Towards a Theory of Special-Purpose Program Obfuscation

Muhammad Rizwan Asghar*, Steven D. Galbraith†, Andrea Lanzi‡, Giovanni Russello* and Lukas Zobernig*†
*School of Computer Science, The University of Auckland, New Zealand
†Department of Mathematics, The University of Auckland, New Zealand
‡Computer Science Department, Universita degli studi di Milano, Italy

***Abstract*—Most recent theoretical literature on program obfuscation is based on notions like virtual black box (VBB) obfuscation and indistinguishability obfuscation (iO). These notions are very strong and are hard to satisfy. Further, they offer far more protection than is typically required in practical applications. On the other hand, the security notions introduced by software security researchers are suitable for practical designs but are not formal or precise enough to enable researchers to provide a quantitative security assurance. Hence, in this paper, we introduce a new formalism for practical program obfuscation that still allows rigorous security proofs. We believe our formalism will make it easier to analyse the security of obfuscation schemes. To show the flexibility and power of our formalism, we give a number of examples. Moreover, we explain the close relationship between our formalism and the task of providing obfuscation challenges.**

I. Introduction

The goal of program obfuscation is often informally stated as "to hide semantic properties of program code from a Man-At-The-End (MATE) attacker who has access to an executable file of the program". This is an imprecise notion, and despite more than twenty years of research in software security that has led to a variety of obfuscation tools and techniques, there is still not a useful and practical formalism for what it means to "hide semantic properties of program code".

In 1996, Goldreich and Ostrovsky [1] suggested that obfuscation should prevent "the release of any information about the original program which is not implied by its input/output relation and time/space complexity". In 1997, Collberg, Thomborson, and Low [2] remarked that "developers are mostly frightened by the prospect of a competitor being able to extract proprietary algorithms and data structures from their applications".

The landmark paper by Barak *et al.* [3], [4] in 2001 introduced the notion of Virtual Black Box (VBB) obfuscation. Informally, a program is VBB obfuscated if an attacker can learn nothing more, when given the executable code, than what could be learned from running it (*i.e.,* oracle access). This very strong notion captures the intention of "hiding semantic properties of program code", but unfortunately, Barak *et al.* show it is impossible to achieve in general.

The work of Barak *et al.* (and also much of the work in the software security community) aims to produce *general-purpose* obfuscation techniques. This is a worthwhile goal, but in our opinion, it is too ambitious. In contrast, there is also a very successful theme in obfuscation research to consider specific classes of programs and to give obfuscation methods that are designed with a specific goal in mind. We call this *special-purpose obfuscation*.

As discussed by Collberg [5], program obfuscation still does have a number of practical applications. Hence, there is a need to find practical solutions.

Several authors have recognised the need for more precise and formal approaches to practical obfuscation. Kuzurin *et al.* [6], in their discussion of solutions from the software security community, state "The principal drawback of all known obfuscation techniques is that they do not refer to any formal definition of obfuscation security and do not have a firm ground for estimating to what extent such methods serve the purpose." They go on to say "We believe that further progress in the study of program obfuscation may ensue primarily from the development of a solid framework which makes it possible to set up security definitions for program obfuscation in the context of various applications". Similarly, Xu *et al.* [7] compare applied (they call it "code-oriented") and theoretical (named "model-oriented") obfuscation approaches. They claim "current evaluation metrics are not adequate for security purposes". Their conclusion urges "the communities to rethink the notion of security and usable program obfuscation and facilitate the development of such obfuscation approaches in the future."

The goal of this paper is to respond to these calls and to propose a useful formalism for obfuscation.

Our Results: The aim of this paper is to introduce a formalism for program obfuscation that achieves three main goals:

1) It is powerful enough to capture real-world applications of program obfuscation.
2) It is precise enough to allow formal analysis of obfuscation techniques and to prove their security.
3) Being weaker than VBB, we avoid the impossibility results of Barak *et al.* [3], [4]. Our examples show that our notion is possible, at least in certain cases.

Our formalism uses the notion of *assets* of a class of programs. In practice, software developers usually have a particular secret or intellectual property that they wish to protect, rather than desiring to secure "all semantic properties" of a program. Hence, we believe it is more effective to develop special-purpose obfuscators that are targeted to protect a *particular type* of asset in a *particular class* of programs.

978-1-6654-0392-4/20/$31.00 ©2020 IEEE
DOI 10.1109/TrustCom50675.2020.00061

The rest of this paper is organised as follows. In Section II, we review related work. Our formalism is presented in Section III. Two of our contributions are to define what it means to specify a class of programs, and to define the correctness of an attacker that learns an asset. One theme that we want to stress is the connection between formalising security and providing obfuscation challenges. We discuss this connection in Section IV. Section V illustrates how our formalism is intended to be used. The goal of Section VI is to discuss how various real-world examples can be modelled within our formalism. In Section VII, we discuss the security of applying different obfuscators iteratively on a program.

II. Previous Work on Obfuscation Definitions

Goldreich and Ostrovsky [1] suggested a notion of obfuscation security with a similar flavour to the later notion of *VBB obfuscation* defined by Barak *et al.* [3]. In the software security field, there has been a broader range of notions discussed, some of them imprecisely. We briefly survey them below.

Collberg, Thomborson, and Low [2] discuss general software engineering measures of code complexity. Several other papers have discussed complexity measures from software engineering (*e.g.,* Anckaert *et al.* in [8]).

The notion of asset has been discussed in several papers, as we now explain. Basile *et al.* [9] introduce the notion of "business goals" of a program and call the developer/obfucator the "defender". The "defender associates many business goals to software" and a successful attack means "business goals not satisfied anymore".

