Positive Results and Techniques for Obfuscation

Benjamin Lynn[1*], Manoj Prabhakaran[2**], and Amit Sahai[2***]

[1] Stanford University, USA
[2] Princeton University, USA

Abstract. Informally, an *obfuscator* $\mathcal{O}$ is an efficient, probabilistic "compiler" that transforms a program P into a new program $\mathcal{O}(P)$ with the same functionality as P, but such that $\mathcal{O}(P)$ protects any secrets that may be built into and used by P. Program obfuscation, if possible, would have numerous important cryptographic applications, including: (1) "Intellectual property" protection of secret algorithms and keys in software, (2) Solving the long-standing open problem of homomorphic public-key encryption, (3) Controlled delegation of authority and access, (4) Transforming Private-Key Encryption into Public-Key Encryption, and (5) Access Control Systems. Unfortunately however, program obfuscators that work on arbitrary programs *cannot* exist [1]. No positive results for program obfuscation were known prior to this work.

In this paper, we provide the first *positive* results in program obfuscation. We focus on the goal of access control, and give several provable obfuscations for complex access control functionalities, in the random oracle model. Our results are obtained through non-trivial compositions of obfuscations; we note that general composition of obfuscations is impossible, and so developing techniques for composing obfuscations is an important goal. Our work can also be seen as making initial progress toward the goal of obfuscating finite automata or regular expressions, an important general class of machines which are not ruled out by the impossibility results of [1]. We also note that our work provides the *first* formal proof techniques for obfuscation, which we expect to be useful in future work in this area.

1 Introduction

Software Obfuscation is an important cryptographic concept with wide applications. However until recently there was little theoretical investigation of obfuscation, despite the great success theoretical cryptography has had in tackling other challenging notions of security.

Roughly speaking, the goal of (program) obfuscation is to hide the secrets inside a program while preserving its functionality. Ideally, an obfuscated program should be a "virtual black box," in the sense that anything one can compute from it could also be computed from the input-output behavior of the program. To be clear (but still informal), an *obfuscator* $\mathcal{O}$ is an efficient, probabilistic "compiler" that transforms a program P into a new program $\mathcal{O}(P)$ such that:

* Email: `blynn@theory.stanford.edu`.
** Email: `mp@cs.princeton.edu`.
*** Email: `sahai@cs.princeton.edu`.

– (**Functionality Preservation.**) The input/output behavior of $\mathcal{O}(P)$ is the same as P.
– (**Secrecy.**) "Anything that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P."

This second property seeks to formalize the notion that all aspects of P which are not obvious from its input/output behavior should be hidden by $\mathcal{O}(P)$. By considering the problem of obfuscation restricted to specific classes of interesting programs, one can further specify exactly what needs to be hidden by the obfuscation, and what doesn't need to be[3].

Program obfuscation, if possible, would have numerous important cryptographic applications, including: (1) "Intellectual property" protection of secret algorithms and keys in software, (2) Solving the long-standing open problem of homomorphic public-key encryption, (3) Controlled delegation of authority and access, and (4) Transforming Private-Key Encryption into Public-Key Encryption. (See [1] for more discussion.) We discuss another important application, access control, in more detail below.

Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan, and Yang [1] initiated the formal cryptographic study of obfuscation, and established several important impossibility results (which we discuss further below). There have been many ad-hoc approaches to program obfuscation (see *e.g.* [3]); Many of these have been broken (e.g. [4] broken by [7]), and none of these have proofs of their security properties. Proven results are known only in models where the adversary has only partial access to the obfuscated program or circuit [5, 6].

In this paper, we provide the first *positive* results in program obfuscation. We focus on the goal of access control, and give several provable obfuscations for complex access control functionalities, in the random oracle model. Our results are obtained through non-trivial compositions of obfuscations; we note that general composition of obfuscations is impossible, and so developing techniques for composing obfuscations is an important goal. Our work can also be seen as making initial progress toward the goal of obfuscating finite automata or regular expressions, an important general class of machines which are not ruled out by the impossibility results of [1]. We also note that our work provides the *first* formal proof techniques for obfuscation, which we expect to be useful in future work in this area.

Context for our work. In order to understand the challenge of program obfuscation, we first recall the impossibility results of [1]. Their central construction demonstrates the existence of a particular family $\mathscr{F}$ of programs, for which no obfuscator can exist. More precisely, every function in $\mathscr{F}$ has an associated secret key such that: (1) no efficient algorithm can extract the secret key given the input/output functionality of a random function from $\mathscr{F}$; (2) however, there exists an adversary which can *always* extract the

[3] In general, one can define a class of programs parametrized by the secrets which are meant to be protected by the obfuscation. For instance, for a program P which sorts the input and then signs it using a secret signature key sk, one can define a program class $\mathscr{F} = \{P_{sk} : P \text{ using key } sk\}$. An obfuscator for $\mathscr{F}$ would then only be required to protect the secret key; it would not be required, for example, to protect the exact nature of the sorting algorithm, since this is the same for all programs in $\mathscr{F}$.

secret key given *any* program which implements a function in $\mathscr{F}$. There are several important observations to be made:

- The program family $\mathscr{F}$ consists of programs which have inputs and outputs of bounded length. Under a widely believed complexity assumption (factoring Blum integers is hard), $\mathscr{F}$ can implemented by constant-depth polynomial-size threshold circuits (*i.e.* $\mathscr{F} \subset \mathbf{TC^0}$). Furthermore, $\mathscr{F}$ can be embedded into specific constructions of most cryptographic primitives, thus ruling out obfuscators that work on, say, any signature scheme.
- If the obfuscated program runs in time T, the adversary which extracts the secret key runs in time roughly only $\tilde{O}(T^2)$. Note also that the adversary's probability of success is 1.
- The impossibility result (with all the properties above) extends to the random oracle model.

The above properties highlight the difficulty of obtaining any *general* methods for obfuscation: Because the adversary runs quickly and always succeeds in extracting the secret key (and the impossibility result holds in the random oracle model), there seems little hope to relax our security requirement: General purpose obfuscation under any meaningful relaxed secrecy definition[4] would seem to find a counterexample in $\mathscr{F}$.

This has consequences for the techniques we can hope to develop to build and prove obfuscations. One of the most useful techniques we could hope for is composition. However, note that any single logic gate is trivially obfuscatable; indeed even a depth 1 threshold circuit ($\mathbf{TC^0_1}$) is trivially obfuscatable since it is learnable with oracle queries. Obviously, an arbitrary circuit can be built from a composition of logic gates; and any $\mathbf{TC_0}$ circuit can be built from just a *constant* number of compositions of $\mathbf{TC^0_1}$ circuits. Thus, no general theorem showing how to compose even a constant number of obfuscations is possible (under reasonable complexity assumptions).

Our Results. We now describe our results in more detail. The starting point for our work is the simple observation that a commonly used practice for hiding passwords can be viewed as a provably secure obfuscation of a "point function" under the random oracle model. That is, consider the family of functions $\{f_\alpha\}$ where $f_\alpha(x) = 1$ if $x = \alpha$, and $f_\alpha(x) = 0$ otherwise. If $\mathcal{R}$ is a random oracle[5] (with a large enough range), then the program which stores $\rho = \mathcal{R}(\alpha)$, and on input x outputs 1 iff $\mathcal{R}(x) = \rho$

[4] There is one intriguing, if limited, possibility that we can imagine: There is nothing known to rule out a general purpose obfuscator that takes circuits of size s, and outputs circuits of size, say, $O(sk)$, such that no adversary running in time $\Omega(sk^2)$ could obtain meaningful information. If k were large enough, this could conceivably provide enough of a slowdown to be useful in some cases. No such transformation is known to exist.

[5] The work of [2] on "perfectly one-way hash functions" can be seen as a way to implement the random oracle within this obfuscation in certain models. By considering an extension of such models, it is possible to apply the techniques of [2] to remove the random oracles from all our constructions. However, these models are not satisfactory, because in general [2] cannot deal with partial information being available to the adversary, which is an important part of the obfuscation model we consider. Extending [2] to deal with partial information is an important open problem. Progress there would lead to progress toward removing the random oracle in our constructions. However, since we seek to give the *first* positive results regarding obfuscation,

is an obfuscation of f_α with high probability over $\mathcal{R}$. Starting with this most basic of access control functionalities, we give a number of novel reduction and composition techniques for obfuscation, and use these to build obfuscations of much more complex access control functionalities.

We show how to obfuscate a functionality we call an *Access Automaton*. Consider a large organization (such as a government) that wishes to implement a complex hierarchical access control system for a large collection of private information. In such a system, a single piece of information may need to accessible by persons with a variety of different credentials (*e.g.* the co-chair of one subcommittee and the secretary of an unrelated working group may need access to the same piece of secret information). In our setting, we allow for an *exponential* number of sets of credentials to give access to a common piece of information. We model this framework as an arbitrary directed graph, where each edge is labeled with a password/credential, and each node is attached to a secret. At the start, the structure of the graph is completely unknown to a user, but by supplying passwords/credentials, the user can explore and learn as much of the graph as she has access to, given the set of passwords/credentials she has. We show how to provably obfuscate this functionality in the random oracle model. We also show that our obfuscation can be dynamically updated, such that secrecy is preserved even if the adversary observes the entire history of obfuscated programs.

