications, especially when it comes to cryptography programs, where inputs have high entropy, it is also true that in many non-cryptographic applications, inputs might have low entropy. In the latter case, to allow any obfuscation approach to maintain desired security properties, one needs to resort to a weaker model for the adversary's resources and/or interaction with the program. (See, for instance, participant, adversary resource and adversary attack models below.)

Participant Model. As the most basic participant model for cryptographic program obfuscation, one can consider 2 logical entities:

1. a *program deployer*, in charge of generating the obfuscated program, and
2. a *program evaluator*, being allowed to evaluate the program (obfuscated by the program deployer).

Figure 1 depicts the interaction between the two parties, as considered in most literature papers in the area. In some applications, however, the obfuscated program

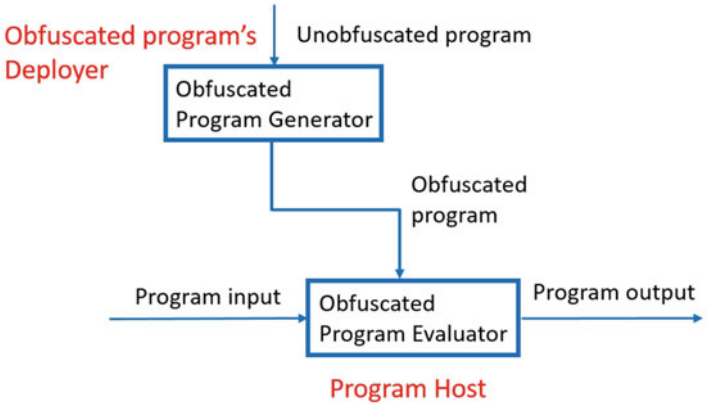

Fig. 1 2-party participant model for cryptographic program obfuscation

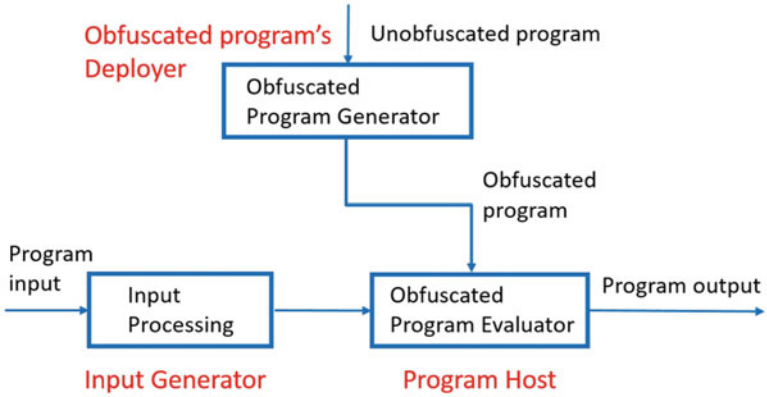

Fig. 2 3-party participant model for cryptographic program obfuscation

is hosted with a server and the input to the program is generated from an additional input source. Thus, one can extend the above 2-participant model into a model with 3 logical entities, defined as follows:

1. a *program deployer*, in charge of generating the obfuscated program,
2. an *input generator*, in charge of generating inputs to the obfuscated program, and
3. a *program host*, being allowed to store the obfuscated program. (Note that an adversary corrupting or intruding into the program host is expected to be also capable of evaluating the program.)

Figure 2 depicts the interaction between the three parties as a natural extension of the interaction between the two parties shown in Fig. 1.

Adversary resources. We identify three main types of program resources that an adversary may use during its attacks:

1. program inputs, chosen by the adversary, and corresponding outputs,
2. program inputs, chosen by a honest user, and corresponding outputs; and
3. a description of the (obfuscated) program's code.

Without knowing further details on how resources of type (2) are generated and the relative state of knowledge of honest users and adversaries with respect to program secrets, it is unclear whether these are less or more valuable (to the adversary) than resources of type (1). For instance, if a honest users generates inputs according to a distribution that somehow depends on program secrets, these inputs and their corresponding outputs might have not been obtained by an adversary with no knowledge of the program secrets. We also note that in practical attacks resources of different types might more or less naturally compose. For instance, access to a resource of type (3), a description of the obfuscated program's code, would directly allow an adversary resources of type (1), as the adversary can use this description to run the program on inputs of its choice and thus see the corresponding outputs.

Table 3 Security notions and hardness assumptions

Adversary attack classes	Adversary resources classes
Remote call	Program inputs, chosen by adversary, and corresponding outputs
Code theft	Description of (obfuscated) program's code
	Program inputs, chosen by adversary, and corresponding outputs
System intrusion	Description of (obfuscated) program's code
	Program inputs, chosen by adversary, and corresponding outputs
	Program inputs, chosen by honest program users, and corresponding outputs, all eavesdropped by adversary

Adversary attacks. We identify three main types of attack that an adversary may run, in order of increasing strength:

1. *remote call* to program functionality, according to which the adversary can remotely execute the program on chosen inputs and receive corresponding outputs;
2. *program theft*, where the adversary can inspect the program's code, run the program with chosen inputs and receive corresponding outputs; and
3. *system intrusion*, according to which the adversary can inspect the program's code, and eavesdrop program executions with inputs chosen by honest users and their corresponding outputs.

There is a natural mapping between these three types of adversary attacks and the three types of program resources available to the adversary, as shown in Table 3.
Adversary goal. Similarly as for other cryptographic primitives, one can define various goals for an adversary attacking an obfuscated program. To protect against such goals, researchers have identified various security notions in the literature. Goals and identified notions in the literature include the following:

1. *distinguishing* a random obfuscation of the given program from information computable in polynomial time given access to *a virtual black-box* computing the same function (with associated security notions identified in [4, 11, 25]);
2. *distinguishing* a random obfuscation of the given program from a random obfuscation of a *randomly chosen program within the defined class* (with associated security notions identified in [5, 25]);
3. *distinguishing* a random obfuscation of *any two programs computing the same given function* (with associated security notions identified in [4]);
4. *computing*, on input a random obfuscation of a program, *an unobfuscated version of the same program* (with associated security notions identified in [16]).

Models and security requirements. Models in cryptographic program obfuscation can be identified as points in a 3-dimensional space (pictorially depicted in Fig. 3), consisting of the previously discussed 3 dimensions: program representationmodel (secret-based programs, general programs, etc.), input entropy model (low-entropy, high-entropy, etc.) and participant model (2-party, 3-party, etc.).

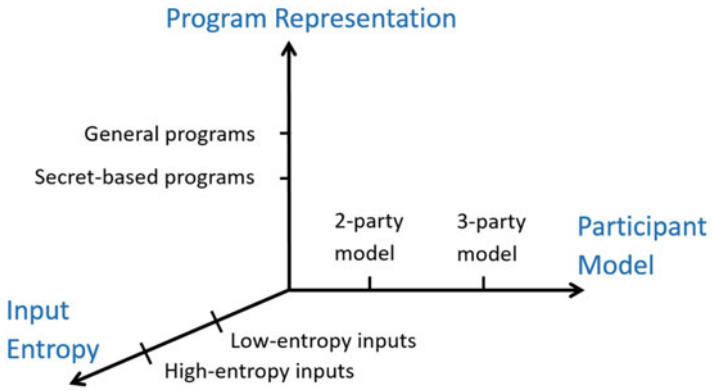

Fig. 3 Problem space for cryptographic program obfuscation

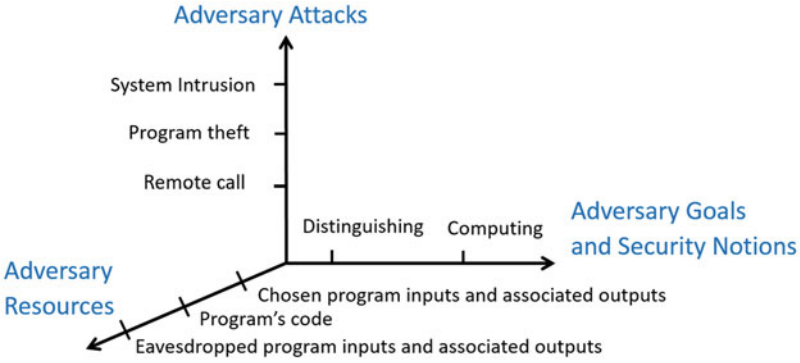

Fig. 4 Security requirement space for cryptographic program obfuscation

Security requirements in cryptographic program obfuscation can be identified as points in a 3-dimensional space (pictorially depicted in Fig. 4), consisting of the previously discussed 3 dimensions: adversary resources (chosen program inputs and associated outputs, program's code, eavesdropped program inputs and associated outputs, etc.), adversary attacks (remote access, program theft, system intrusion, etc.) and adversary goals (on distinguishing the obfuscated program from virtual black boxes, obfuscations of random programs in the class, or other obfuscations for the same program, or computing unobfuscated versions of the same program, etc.).