A potential drawback of the above functionality concerns *weak* passwords. Suppose there is a document which is accessible by giving a sequence of 5 passwords, but the adversary has partial information allowing him to narrow each password to a (different) set of 10^4 possibilities. The adversary could efficiently "break" each password one by one, and access the document, even though the document itself had $\log(10^{20})$ "bits" of security. We show how to address this problem: Suppose we have a public regular expression over hidden strings (*e.g.* the expression "$x_1(x_1|x_4)^*(x_2|x_3)x_3x_4)$", where x_1, x_2, x_3, x_4 are unknown strings). Then we show how to essentially obfuscate this expression in a way that preserves the natural security inherent in the expression. In the example above, the adversary would not gain any partial information even if he knew that x_3 was one of only two possibilities – without knowing x_1 and x_4, he cannot resolve his uncertainty about x_3. The main difference between this case and the Access Automaton is that the overall structure of the regular expression is not hidden by the obfuscation. We also give another obfuscation for public regular expressions over "black boxes" – this does not have the security property above, but can be seen as providing a nontrivial obfuscation of a composition of individually obfuscatable functions. We also show how to go beyond just "equality checking" by giving an obfuscation for *proximity checking* in tree metrics.

We believe that the proof techniques we introduce are as important as the results we obtain. In particular, we give a new notion of reduction between classes of functions which implies that if one is obfuscatable, then so is the other. The significance of this is that this allows obfuscations of complex functions to be built using obfuscations of simpler functions. The latter may be implemented in anyway, possibly in the hardware. From a theoretical perspective, this is important because obfuscations built this way

we do not concern ourselves with removing the random oracle in this work. We stress that it is indeed an important problem to address in the future.

need not be based on the random-oracle model, but can be in a model where the simpler obfuscations are available as primitives. We also make many observations about the possibility of putting together multiple obfuscations. We believe our techniques and observations will be of further use in the nascent field of program obfuscation.

2 Preliminaries

Following Barak et al. [1] we define obfuscation of a family of functions $\mathscr{F}$ as follows.

Definition 1. *A family of functions $\mathscr{F}$ is* obfuscatable *if there exists an algorithm $\mathcal{O}$ which takes a Turing Machine (or circuit) that computes $F \in \mathscr{F}$ and outputs a Turing Machine (circuit, respectively) such that the following conditions hold (the TM or circuit is also denoted by F).*

1. *(Functionality) For all $F \in \mathscr{F}$ and all inputs $x \in \{0,1\}^*$ we have $\mathcal{O}(F)(x) = F(x)$*
2. *(Polynomial Slowdown) There exists a polynomial p such that for all $F \in \mathscr{F}$ we have $|\mathcal{O}(F)| \leq p(|F|)$ and (in the case of Turing Machines) if F takes t time steps on an input $x \in \{0,1\}^*$, $\mathcal{O}(F)$ takes at most $p(t)$ time steps.*
3. *(Virtual Blackbox) For all PPT $\mathcal{A}$, there exists a PPT $\mathcal{S}$ and a negligible function ν such that for all $F \in \mathscr{F}$ we have*

$$|\,\mathbf{Pr}\,[\mathcal{A}(\mathcal{O}(F))) = 1] - \mathbf{Pr}\,[\mathcal{S}^F(1^{|F|}) = 1]| \leq \nu(|M|).$$

Here the probabilities are taken over the randomness of $\mathcal{A}$ and $\mathcal{S}$ (and $\mathcal{O}$ and F if they are randomized).

$\mathcal{O}$ is called an obfuscator *for $\mathscr{F}$, and $\mathcal{O}(F)$ an obfuscation of F. $\mathcal{O}$ is said to be* efficient *if it runs in polynomial time, in which case we say $\mathscr{F}$ is* efficiently obfuscatable.

Now we extend this definition so that random oracles are taken into account.

We consider a parameter k associated with the family $\mathscr{F}_k$ of functions being obfuscated. The size of $F \in \mathscr{F}_k$ is polynomial in k, and the random oracle that can be used in the obfuscation will be a random member of $\mathscr{R}_k$, the set of all functions from $\{0,1\}^*$ to $\{0,1\}^{\ell(k)}$ for some polynomial ℓ. We shall refer to k as the *feasibility parameter*.

Definition 2. (Obfuscation in the Random Oracle Model) *An oracle algorithm $\mathcal{O}$ which takes as input a Turing Machine (or circuit) and produces an oracle Turing Machine (or oracle circuit) is said to be an obfuscator of the family $\mathscr{F} = \cup_k \mathscr{F}_k$ if we have that*

$1'$. *(Approximate Functionality) There exists a negligible function ν such that, for all k, for all $F \in \mathscr{F}_k$ we have $\mathbf{Pr}\,[\exists x \in \{0,1\}^* : \mathcal{O}^{\mathcal{R}}(F)(x) \neq F(x)] \leq \nu(k)$.*[6]

$2'$. *(Polynomial Slowdown) There exists a polynomial p such that for all k, for all $F \in \mathscr{F}_k$ we have $|\mathcal{O}(F)| \leq p(k)$ and (in the case of Turing Machines) if F takes t time steps on an input $x \in \{0,1\}^*$, $\mathcal{O}(F)$ takes at most $p(t)$ time steps.*

[6] A weaker requirement would be that for all $F \in \mathscr{F}_k$ and $x \in \{0,1\}^*$, we have $\mathbf{Pr}\,[\mathcal{O}^{\mathcal{R}}(F)(x) \neq F(x)] \leq \nu(k)$.

$3'$. *(Virtual Blackbox) For all PPT $\mathcal{A}$, there exists a PPT $\mathcal{S}$ and a negligible function ν such that for all k, for all $F \in \mathscr{F}_k$ we have*

$$|\,\mathbf{Pr}\,[\mathcal{A}^{\mathcal{R}}(\mathcal{O}^{\mathcal{R}}(F))) = 1] - \mathbf{Pr}\,[\mathcal{S}^F(1^k) = 1]| \leq \nu(k)$$

Here the probabilities are taken over $\mathcal{R} \in \mathscr{R}_k$ as well as the randomness of $\mathcal{A}$ and $\mathcal{S}$ (and $\mathcal{O}$ if it is randomized).

$\mathcal{O}$ is called an obfuscator *for $\mathscr{F}$, and $\mathcal{O}(F)$ an obfuscation of F. $\mathcal{O}$ is said to be* efficient *if it runs in polynomial time, in which case we say $\mathscr{F}$ is* efficiently obfuscatable.

In the sequel, all our results will apply to the definition presented here (in the random oracle model). For notational convenience we shall often abbreviate $\mathcal{O}^{\mathcal{R}}, \mathcal{A}^{\mathcal{R}}$ etc. to simply $\mathcal{O}, \mathcal{A}$ etc.

3 Reductions and Composition

3.1 Reductions

Definition 3. *A class of Turing Machines (or circuits) $\mathscr{F}$ is said to be* polynomial-time black-box implementable relative to $\mathscr{G}$ *(denoted $\mathscr{F} \ll \mathscr{G}$) if there exist polynomial time TMs (circuits) M and N such that for every $F \in \mathscr{F}$ there is a $G \in \mathscr{G}$, such that M^G computes the same function as F, and N^F computes the same function as G.*

So, if $\mathscr{F} \ll \mathscr{G}$, for every $F \in \mathscr{F}$, $\mathscr{G}$ contains a function G which is "equivalent" to F in some extended sense. Now we give the main tool which lets us reuse results on obfuscatability.

Lemma 1. *If $\mathscr{F} \ll \mathscr{G}$ and $\mathscr{G}$ is obfuscatable (when every $G \in \mathscr{G}$ is given as N^F for an $F \in \mathscr{F}$),*[7] *then so is $\mathscr{F}$. Further if $\mathscr{G}$ is efficiently obfuscatable, then $\mathscr{F}$ is efficiently obfuscatable too.*

Proof: Given $F \in \mathscr{F}$, let $G \in \mathscr{G}$ be such that $M^G \equiv F$ and $G \equiv N^F$. Since $\mathscr{G}$ is obfuscatable, let $\mathcal{O}'$ be an obfuscator for $\mathscr{G}$. We claim that $\mathcal{O}(F) = M^{\mathcal{O}'(G)}$ (i.e., the code of M and the code $\mathcal{O}'(G)$) is an obfuscation of F.