Performance requirements in cryptographic program obfuscation can be identified as points in a 3-dimensional space, consisting of the previously mentioned 3 dimensions: running time, storage and memory use of the evaluator program.

State of the art and open problems. By taking a model from the 3-dimensional space in Fig. 3, a security requirement from the 3-dimensional space in Fig. 4, and a desired level of performance with respect to the above mentioned performance

requirements, one can generate a meaningful research problem for cryptographic program obfuscation. Most of these research problems have not yet been considered in the literature. More specifically, with respect to models, the literature has focused so far on the obfuscation of general programs, as well as various secret-based programs, including point functions, with high-entropy inputs in the 2-party model. With respect to attack classes, the literature has only considered program theft, for which several different security notions have been defined including those for point functions discussed in Sect. 2.2. With respect to performance requirements, the literature has mainly considered running time of the evaluator program, and practical runtime has been achieved by point function obfuscators, as discussed in Sect. 9.

Application-driven Solutions: a simple example. As a simple example of the increased applicability of the models introduced in this section, we discuss how to design a cryptographic program obfuscator of a program that checks equality between an input bit and a secret bit. More formally, define the *family of bit equality functions* as $beF = \{f_b | b \in \{0,1\}\}$, where b is a secret bit, and $f_b : \{0,1\} \rightarrow \{0,1\}$ maps an input $x \in \{0,1\}$ to 1 if $x = b$ or to 0 otherwise.

Note that for any obfuscation of such program in the most common 2-party model, the secret bit b is easily learnable from the obfuscated program by an adversary that can run the obfuscated program on inputs of its choice. (The adversary runs the obfuscated program on a bit x, obtains an output bit y, and returns x if $y = 1$ and $1 - x$ if $y = 0$, which is a correct guess for secret bit b).

On the other hand, in the 3-party model, one can construct an obfuscator for beF, starting from any block cipher BC, as follows. Let k denote a random key shared by the obfuscated program generator and the input generator. On input secret bit b, the obfuscated program generator computes a nonce $r_0|R = BC(k, 0)$, for some bit r_0, and returns $c = BC(k, b|r)$. On input a bit x, the input generator computes the same nonce $r_0|R = BC(k, 0)$, with the same bit r_0, and returns $d = BC(k, x|r)$. On input c, d, the obfuscated program evaluator returns 1 if $c = d$ and 0 otherwise.

It is not hard to see that this program obfuscator satisfies almost exact functionality, polynomial slowdown and adversary view black box simulation (assuming block cipher BC behaves like a pseudo-random permutation). The almost exact functionality follows from $BC(k, \cdot)$ being a deterministic function that returns the same value when evaluated twice on the same input string. The adversary view black box simulation follows from the pseudo-randomness of BC, as k and r are unknown to the adversary (only attacking the program host).

11 Conclusions

Cryptographic program obfuscation is very promising as it might change the heuristic nature of previous code obfuscation techniques into rigorous and provable solutions, along the paradigm of modern cryptography research. Early negative results on the existence of a single obfuscator for all polynomial-time programs have

been recently mitigated by constructions of obfuscators for specific polynomial-time programs. As of today, the literature contains many implementations of point function obfuscators, satisfying different obfuscation notions, many of which can be used with practical performance guarantees. Moreover, the early theory-driven obfuscation models can be enriched with multiple application-driven obfuscation models by which researchers can protect computer programs against practical attack classes by leveraging current implementations of point function obfuscators, as well as upcoming future practical implementations of obfuscators for other specific functions.