Clearly, conditions $1'$ and $2'$ of Definition 2 are satisfied. To prove condition $3'$, consider any adversary $\mathcal{A}$ which accepts the code $\mathcal{O}(F) = M^{\mathcal{O}'(G)}$. We need to demonstrate a PPT $\mathcal{S}$ as required by condition $3'$. First, we build an adversary $\mathcal{A}'$ which accepts the code $\mathcal{O}'(G)$, adds the code of M to it to get $\mathcal{O}(F)$, passes it on to an internally simulated copy of $\mathcal{A}$, and outputs whatever $\mathcal{A}$ outputs. Now, since $\mathcal{O}'(G)$ is an obfuscation of G, there exists a simulator $\mathcal{S}'$ such that

$$|\,\mathbf{Pr}\,[\mathcal{S}'^G(|\mathcal{O}'(G)|) = 1] - \mathbf{Pr}\,[\mathcal{A}'(\mathcal{O}'(G)) = 1]| \leq \epsilon \tag{1}$$

for some negligible function $\epsilon(|\mathcal{O}'(G)|)$.

[7] If $G \in \mathscr{G}$ is obfuscatable only when represented in some other format, still this Lemma holds, but now the obfuscator for $\mathscr{F}$ takes F as M^G with G specified in that obfuscatable format.

We use $\mathcal{S}'$ to build $\mathcal{S}$, as follows. Note that $\mathcal{S}$ gets oracle access to F and receives $|\mathcal{O}(F)|$ as input. $\mathcal{S}^F$ can implement an oracle equivalent to G as N^F, using its oracle access to F. It runs $\mathcal{S}'$ with oracle access to G implemented in this way, and input $|\mathcal{O}'(G)|$ calculated from $|\mathcal{O}(F)|$ (by subtracting the size of M). $\mathcal{S}$ outputs whatever $\mathcal{S}'$ outputs.

Clearly, by construction,

$$\mathbf{Pr}\left[\mathcal{A}(\mathcal{O}(F)) = 1\right] = \mathbf{Pr}\left[\mathcal{A}'(\mathcal{O}'(G)) = 1\right]$$
$$\mathbf{Pr}\left[\mathcal{S}^F(|\mathcal{O}(F)|) = 1\right] = \mathbf{Pr}\left[\mathcal{S}'^G(|\mathcal{O}'(G)|) = 1\right]$$

and so by Equation (1), $|\,\mathbf{Pr}\left[\mathcal{S}^F(|\mathcal{O}(F)|) = 1\right] - \mathbf{Pr}\left[\mathcal{A}(\mathcal{O}(F)) = 1\right]| \leq \epsilon$. Finally $|\mathcal{O}(F)| \geq |\mathcal{O}'(G)|$, so that ϵ is still negligible when considered a function of $\mathcal{O}(F)$, completing the proof.

Note that in building $\mathcal{O}(F) = M^{\mathcal{O}'(G)}$, the obfuscator $\mathcal{O}$ needs to obtain $\mathcal{O}'(G)$, given F. Since G can be specified as N^F to $\mathcal{O}'$, if $\mathcal{O}'$ is efficient so is $\mathcal{O}$. $\square$

3.2 Extending Lemma 1

We extend Definition 3, and Lemma 1 to allow reductions to probabilistic families of functions. We do this for proving Theorem 3. In fact, somewhat more general extensions are possible. But for the sake of simplicity we restrict ourselves more or less to the minimum extensions we will need. The reader may skip this section, and return to it while reading Section 5. The other results in this paper do not need these extensions.

Definition 4. *Suppose $\widetilde{\mathscr{G}}$ is a family of probabilistic Turing Machines (or circuits), and $\mathscr{F}$ a family of deterministic TMs (circuits). We say $\mathscr{F} \lll \widetilde{\mathscr{G}}$ if there exist probabilistic polynomial time TMs (circuits) M and N such that for every $F \in \mathscr{F}$ there is a $G \in \widetilde{\mathscr{G}}$, such that the distributions of outputs of M^G and F are computationally indistinguishable, and those of N^F and G are computationally indistinguishable.*

Note that unlike Definition 3, the above definition is *not* information theoretic. It involves the notion of computational indistinguishability, and hence inherently all the results which use the following lemma requires the adversary ($\mathcal{A}$ and $\mathcal{S}$) to be PPT machines or circuits. The proof of the lemma closely follows that of Lemma 1. It is given in the extended version [8].

Lemma 2. *Suppose $\mathscr{F} \lll \widetilde{\mathscr{G}}$. Let $\mathscr{G}$ be the family of deterministic TMs (circuits) obtained by fixing in all possible ways the random-tapes of the TMs (circuits) in $\widetilde{\mathscr{G}}$. Then, if $\mathscr{G}$ is obfuscatable, so is $\mathscr{F}$.*

3.3 Composition of Obfuscations

An obfuscated program can be idealized as oracle access to the corresponding function. We ask if obfuscations compose: can we put together different obfuscations and expect them to behave ideally as the corresponding collection of oracles. Note that here we use the term *compose* in the same way as one refers to composition of cryptographic protocols- to ask whether having multiple instances in the system breaks the security or not. It does not necessarily refer to composition of functions in the usual mathematical sense, something which we will address later in this section. We make the following definition to define a simple composition of obfuscations, where there is no interaction between the different instances.

Definition 5. *An* array *of* t *functions* $F_1, \ldots, F_t$ *is defined as follows:*

$$[\![F_1, \ldots, F_t]\!](i, x) = F_i(x) \quad \text{if } i \in \{1, \ldots, t\}; \text{ else } \perp$$

Let $[\![\mathcal{O}(F), \mathcal{O}(G)]\!]$, by abuse of notation stands for the code which consists of the codes $\mathcal{O}(F)$ and $\mathcal{O}(G)$ as modules, and a small driving unit which directs the calls to one of the modules as appropriate.

Definition 6. (Simply Composing Obfuscations) *An obfuscator* $\mathcal{O}$ *for a family* $\mathscr{F}$ *is said to produce* simply t-self-composing obfuscations *if*

$$\mathcal{O}^*([\![F_1, \ldots, F_t]\!]) = [\![\mathcal{O}(F_1), \ldots, \mathcal{O}(F_t)]\!]$$

is an obfuscation of the family $\{[\![F_1, \ldots, F_t]\!] | F_i \in \mathscr{F}\}$.[8]

This can be extended to multiple families of obfuscatable functions to define a set of simply composing obfuscations*.*

In fact, in the random oracle model we have the following claim (which we conjecture to extend to the plain model too):

Claim 1. *There exists a class of functions* $\mathscr{F}$*, and an obfuscator* $\mathcal{O}$ *for* $\mathscr{F}$ *in the random oracle model, such that obfuscations produced by* $\mathcal{O}$ *are* not *simply 2-self-composing.*

Proof: We consider the class of point functions $\mathscr{P}$ (defined later, in Section 4). By Lemma 4, this class is obfuscatable in the random oracle model. Note that when F and G are identical (randomly chosen) functions, oracle access to the function $[\![F, G]\!]$ does not reveal the fact that they are identical, to a PPT machine. On the other hand the obfuscation given in Lemma 4 does reveal this. (Of course, it is easy to modify the obfuscation, in order to avoid this problem.) Thus no simulator can simulate the behaviour of an adversary $\mathcal{A}$ (which has access to these obfuscations) which outputs 1 if $F = G$ and 0 otherwise. □

Conjecture 1. *If there are non-trivial obfuscations in the plain model, Claim 1 holds in the plain model too. Indeed, in that case, we conjecture that there exists an obfuscatable family* $\mathscr{F}$*, such that* $\mathscr{A} = \{[\![F, G]\!] : F, G \in \mathscr{F}\}$ *is* unobfuscatable*.*

The difficulty in attempting to prove this conjecture is that it requires a non-trivial obfuscatable family $\mathscr{F}$, and we have virtually nothing known beyond what is being presented in this work (which is in the random oracle model).

On the other hand, an obfuscatable function composes with any *trivially obfuscatable* function (defined below).

[8] We can have t constant, or polynomial in the feasibility parameter k.

Definition 7. *A family of functions $\mathscr{F}$ is* learnable as polynomial time circuits *if there exists an oracle circuit P such that for all $F \in \mathscr{F}$, P^F outputs a polynomial sized circuit C_F which computes F.*

If $\mathscr{F}$ is learnable it is obfuscatable: the obfuscator $\mathcal{O}$ takes a circuit for F and runs P with oracle access to that circuit; it outputs C_F produced by P as $\mathcal{O}(F)$. This is clearly an obfuscation, because for every adversary $\mathcal{A}$, a simulator S simply runs P with the oracle for F, obtains C_F and runs $\mathcal{A}$ on it.

Definition 8. *A family of learnable functions is called a* family of trivially obfuscatable functions*. The obfuscation obtained via learning the function is called the* trivial obfuscation of the function*.*

Simple as the following lemma is, it is interesting that its intuitive extension from *trivially* obfuscatable family to *any* obfuscatable family is an open problem.

Lemma 3. *Let $\mathscr{F}$ be a trivially obfuscatable family of functions. Then, $\mathscr{G}$ is obfuscatable, if and only if the family of functions $\mathscr{A} = \{[\![F, G]\!] \ : \ F \in \mathscr{F}, G \in \mathscr{G}\}$ is obfuscatable.*

Proof: First, we show that $\mathscr{G} \ll \mathscr{A}$. Then it follows from Lemma 1 that $\mathscr{G}$ is obfuscatable if $\mathscr{A}$ is.

To see that $\mathscr{G} \ll \mathscr{A}$, for each $G \in \mathscr{G}$ we choose $A = [\![F, G]\!] \in \mathscr{A}$, where $F \in \mathscr{F}$ is a fixed function for all G. Then a machine M which internally implements F can implement A with access to only G. On the other hand a machine N which has access to A can clearly implement G.

Now we show that $\mathscr{A}$ is obfuscatable if $\mathscr{G}$ is. Intuitively, an obfuscation of $\mathscr{A}$ does not "hide" the $\mathscr{F}$ component (which is easily learnable). So it is sufficient if we are able to obfuscate the $\mathscr{G}$ part. Formally, we show that for $A = [\![F, G]\!] \in \mathscr{A}$, the following is a valid obfuscation: $\mathcal{O}(A) = [\![\mathcal{O}'(F), \mathcal{O}'(G)]\!]$, where $\mathcal{O}'(F)$ is the trivial obfuscation of F and $\mathcal{O}'(G)$ is the obfuscation of G given by the assumption that $\mathscr{G}$ is obfuscatable. As earlier the notation $[\![\mathcal{O}'(F), \mathcal{O}'(G)]\!]$ refers to the code which has $\mathcal{O}'(F)$ and $\mathcal{O}'(G)$ as internal modules, plus a small control module to activate the appropriate one depending on the input.

To show that $\mathcal{O}(A)$ is a valid obfuscation, for every adversary $\mathcal{A}$ which accepts $\mathcal{O}(A)$, we show a simulator $\mathcal{S}$ such that $|\,\mathbf{Pr}\,[\mathcal{S}^A(|\mathcal{O}(A)|) = 1] - \mathbf{Pr}\,[\mathcal{A}(\mathcal{O}(A)) = 1]|$ is negligible. The structure of the argument is similar to that in the proof of Lemma 1.

From $\mathcal{A}$, we first build an adversary $\mathcal{A}'$ which takes as input $\mathcal{O}'(G)$, uses it to build the code $\mathcal{O}(A)$ =$[\![\mathcal{O}'(F), \mathcal{O}'(G)]\!]$, passes it on to an internally simulated copy of $\mathcal{A}$, and outputs whatever $\mathcal{A}'$ outputs. Using the fact that $\mathcal{O}'(G)$ is an obfuscation of G, there exists a simulator $\mathcal{S}'$ such that

$$|\,\mathbf{Pr}\,[\mathcal{S}'^G(|\mathcal{O}'(G)|) = 1] - \mathbf{Pr}\,[\mathcal{A}'(\mathcal{O}'(G)) = 1]| \leq \epsilon \tag{2}$$

for some negligible function $\epsilon(|\mathcal{O}'(G)|)$.

We use $\mathcal{S}'$ to build a simulator $\mathcal{S}$ as follows. Note that $\mathcal{S}$ gets oracle access to A and receives $|\mathcal{O}(A)|$ as input. Oracle access to A in particular gives oracle access to F.

Since F is trivially obfuscatable, it is possible to obtain the trivial obfuscation $\mathcal{O}'(F)$ just using this oracle access to F. So $\mathcal{S}$ first computes $\mathcal{O}'(F)$. Next, note that given oracle access to A, oracle access to G can also be implemented. So $\mathcal{S}$ runs $\mathcal{S}'$ with oracle access to G implemented in this way, and input $|\mathcal{O}'(G)|$ calculated from $|\mathcal{O}(A)|$ (by subtracting the size of $\mathcal{O}'(F)$). $\mathcal{S}$ outputs whatever $\mathcal{S}'$ outputs.

By construction,

$$\mathbf{Pr}\left[\mathcal{A}(\mathcal{O}(A)) = 1\right] = \mathbf{Pr}\left[\mathcal{A}'(\mathcal{O}'(G)) = 1\right]$$
$$\mathbf{Pr}\left[\mathcal{S}^{A}(|\mathcal{O}(A)|) = 1\right] = \mathbf{Pr}\left[\mathcal{S}'^{G}(|\mathcal{O}'(G)|) = 1\right]$$

and so by Equation (2), $|\,\mathbf{Pr}\left[\mathcal{S}^{F}(|\mathcal{O}(F)|) = 1\right] - \mathbf{Pr}\left[\mathcal{A}(\mathcal{O}(F)) = 1\right]| \leq \epsilon$. Finally to complete the proof, we note that $|\mathcal{O}(A)| \geq |\mathcal{O}'(G)|$ and so ϵ is still negligible when considered a function of $\mathcal{O}(A)$. $\square$

Now we consider the question of more complex composition of obfuscations. We ask if obfuscations of composed functions can be obtained by using obfuscations of the component functions. In particular we look at function compositions (in the usual mathematical sense, of one function invoking another).

Conjecture 2. *Conjecture on Obfuscatability of Function Compositions: Given two classes $\mathscr{F}$ and $\mathscr{G}$ of obfuscatable programs, the family $\mathscr{A} = \{A(x) = F(G(x)) : F \in \mathscr{F}, G \in \mathscr{G}\}$ is obfuscatable.*

Theorem 1. *The Conjecture on Obfuscatability of Function Compositions is false, if factoring Blum integers is hard or the DDH assumption is true.*

Proof Sketch: The Conjecture on Obfuscatability of Function Compositions, if true, could be applied any constant number of times: if $\mathscr{F}$ is obfuscatable, then $\cup_t \{A(x) = F_1(F_2(\cdots(F_t(x))\cdots)) | F_i \in \mathscr{F}\}$ is obfuscatable. However, it is known that if the assumptions of the theorem hold, then there exists a family of functions $\mathscr{A} \subset \mathbf{TC^0}$ that is unobfuscatable. On the other hand it is not hard to see that $\mathscr{F} = \mathbf{TC^0_1}$, the family of depth 1 threshold circuits, is trivially obfuscatable, because they can be easily learned from input/output queries. Noting that $\mathscr{A}$ is obtained by a constant number of compositions of functions from $\mathscr{F}$ completes the contradiction, and the proof. $\square$

4 Point Functions and Extensions

In this section we define a few basic functions which can be obfuscated under the random oracle model. The proofs are easy and we include a couple of them.

Definition 9. (Class of Point Functions) *A point function $P_\alpha : \{0,1\}^k \to \{0,1\}$ is defined by $P_\alpha(x) = 1$ if $x = \alpha$ and 0 otherwise. Define $\mathscr{P}_k = \{P_\alpha : \alpha \in \{0,1\}^k\}$ and $\mathscr{P} = \cup_k \mathscr{P}_k$.*

We observe that the following simple obfuscation heuristic is indeed an obfuscation in the random oracle model (Definition 2).

Lemma 4. *For random oracles $\mathcal{R} : \{0,1\}^* \to \{0,1\}^{2k}$, let $\mathcal{O}^{\mathcal{R}}(P_\alpha)$ be a program which stores $r = \mathcal{R}(\alpha)$, and on input $x \in \{0,1\}^k$, checks if $\mathcal{R}(x) = r$; if so it outputs 1, else 0.*

Then, $\mathcal{O}$ is an obfuscator of $\mathscr{P}$ as defined in Definition 2.

Proof: Polynomial Slowdown is evident (by convention oracle queries are answered in one time step). The Approximate Functionality condition is true since

$$\begin{aligned} &\mathbf{Pr}_{\mathcal{R}}[\exists x \in \{0,1\}^k \backslash \{\alpha\} : \mathcal{R}(x) = \mathcal{R}(\alpha)] \\ \leq &\sum_{x \in \{0,1\}^k \backslash \{\alpha\}} \mathbf{Pr}_{\mathcal{R}}[\mathcal{R}(x) = \mathcal{R}(\alpha)] = (2^k - 1)/2^{2k} \end{aligned}$$

which is negligible in k.