References

1. Lisa Bahler, Giovanni Di Crescenzo, Yuriy Polyakov, Kurt Rohloff, and David Bruce Cousins. Practical implementation of lattice-based program obfuscators for point functions. In *2017 International Conference on High Performance Computing & Simulation, HPCS 2017, Genoa, Italy, July 17-21, 2017*, pages 761–768, 2017.
2. Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Proc. of EUROCRYPT 2012*, pages 719–737.
3. Boaz Barak. Hopes, fears, and software obfuscation. *Commun. ACM*, 59(3):88–96, 2016.
4. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proc. of CRYPTO 2001*, pages 1–18.
5. Mihir Bellare and Igors Stepanovs. Point-function obfuscation: A framework and generic constructions. In *Proc. of TCC 2016-A2*, pages 565–594.
6. Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo random bits. In *Proc. of 23rd IEEE FOCS 1982*, pages 112–117, 1982.
7. Alexandra Boldyreva, Serge Fehr, and Adam O'Neill. On notions of security for deterministic encryption, and efficient constructions without random oracles. In *Proc. of CRYPTO 2008*, pages 335–359.
8. Dan Boneh. The decision diffie-hellman problem. In *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21–25, 1998, Proceedings*, pages 48–63, 1998.
9. Joppe W. Bos and Peter L. Montgomery. Montgomery arithmetic from a software perspective. *IACR Cryptology ePrint Archive*, 2017:1057, 2017.
10. Zvika Brakerski, Vinod Vaikuntanathan, Hoeteck Wee, and Daniel Wichs. Obfuscating conjunctions under entropic ring LWE. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14–16, 2016*, pages 147–156, 2016.
11. Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *Proc. of CRYPTO 1997*, pages 455–469.
12. Ran Canetti, Daniele Micciancio, and Omer Reingold. Perfectly one-way probabilistic hash functions (preliminary version). In *Proc. of 13th ACM STOC, 1998*, pages 131–140.
13. Ran Canetti, Guy N. Rothblum, and Mayank Varia. Obfuscation of hyperplane membership. In *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9–11, 2010. Proceedings*, pages 72–89, 2010.
14. Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.
15. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. In *Technical Report 148, Department of Computer Science, University of Auckland*, 1997.

16. Giovanni Di Crescenzo, Jeyavijayan Rajendran, Ramesh Karri, and Nasir D. Memon. Boolean circuit camouflage: Cryptographic models, limitations, provable results and a random oracle realization. In *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security, ASHES@CCS 2017, Dallas, TX, USA, November 3, 2017*, pages 7–16, 2017.
17. Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Proc. of PKC 2001*, pages 119–136, 2001.
18. Giovanni DiCrescenzo, Lisa Bahler, Brian A. Coan, Yuriy Polyakov, Kurt Rohloff, and David Bruce Cousins. Practical implementations of program obfuscators for point functions. In *Proc. of HPCS 2016*, pages 460–467.
19. Yevgeniy Dodis and Adam D. Smith. Correcting errors without leaking partial information. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22–24, 2005*, pages 654–663, 2005.
20. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Proc. of CRYPTO 2013*, pages 40–56.
21. David Mandell Freeman, Oded Goldreich, Eike Kiltz, Alon Rosen, and Gil Segev. More constructions of lossy and correlation-secure trapdoor functions. In *Proc. of PKC 2010*, pages 279–295.
22. Steven D. Galbraith. Space-efficient variants of cryptosystems based on learning with errors, 2013.
23. Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *Proc. of CRYPTO 2012 (see also updated version on eprint)*, pages 850–867.
24. Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, 1998.
25. Satoshi Hada. Zero-knowledge and code obfuscation. In *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3–7, 2000, Proceedings*, pages 443–457, 2000.
26. Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.
27. Susan Hohenberger, Guy N. Rothblum, Abhi Shelat, and Vinod Vaikuntanathan. Securely obfuscating re-encryption. *J. Cryptology*, 24(4):694–719, 2011.
28. Ben Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In *Proc. of EUROCRYPT 2004*, pages 20–39.
29. Daniele Micciancio. Lattice-based cryptography. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 713–715. 2011.
30. Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *Proc. of EUROCRYPT 2012*, pages 700–718.
31. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. of EUROCRYPT '99*, pages 223–238, 1999.
32. Chris Peikert. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science*, 10(4):283–424, 2016.
33. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proc. of 37th ACM STOC*, pages 84–93, 2005.
34. Gadiel Seroussi. Table of low-weight binary irreducible polynomials. In *Technical Report HPL-98-135*, 1998.
35. Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering, WCRE 2005, Pittsburgh, PA, USA, November 7–11, 2005*, pages 45–54, 2005.
36. Hoeteck Wee. On obfuscating point functions. In *Proc. of 37th ACM STOC 2005*, pages 523–532.
37. Xiang Xie, Rui Xue, and Rui Zhang. Deterministic public key encryption and identity-based encryption from lattices in the auxiliary-input setting. In *Proc. of SCN 2012*, pages 1–18.