To show the Virtual Black-Box property $(3')$, for any adversary $\mathcal{A}$, define the simulator $\mathcal{S}$ (with oracle access to P_α which does the following. Pick a random string $r \leftarrow \{0,1\}^{2k}$, prepare a purported obfuscation of P_α with this r and hand it to an internally simulated copy of $\mathcal{A}$. Recall that $\mathcal{A}$ can make queries to a random oracle, which in this case will be simulated by $\mathcal{S}$. W.l.o.g we assume $\mathcal{A}$'s queries to the oracle are distinct, since oracle replies can be cached. When $\mathcal{A}$ makes a query q to the random oracle, $\mathcal{S}$ queries the P_α oracle with q. If P_α answers 1, it answers $\mathcal{A}$'s query with r. Else it picks a random string in $\{0,1\}^{2k}$ and sends it to $\mathcal{A}$. Finally $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. It is easy to see that the view of this internally simulated $\mathcal{A}$ is *identical* to that of an $\mathcal{A}$ which receives the obfuscation and access to the random oracle. Thus the Virtual Black-box requirement is satisfied (with $\nu(k) = 0$). □

Though we defined the point function as $P_\alpha : \{0,1\}^k \to \{0,1\}$ with $\alpha \in \{0,1\}^k$, it is easy to see that it can be modified to $P_\alpha : \cup_{i=0}^{k} \{0,1\}^i \to \{0,1\}$ with $\alpha \in \cup_{i=0}^{k} \{0,1\}^i$

4.1 Composable Obfuscations of Point Functions with General Output

Definition 10. (Class of Point Functions with General Output) *A point function with general output $Q_{(\alpha,\beta)} : \{0,1\}^k \to \{0,1\}^{s(k)}$ is defined by $Q_{\alpha,\beta}(x) = \beta$ if $x = \alpha$ and $\perp$ otherwise. Define $\mathscr{Q}_k = \{P_\alpha \ : \ \alpha \in \{0,1\}^k\}$ and $\mathscr{Q} = \cup_k \mathscr{Q}_k$.*

We omit the proof of the following theorem, as it is similar to the proof of Lemma 4.

Theorem 2. *For random oracles $\mathcal{R} : \{0,1\}^* \to \{0,1\}^{2k+s(k)}$, let $\mathcal{O}^{\mathcal{R}}(P_{\alpha,\beta})$ be a program as follows: Let $\mathcal{R}_1(\cdot)$ denote the first $2k$ bits of $\mathcal{R}(\cdot)$, and $\mathcal{R}_2(\cdot)$ denote the remaining bits. Choose ψ at random from $\{0,1\}^k$. Let $a = \mathcal{R}_1(\psi, \alpha)$ and $b = \mathcal{R}_2(\psi, \alpha)$. The program stores ψ, a and $c = \beta \oplus b$. On input $x \in \{0,1\}^k$, it computes $a' = \mathcal{R}_1(\psi, x)$ and $b' = \mathcal{R}_2(\psi, x)$; if $a' = a$ it outputs $b' \oplus c$; else it outputs $\perp$.*

Then, $\mathcal{O}$ is an obfuscator of $\mathscr{P}$ as defined in Definition 2.

We further observe that the above obfuscation self-composes according to Definition 6. As long as there only polynomially many (polynomial in k) obfuscations in the system, the probability that two of the obfuscations will have the same value of ψ is negligible. Conditioned on this (negligible probability) event not happening, a simulator with black-box access to all the (polynomially many) $Q_{\alpha,\beta}$ functions can perfectly simulate the behavior of an adversary with access to the obfuscations. Note that here the obfuscator is a randomized algorithm.

4.2 Multi-Point Functions with General Output

Finally, we define a multi-point function *with general output* as follows.

Definition 11. (Class of Multi-Point Functions with General Output) *A multi-point function* $Q_{(\alpha_1,\beta_1)\ldots,(\alpha_t,\beta_t)} : \{0,1\}^k \to \left(\{0,1\}^{s(k)}\right)^t$ *is defined as follows: On input* x*, output* $b \in \left(\{0,1\}^{s(k)}\right)^t$ *where* $b_i = \beta_i$ *if* $x = \alpha_i$*, and else* $b_i = \perp$*. Define* $\mathscr{Q}_k^t = \{Q_{(\alpha_1,\ldots,\alpha_{t(k)})} : \alpha_i \in \{0,1\}^k\}$ *and* $\mathscr{Q}^t = \cup_k \mathscr{Q}_k^t$*. Define* $\mathscr{Q}^* = \cup_{\text{polynomials } t} \mathscr{Q}^t$*.*

Since from last section we have a self-composable obfuscation for the single point function with general output, we simply put together the t programs $\mathcal{O}(Q_{\alpha_i,\beta_i})$, $i = 1,\ldots,t$ to obtain an obfuscation for $Q_{(\alpha_1,\beta_1)\ldots,(\alpha_t,\beta_t)}$.

Lemma 5. *The family of functions* $\mathscr{Q}^*$ *is efficiently obfuscatable in the random oracle model, in a self-composable manner.*

Proof Sketch: It is easy to see that $\mathscr{Q}^t \ll \{[\![F_1,\ldots,F_t]\!] : F_i \in \mathscr{Q}\}$. Since the obfuscation in Theorem 2 is self-composable, $\{[\![F_1,\ldots,F_t]\!] : F_i \in \mathscr{Q}\}$ is obfuscatable, and by Lemma 1, so is $\mathscr{Q}^t$ (and hence $\mathscr{Q}^*$). To see that this composition is self-composable, note that the obfuscation of an array of functions from $\mathscr{Q}^*$ is identical to the obfuscation of a (much larger) array of functions from $\mathscr{Q}$. □

5 Obfuscating a Complex Access Control Mechanism

Consider the following (interactive) access control task. There are multiple access points to various functions or secrets. There is an underlying directed multi-graph (possibly with multiple edges between nodes, and self-loops), with each node representing an access point. The user starts at a predefined access point, or "start node" and proceeds to establish her access privileges which allows her to move from one access point to another, through the edges of the graph. The access control task is the following:

- The user can reach an access point only by presenting credentials that can take her from the start node to that point.
- The user gains complete access to a function or secret available at an access point if and only if the user has reached that access point.
- The user does not learn anything about the structure of the graph, except what is revealed by the secrets at the access points she reached and the edges she traversed.

We specify this task as access to a black-box with which the user interacts, giving her credentials at various points and receiving the secrets; the black-box internally maintains the current access point of the user. But we would like to implement this task as a program which we then hand over to the user. To maintain the security of the task, we need to obfuscate this program.

In this section we explore this obfuscation problem. We show that in the random oracle model this access control mechanism can indeed be obfuscated. We model the interactive task as a non-interactive function (formulated below) which takes the "history" of interaction and gives a response to the last query.

Definition 12. *A graph-based access control problem X_G with parameters k and d is defined by the following:*

1. *Directed multi-graph G on k vertices. Each node $u \in k$ has at most d ordered neighbors $\mu_u^{(1)}, \ldots, \mu_u^{(d)}$. Let $E = \{(u, v, i) : v = \mu_u^{(i)}$ for some $i \in [d]\ \}$ be the set of all edges (i is used to differentiate between the multiple edges possible between the same pair of nodes).*
2. *A set of passwords on the edges $\{\pi_e | e \in E\}$, and*
3. *A set of secrets at the nodes $\{\sigma_v | v \in [k]\}$.*

Then,

$$X_G((i_1, x_1), \ldots, (i_n, x_n)) = \begin{cases} (v_n, \sigma_{v_n}) & \text{if } \exists v_0, \ldots, v_n, \in [k] \text{ and } e_0, \ldots, e_{n-1} \in E \\ & \text{such that } v_0 = 1, e_j = (v_j, v_{j+1}, i_j), \text{ and} \\ & x_j = \pi_{e_j} \\ \perp & \text{otherwise.} \end{cases}$$

We define the family of functions $\mathscr{X}$ as the set of all X_G with parameters (k, d) over all multi-graphs G, sets of edge-passwords and sets of node-secrets.

Above, (i, x) is a query in which the user provides a purported password x for the i-th edge going out of the "current" node. For later notational convenience we shall assume that there is no secret available at node 1: i.e., $\sigma_1 = \perp$.

We are interested in cases where the inputs to X_G are of size polynomial in k and d. We point out that there may be exponentially many *valid* inputs for which X_G outputs a secret (though the number of distinct secrets is only k). So it is not possible to obfuscate X_G directly using Lemma 5.

Instead we proceed in the following manner: each node is represented by the tuple $(v, \sigma_v, e_1, \ldots, e_d, \pi_{e_1}, \ldots, \pi_{e_d})$ where $e_i \in E$ (if there are less than d outgoing edges pick dummy values for the remaining edges). For each node $1 < u \leq k$ pick a random "key" κ_u from $\{0, 1\}^\ell$; let $\kappa_1 = 0^\ell$ (recall that 1 is the start node). Define the function $W_G^{\bar{\kappa}}$ as follows:

$$W_G^{\bar{\kappa}}(u, z, i, x) = \begin{cases} (v, \sigma_v, \kappa_v) & \text{if } z = \kappa_u \text{ and} \\ & \exists v \in [k] \text{ such that } \pi_{u,v,i} = x \\ \perp & \text{otherwise.} \end{cases}$$

The obfuscation consists of an obfuscation of $W_G^{\bar{\kappa}}$ (which is a multi-point function with at most kd input points where the output is not $\perp$, and hence can be obfuscated).

Intuitively, this is a good obfuscation because the adversary cannot find the randomly chosen key of a node κ_v, unless it was given out by the (obfuscated) function $W_G^{\bar{\kappa}}$. But the only way to obtain that is to give π_e for an edge leading to v from a node u to which the adversary already has the key. Since, to start with, the only key the adversary knows is κ_1, it must indeed traverse a path from 1 to v by providing the all the edge-passwords in order to get to v.

Formally, we first define a probabilistic program $\widetilde{W}_G$ which picks the random keys above to get a particular deterministic function $W_G^{\bar{\kappa}}$. Then we show that the family $\mathscr{X} \lll \widetilde{\mathscr{W}}$, where $\widetilde{\mathscr{W}}$ is the family of all $\widetilde{W}_G$ as above.

Definition 13. *Define the randomized algorithm $\widetilde{W}_G$ as follows: for $v \in [k]$, pick random keys $\kappa_v \leftarrow \{0,1\}^k$. On input (u, z, i, x) return $W_G^{\bar{\kappa}}(u, z, i, x)$.*

We define the family of functions $\widetilde{\mathscr{W}}$ as the set of all $\widetilde{W}_G$ (with parameters (k, d)) over all multi-graphs G, sets of edge-passwords and sets of node-secrets.

Lemma 6. *$\mathscr{X} \lll \widetilde{\mathscr{W}}$.*

Proof: For $X_G \in \mathscr{X}$ we pick $\widetilde{W}_G \in \widetilde{\mathscr{W}}$ and demonstrate M and N as required by the definition of the relation $\lll$.

M such that $M_G^{\widetilde{W}} \equiv X_G$: On input $(i_1, x_1), \ldots, (i_n, x_n)$ query $\widetilde{W}_G$ with $(1, 0^\ell, i_1, x_1)$; if $\widetilde{W}_G$ returns (v_2, σ_{v_2}), query it with $(v_2, \sigma_{v_2}, i_2, x_2)$ and so on, until it either returns $\perp$ or we reach the end of the input and receive (v_n, σ_{v_n}). In either case output this value.

N such that $N^{X_G} \approx \widetilde{W}_G$: N internally maintains two tables: one table is for keys κ_i, and one for *paths* to each node v from node 1, with edge passwords for each edge appearing on the edge. Initially it sets $\kappa_1 = 0^k$ and all other keys as $\perp$, and does not have any paths recorded for any node. On input (u, z, i, x) N checks if $z = \kappa_u \neq \perp$. If not it returns $\perp$. Else it will have recorded a path $(v_1 = 1, v_2, i_1, x_1), \ldots, (v_t, v_{t+1} = u, i_t, x_t)$ such that $x_j = \pi_{(v_j, v_{j+1}, i_j)}$. It makes a query $(i_1, x_1), \ldots, (i_t, x_t), (i, x)$ to X_G. If X_G responds with $\perp$, N outputs $\perp$. Else, it receives (v, σ_v) from X_G. It checks if a key has been already assigned to v; if not it picks a random key and assigns that to v. Then it returns (v, σ_v, κ_v).

It is not hard to see that for any PPT $\mathcal{S}'$ interacting with $\widetilde{W}_G$ or N^{X_G}, the output distribution of N^{X_G} is the same as that of $\widetilde{W}_G$, but both distributions conditioned on the event that $\mathcal{S}'$ never makes a query with a valid key which it did not receive as answer to a previous query. But that event is of negligible probability, and so $N^{X_G} \approx \widetilde{W}_G$. □

Note that $\widetilde{\mathscr{W}}$ is a family of probabilistic machines, such that if we consider the family obtained by fixing the random-tapes of machines in $\widetilde{\mathscr{W}}$ in all possible ways, we get a sub-family of $\mathscr{Q}^*$ (Definition 11). This sub-family is obfuscatable (because $\mathscr{Q}^*$ is obfuscatable, by Lemma 5). Then, from the above lemma and Lemma 2, we conclude the following.

Theorem 3. *The family $\mathscr{X}$ is efficiently obfuscatable in the random oracle model.*

6 Regular Expressions and Obfuscations

Let Σ be an alphabet (of constant size). We consider regular expressions over $\Sigma \cup \{\zeta^{L_1}, \ldots, \zeta^{L_t}\}$, where ζ^{L_i} are formal symbols corresponding to languages L_i. We define whether or not a string $s \in \Sigma^*$ *matches* such a regular expression $\rho(L_1, \ldots, L_t)$ as follows: s matches a symbol ζ^{L_i} if $s \in L_i$. The rest of the rules are the usual ones: a single character $a \in \Sigma$ matches itself; $s \in \Sigma^*$ matches $\rho_1|\rho_2$ if it matches either ρ_1 or ρ_2; s matches $\rho_1 \cdot \rho_2$ if $s = s_1 \cdot s_2$ such that s_1 matches ρ_1 and s_2 matches ρ_2; finally s matches ρ^* if s is the null-string, or $s = s_1 \cdot s_2 \cdots s_k$ where each s_i matches ρ. If s matches a regular expression ρ, we write $s \sim \rho$. Below $\mathcal{L}_{\rho(L_1,\ldots,L_t)}$ stands for the language defined as the set of all strings matching $\rho(L_1, \ldots, L_t)$.

6.1 Obfuscating $\mathcal{L}_{\rho(P_{\alpha_1},\ldots,P_{\alpha_t})}$

Consider the case when the languages L_i above are the point functions P_{α_i}. In this section we consider a family of functions $\mathscr{U}_\rho = \cup_k \mathscr{U}_{\rho\,k}$ where for all k and all $U_\rho^{\alpha_1,\ldots,\alpha_t} \in \mathscr{U}_{\rho\,k}$ there is a single fixed regular expression ρ. However, for each k, the point functions P_{α_i} belong to the $\mathscr{P}_k$, the family of point functions on $\cup_{j=0}^{k}\{0,1\}^j$. For brevity we denote $\mathcal{L}_{\rho(P_{\alpha_1},\ldots,P_{\alpha_t})}$ by $\mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$.

Definition 14. *Define the function $U_\rho^{\alpha_1,\ldots,\alpha_t}$ as follows: on input $x \in \{0,1\}^*$, check if $x \in \mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$. If so return $\alpha_1, \ldots, \alpha_t$; else return $\perp$. Let $\mathscr{U}_{\rho\,k} = \{U_\rho^{\alpha_1,\ldots,\alpha_t} : \alpha_i \in \cup_{j=0}^{k}\{0,1\}^j\}$, and $\mathscr{U}_\rho = \cup_k \mathscr{U}_{\rho\,k}$.*

Unless a string in the language $\mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$ is given as input $U_\rho^{\alpha_1,\ldots,\alpha_t}$ reveals nothing beyond the fact that the string is not in the language. We show that this function can be completely obfuscated.

Theorem 4. *For any regular expression ρ, the family $\mathscr{U}_\rho$ is efficiently obfuscatable in the random oracle model.*

To prove this, we introduce another family of functions $\mathscr{V}_\rho$, and show that $\mathscr{U}_\rho \ll \mathscr{V}_\rho$. Then, we show that $\mathscr{V}_\rho$ can be obfuscated (in the random oracle model).

Recall that ρ is a regular expression over the symbols $\Sigma \cup \{\zeta^{\alpha_1}, \ldots, \zeta^{\alpha_t}\}$. We can convert this to a deterministic finite-state automaton (DFA), with some of the edges labeled with ζ^{α_i}. Define a set $\mathcal{Z}_\rho \subseteq 2^{[t]}$ of subsets of $[t]$ as follows. If there is a path in the above DFA from the start state to some accept state, in which the set of non-Σ symbols appearing are $\{\zeta^{\alpha_i} : i \in Z \subseteq [t]\}$, then $Z \in \mathcal{Z}_\rho$. In other words, $\mathcal{Z}_\rho$ is the set of all subsets of α_i's, such that knowing α_i's in any of these subsets will enable one to construct a string in $\mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$. Note that $\mathcal{Z}_\rho$ can be constructed from ρ, independent of $\alpha_1, \ldots, \alpha_t$.

Definition 15. *Define the function $V_\rho^{\alpha_1,\ldots,\alpha_t}$ as follows: on input $(\beta_1, \ldots, \beta_t)$, $\beta_i \in \{0,1\}^*$, check if $\exists Z \in \mathcal{Z}_\rho$ such that $\forall i \in Z$, $\beta_i = \alpha_i$. If so return $\alpha_1, \ldots, \alpha_t$; else return $\perp$. Let $\mathscr{V}_{\rho\,k} = \{V_\rho^{\alpha_1,\ldots,\alpha_t} : \alpha_i \in \cup_{j=0}^{k}\{0,1\}^j\}$, and $\mathscr{V}_\rho = \cup_k \mathscr{V}_{\rho\,k}$.*

Lemma 7. *$\mathscr{U}_\rho \ll \mathscr{V}_\rho$ for all regular expressions ρ.*

Proof: Corresponding to $U_\rho^{\alpha_1,\ldots,\alpha_t} \in \mathscr{U}_\rho$ we pick $V_\rho^{\alpha_1,\ldots,\alpha_t} \in \mathscr{V}_\rho$.

Constructing M such that $M^{V_\rho^{\alpha_1,\ldots,\alpha_t}} \equiv U_\rho^{\alpha_1,\ldots,\alpha_t}$: As input $M^{V_\rho^{\alpha_1,\ldots,\alpha_t}}$ receives a string $x \in \{0,1\}^*$. It needs to check if $x \in \mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$. M chooses t substrings of x as guesses for $\alpha_1,\ldots,\alpha_t$. If $|x| = n$ there are $O(n^{2t}$ such choices. But by our convention, since ρ is fixed, t is a constant and n^{2t} is still polynomial in n, the size of input to M. For each such guess $(\beta_1,\ldots,\beta_t)$, M queries $V_\rho^{\alpha_1,\ldots,\alpha_t}$ on $(\beta_1,\ldots,\beta_t)$. If $V_\rho^{\alpha_1,\ldots,\alpha_t}$ returns $\perp$ for all choices, M also outputs $\perp$. If $V_\rho^{\alpha_1,\ldots,\alpha_t}$ returns $(\alpha_1,\ldots,\alpha_t)$ for any choice of $(\beta_1,\ldots,\beta_t)$, then M constructs the complete DFA (replacing the variables ζ^{α_i} with α_i) and checks if x is accepted by the DFA. If so, M outputs $\alpha_1,\ldots,\alpha_t$; if not it outputs $\perp$.

If $x \in \mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$, then there is some path in the DFA for ρ which accepts x. Let Z be the set of all i such that ζ^{α_i} appears on this accepting path. By the way $\mathcal{Z}_\rho$ was constructed, $Z \in \mathcal{Z}_\rho$. Further all these ζ^{α_i} appear as part of x. Thus, for some guess $\beta_1,\ldots,\beta_t$, it will be the case that for all of $i \in Z$ $\beta_i = \alpha_i$. Thus if $x \in \mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$, M will obtain all of $\alpha_1,\ldots,\alpha_t$ from $V_\rho^{\alpha_1,\ldots,\alpha_t}$, and will be able to verify that $x \in \mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$. On the other hand if $x \notin \mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$ either $\alpha_1,\ldots,\alpha_t$ are not revealed to M, or they are and M will discover that $x \notin \mathcal{L}_{\rho(\alpha_1,\ldots,\alpha_t)}$. In either case M will output $\perp$, as required.

Constructing N such that $N^{U_\rho^{\alpha_1,\ldots,\alpha_t}} \equiv V_\rho^{\alpha_1,\ldots,\alpha_t}$: As input $N^{U_\rho^{\alpha_1,\ldots,\alpha_t}}$ receives t strings $(\beta_1,\ldots,\beta_t)$. It needs to check if there is any $Z \in \mathcal{Z}_\rho$ such that $\forall i \in Z$ $\alpha_i = \beta_i$. Associated with each Z is a path from the start state to an accept state in which the variable ζ^{α_i} appear for exactly those $i \in Z$. N chooses for each Z such a path, and constructs a string x_Z corresponding to that path, substituting β_i for ζ^{L_i}. It then submits x_Z to $U_\rho^{\alpha_1,\ldots,\alpha_t}$ (to which it has oracle access). If $U_\rho^{\alpha_1,\ldots,\alpha_t}$ responds with $\perp$ for all x_Z, $Z \in \mathcal{Z}_\rho$ then N outputs $\perp$. If $U_\rho^{\alpha_1,\ldots,\alpha_t}$ responds with $\alpha_1,\ldots,\alpha_t$ for any x_Z, then N then checks if $\exists Z \in \mathcal{Z}_\rho$ $\forall i \in Z$ $\alpha_i = \beta_i$, and responds accordingly. It can be easily verified that $N^{U_\rho^{\alpha_1,\ldots,\alpha_t}} \equiv V_\rho^{\alpha_1,\ldots,\alpha_t}$. □

Next we observe that $\mathscr{V}_\rho \ll \mathscr{Q}^*$, where $\mathscr{Q}^*$ is the class of multi-point functions with general output (Definition 11).

Lemma 8. $\mathscr{V}_\rho \ll \mathscr{Q}^*$

Proof: Let $\mathcal{Z}_\rho = \{Z_1,\ldots,Z_\ell\}$, and for each $Z_i \in \mathcal{Z}_\rho$, let the string γ_i be $(\gamma_i^1,\ldots,\gamma_i^t)$ where if $j \in Z_i$, $\gamma_i^j = \alpha_j$ and else $\gamma_i^j = 0$.

For every $V_\rho^{\alpha_1,\ldots,\alpha_t} \in \mathscr{V}_\rho$, consider $Q = Q_{(\gamma_1,\Delta),\ldots,(\gamma_\ell,\Delta)} \in \mathscr{Q}^*$ where $\Delta = (\alpha_1,\ldots,\alpha_t)$ (i.e., if Q is given one of the strings $\gamma_1,\ldots,\gamma_\ell$, it outputs Δ. It is easy to verify that the following machines M and N are as required by Definition 4.

M^Q, on input $(\beta_1,\ldots,\beta_t)$ does the following: for each $Z_i \in \mathcal{Z}_\rho$ it constructs a string $\delta_i = (\delta_i^1,\ldots,\delta_i^t)$ where if $j \in Z_i$, $\delta_i^j = \beta_j$ and else $\delta_i^j = 0$; then it queries Q with δ_i; if for any i it receives Δ from Q it outputs that and else $\perp$.

$N^{V_\rho^{\alpha_1,\ldots,\alpha_t}}$ on input $\delta = (\delta^1,\ldots,\delta^t)$, queries $V_\rho^{\alpha_1,\ldots,\alpha_t}$ with δ. If it receives $\perp$ as an answer, it also outputs $\perp$. Else it receives Δ, and can then can compute $Q(\delta)$, which it outputs. □

By Lemma 5, $\mathscr{Q}^*$ is obfuscatable, thereby completing the proof of $\mathscr{V}_\rho$ being obfuscatable. To complete the proof of Theorem 4, we appeal to Lemma 1, along with Lemma 7 and the above fact that $\mathscr{V}_\rho$ is obfuscatable.

We remark that the construction above can easily be extended to also produce an arbitrary secret output if the input matches the regular expression.

6.2 Obfuscating a function related to $\rho(L_1, \ldots, L_t)$

In this section we allow ρ to be part of the function (and therefore can have size polynomial in k). We are interested in matching a given string against $\rho(L_1, \ldots, L_t)$ without compromising the black-box nature of $[\![L_1, \ldots, L_t]\!]$. The family of functions we are interested in is $\mathscr{F}_\mathcal{C}$ below.

Definition 16. *Define $G_\rho^{L_1,\ldots,L_t}$ and $F_\rho^{L_1,\ldots,L_t}$ as follows:*

$$G_\rho^{L_1,\ldots,L_t}(a,x) = \begin{cases} \rho & \textit{if } a = 1 \\ L_{a-1}(x) & \textit{if } a \in \{2,\ldots,t+1\} \\ \bot & \textit{otherwise} \end{cases}$$

$$F_\rho^{L_1,\ldots,L_t}(a,x) = \begin{cases} 1 & \textit{if } a = 0 \textit{ and } x \textit{ matches } \rho(L_1,\ldots,L_t) \\ 0 & \textit{if } a = 0 \textit{ and } x \textit{ does not match } \rho(L_1,\ldots,L_t) \\ G_\rho^{L_1,\ldots,L_t}(a,x) & \textit{otherwise} \end{cases}$$

$$\mathscr{G}_\mathcal{C} = \{G_\rho^{L_1,\ldots,L_t} \ : \ \rho \textit{ a regular expression and } L_i \in \mathcal{C}\}$$
$$\mathscr{F}_\mathcal{C} = \{F_\rho^{L_1,\ldots,L_t} \ : \ \rho \textit{ a regular expression and } L_i \in \mathcal{C}\}$$

In other words, both $G_\rho^{L_1,\ldots,L_t}$ and $F_\rho^{L_1,\ldots,L_t}$ provide access to the languages L_i and to (the description of) the regular expression ρ. In addition, $F_\rho^{L_1,\ldots,L_t}$ gives access to the language defined by the regular expression $\rho(L_1, \ldots, L_t)$.

Theorem 5. *$\mathscr{F}_\mathcal{C}$ is obfuscatable if and only if $\{[\![L_1, \ldots, L_t]\!] \ : \ L_i \in \mathcal{C}\}$ is. Further this statement holds restricted to efficient obfuscations too.*

First we prove the following lemma, which is the heart of the proof. It shows how to evaluate the regular expressions involving L_i's just with access to $\mathscr{G}_\mathcal{C}$.

Lemma 9. *$\mathscr{F}_\mathcal{C} \ll \mathscr{G}_\mathcal{C}$ and $\mathscr{G}_\mathcal{C} \ll \mathscr{F}_\mathcal{C}$, for all families $\mathcal{C}$.*

Proof: It is easy to see that $\mathscr{G}_\mathcal{C} \ll \mathscr{F}_\mathcal{C}$. For the other direction, we have to demonstrate the polynomial time oracle machines M and N as in Definition 3. But N is trivial, and so is M's behaviour when on input (a, x), it sees $a \neq 0$. The non-trivial case is when $a = 0$: M should match the input x with the regular expression ρ with only black-box access to L_i. We give a fairly efficient algorithm using dynamic programming to achieve this.

First M obtains the regular expression ρ from G (by giving input $(1, \epsilon)$. It constructs a tree corresponding to ρ with leaf nodes corresponding to symbols from $\Sigma \cup \{\zeta^{L_1}, \ldots, \zeta^{L_n}\}$. Each internal node corresponds to one of the three operators $|$, $\cdot$ and

*; in the first two cases the node will have two children and in the last case a single child. The root node corresponds to the whole regular expression ρ. The algorithm will consider the set S of all substrings of the input string $x = x_1 \dots x_n$; i.e., $S = \{x_i^j \ : \ 1 \le i \le j \le n\} \cup \{\epsilon\}$. For each node it will try to find out all the strings in S which match the regular expression at that node. This is done bottom-up in the tree. To obtain this information at the leaf nodes, M makes $O(n^2)$ queries to each L_i.

Given this information for the children of a node, the information for that node itself can be obtained. In the case of a (|)-node (denoted by $\mathcal{Q} = \mathcal{Q}_1 | \mathcal{Q}_2$) this is simple: for each string $s \in S$ check if $s \sim \mathcal{Q}_1$ or $s \sim \mathcal{Q}_2$. If either case holds record that $s \sim \mathcal{Q}$. For ($\cdot$)-node $\mathcal{Q} = \mathcal{Q}_1 \cdot \mathcal{Q}_2$ we do the following:

```
for each s ∈ S do
  for i = 0 to |s| do
    if s_1^i ~ Q_1 AND s_{i+1}^{|s|} ~ Q_2 then
      record s ~ Q
```

The checks $s_1^i \sim \mathcal{Q}_1$ and $s_{i+1}^{|s|} \sim \mathcal{Q}_2$ are done by checking if those matchings have already been recorded. The (*)-nodes require a little more work. At a node $\mathcal{Q} = \mathcal{Q}_1^*$ we do the following:

```
Let Q_1^1 denote Q_1
for k = 2 to n do
  for each s ∈ S\{ε} do
    for i = 0 to |s| do
      if s_1^i ~ Q_1^{k-1} AND s_{i+1}^{|s|} ~ Q_1 then
        record s ~ Q_1^k
record ε ~ Q
for each s ∈ S\{ε} do
  if s ~ Q_1^k for some k ∈ {1, ..., n} then
    record s ~ Q
```

It is not hard to see that at each node the algorithm correctly records all $s \in S$ which match the node. Finally, it checks if $x \sim \rho$ by checking if it is recorded at the root node. □

Proof: **(of Theorem 5)** By the above Lemma and Lemma 1, we can obfuscate $\mathscr{F}_{\mathcal{C}}$, if and only if we can obfuscate $\mathscr{G}_{\mathcal{C}}$. We can view $G \in \mathscr{G}_{\mathcal{C}}$ as $[\![\langle\rho\rangle, [\![L_1, \dots, L_n]\!]]\!]$, where $\langle\rho\rangle$ stands for the constant (and hence trivially obfuscatable) function which outputs ρ. Then by Lemma 3, $\mathscr{G}_{\mathcal{C}}$ is obfuscatable if and only if $\{[\![L_1, \dots, L_n]\!] \ : \ L_i \in \mathcal{C}\}$ is obfuscatable. □

7 Obfuscating Neighborhoods in Tree Metrics

Point functions are identity checks- they check if the input is identical to a particular value. A natural relaxation thereof is a neighborhood check. Consider some metric space from which the inputs are drawn. We would like to have a program which checks if the input is "near" a hidden point.

We work in a restricted metric space- the space of "tree metrics," where the the points are nodes in a (rooted, undirected) tree, and the distance between two points is the length of the (unique) path between them. (We can allow a metric space that can be decomposed as a collection of a *constant* number of tree metrics, but for simplicity we stick to a single tree-metric.)

Let $\mathcal{M}$ stand for the metric space as well as (by abuse of notation) the tree defining it. Let $\mathrm{d}_{\mathcal{M}}(\cdot,\cdot)$ be the distance function in $\mathcal{M}$.

Definition 17. *Define the function $T_\alpha^{\mathcal{M}} : \mathcal{M} \to \mathcal{M} \cup \{\perp\}$ as follows:*

$$T_\alpha^{\mathcal{M}}(x) = \begin{cases} \alpha & \mathrm{d}_{\mathcal{M}}(\alpha, x) \le \delta \\ \perp & \mathrm{d}_{\mathcal{M}}(\alpha, x) > \delta \end{cases}$$

$\mathscr{T}_k = \{T_\alpha^{\mathcal{M}} \ : \ \mathcal{M}$ *a tree-metric* $, |\mathcal{M}| = 2^{O(k)}, \alpha \in \mathcal{M}\}$ *and* $\mathscr{T} = \cup_k \mathscr{T}_k$.

Obfuscating δ-neighborhoods in general metric spaces (beyond what can be achieved by exhaustively searching the entire δ-neighborhood of a point) is a challenging problem. But we show that for tree metrics this problem can be satisfactorily solved using a simple technique. To obfuscate $T_\alpha^{\mathcal{M}}$, traverse the tree $\mathcal{M}$, starting at the node α, towards the root of the tree, for a distance δ, and pick the node at which we finish. (If we reach the root before δ steps pick the root.) Call this node β. We show that obfuscating $T_\alpha^{\mathcal{M}}$ is essentially the same as obfuscating the point function on β with output α (which as we have shown, can be efficiently obfuscated in the random oracle model).

Lemma 10. *$\mathscr{T} \ll \mathscr{Q}$ (where $\mathscr{Q}$ is the point function with general output, as in Definition 10).*

Proof: For $T^{\mathcal{M}} \in \mathscr{T}$ we pick $Q_{\beta,\alpha} \in \mathscr{Q}$. $Q_{\beta,\alpha}$ is the function which outputs α on input β and $\perp$ everywhere else.

$N^{T_\alpha^{\mathcal{M}}}$ *works as follows* : On input $x \in \mathcal{M}$ query $T_\alpha^{\mathcal{M}}$ with x. If x were indeed equal to β then $T_\alpha^{\mathcal{M}}$ would respond with α. So if $T_\alpha^{\mathcal{M}}$ gives $\perp$ return $\perp$. If it gives α, locate β by traversing $\mathcal{M}$, and check if the x is indeed β or not and answer accordingly.

$M^{Q_{\beta,\alpha}}$ *works as follows* : on input $x \in \mathcal{M}$, check the first 2δ ancestors of x for being identical to β (using $Q_{\beta,\alpha}$). If $Q_{\beta,\alpha}$ returns α on some query, check $\mathrm{d}_{\mathcal{M}}(x,\alpha)$ and answer appropriately. If it returns $\perp$ in all 2δ queries, then it is easy to see that the distance $\mathrm{d}_{\mathcal{M}}(x,\alpha) > \delta$. In this case, output $\perp$. $\square$

By Lemma 1 and Theorem 2, we get:

Theorem 6. *$\mathscr{T}$ is obfuscatable in the random oracle model.*

8 Conclusions and Open Problems

We have given the first positive results and techniques for program obfuscation, but many important open problems remain. We are hopeful our reduction and composition techniques will aid in resolving these problems. The most pressing open problem is to extend our positive results beyond what we have. In particular, can regular languages be obfuscated? Is there *any* example of a keyed cryptographic primitive (even a contrived one) other than password checking which can be obfuscated? Another important problem to be resolved is to find *any* non-trivial obfuscation result without using the random oracle model. Our approach, of reducing obfuscation of one family to obfuscating another, could then be used to produce more obfuscations in the plain model. Also, such techniques are useful in a model where some basic functions may be obfuscated in hardware; so one direction to pursue is to explore developing these techniques further.

Acknowledgments

We thank Dan Boneh for many useful discussions, and collaboration in early parts of this work. We also thank the anonymous referees for detailed comments on the presentation.

References

1. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of CRYPTO 2001*.
2. Ran Canetti, Daniele Micciancio, and Omer Reingold. Perfectly one-way probabilistic hash functions. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing 1998*.
3. Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report TR00-03, The Department of Computer Science, University of Arizona, February 2000.
4. S. Chow, H. Johnson, P. C. van Oorschot, and P. Eisen. A White-Box DES Implementation for DRM Applications. In *Proceedings of ACM CCS-9 Workshop DRM 2002*.
5. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. Preliminary versions appeared at *CRYPTO 1989* and *STOC 1990*. *Journal of the ACM*, 43(3):431–473, 1996.
6. Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *Proceedings of CRYPTO 2003*.
7. Matthias Jacob, Dan Boneh, and Edward Felten. Attacking an obfuscated cipher by injecting faults In *Proceedings of ACM CCS-9 Workshop DRM 2002*.
8. Benjamin Lynn, Manoj Prabhakaran, Amit Sahai. Positive Results and Techniques in Obfuscation. In the Cryptology ePrint Archive (http://eprint.iacr.org), 2004.