In Chapter 5 of [10], Collberg and Nagra give the notion of an *asset*, specified by programmer/developer. Their work restricts focus in each example to a class "Input Programs".

Ahmadvand, Pretschner, and Kelbert (see Section 3.1 of [11]) define assets to be "elements whose integrity needs to be protected against attacks" and "valuable data or sensitive behaviour(s) of the system, tampering with which renders the system's security defeated".

Schrittweiser *et al.* [12] give a selection of assets phrased in terms of attack goals. These goals are illustrated in the context of Digital Rights Management (DRM) applications.

Note that the choice of asset depends on the context, and so a particular asset only makes sense within some class of programs. Hence, it is appropriate to consider *special-purpose obfuscation tools*, tailored for certain types of asset. This idea is implicit in much of the theoretical research on obfuscation. An explicit argument in favour of special-purpose obfuscation is given in Section 3.4 of [13]: "What we advocate here is to consider specific obfuscators for specific function families".

III. Our Formalism

Following Collberg and Nagra [10], we take a developer-centered approach to defining security of an obfuscator. Building on [9]–[11], [14], we work with the notion of an asset and define security in a similar way to the definition of potency given as Definition 4.3 in [10]. We do not concern ourselves in this paper with stealth (*i.e.,* can an attacker determine that some form of obfuscation has been applied to the program segment) because, in most non-malware contexts, it will be public knowledge that obfuscation is being used.

Also, following much of the theoretical work, we restrict attention to *classes of programs* rather than trying to formulate definitions that work for any circuit or any Turing machine. Two of our contributions are to define what it means to specify a class of programs, and to define the correctness of an attacker that learns an asset.

We assume that the adversary is a Turing machine, so that we can use the standard theoretical tools of security reductions.

We will evaluate the threat of an attacker in terms of its running time. To be able to make complexity-theoretic statements (*e.g.,* the attacker is polynomial time or exponential time), there must be a parameter that takes arbitrarily large values with respect to which the running time is expressed. It does not make sense to think of an "exponential-time attack" against a single program. Hence, we need to parameterise program classes with a security parameter n.

There are three entities in our formalism:

- **Programmer/developer:** A human who has developed a program (written in some high-level source code) that contains some asset or intellectual property.
- **Obfuscator/defender:** A randomised algorithm that takes as input a program in some format and outputs a program in some format. Note that the input format may not be same as output format, *e.g.,* a compiler is an obfuscator according to some research papers. We study *special-purpose obfuscators* that depend on the particular program class and asset.
- **Attacker/de-obfuscator:** An algorithm that takes program code in some format and tries to learn the asset in the program.

We now state our formalism, step by step.

A. Program (Segment) Classes

Definition 1. *A* ***program class*** *(or* ***class of program segments****) $\mathcal{C}$ is a set of programs, all written in the same language. For each $n \in \mathbb{N}$, there is a (possibly empty) subset $\mathcal{C}_n \subseteq \mathcal{C}$. The sets $\mathcal{C}_n$ are disjoint and $\mathcal{C} = \cup_{n\in\mathbb{N}}\mathcal{C}_n$. There is a polynomial $p(x)$ such that for all $n \in \mathbb{N}$ and all $P \in \mathcal{C}_n$ the running time of P on any input is bounded by $p(n)$. Further, we require that the class $\mathcal{C}$ has a compact description in the following sense:*

1) *There is an efficient program generator $\mathsf{Gen}_{\mathcal{C}}$ for the class $\mathcal{C}$. Formally, this is a randomised polynomial-time algorithm that on input 1^n (a bitstring of length n) computes in time bounded by a fixed polynomial in n a random program $P \in \mathcal{C}_n$ and some auxiliary data aux. The auxiliary data is typically related to assets in the program, and may include information that is needed by the obfuscator and/or to verify the asset.*
2) *$\mathcal{C}_n$ should asymptotically be super-polynomial in size, so for all polynomials $p(n)$ we have $\#\mathcal{C}_n \geq p(n)$ for sufficiently large n.*

Here, n acts as a security parameter, and the sets $\mathcal{C}_n$ are intended to consist of programs with a similar security level in terms of finding an asset.

The requirements on $\mathcal{C}_n$ are implicit in the previous literature on theoretical obfuscation, but we believe it is worthwhile to make them explicit. For example, Barak *et al.* [3] define obfuscation for "a family of Turing machines" and "a family of circuits", without explaining how such a family is represented. They define a security experiment that implicitly assumes the ability to sample Turing machines/circuits from the family. We have made this explicit with the algorithm Gen_C.

We could consider $\mathcal{C}$ to be a set of circuits or Turing machines. But in practice, it will be more convenient to think of a set of instructions in some programming language. The formalism can be applied to programs in any code. For example, the class $\mathcal{C}$ could comprise programs in source code written by the developer, or bytecode, or compiled native code.

In practice, program classes are defined by software developers (we will give examples in Section VI) so that they have some meaningful common feature or functionality. The parameter n has some natural meaning to the developer, such as bit-length of user input or key-length of some secret that is stored in the program. We stress that we focus on program segments, since a large software project will have many components and the type of intellectual property and protection level required may vary among them.

The requirement that $\mathcal{C}$ is defined by a random program generator may seem counter-intuitive. However, we view this as a kind of Kerckhoff's principle. The crucial point is that the hard part of software development is finding the exactly right program $P \in \mathcal{C}_n$ for the problem at hand. So, a software developer can easily generate a random program $P \in \mathcal{C}_n$, but the probability to have the asset must be low.

B. Assets

A rigorous formulation of asset is non-trivial, since the set of all possible assets might be exponentially large (*e.g.,* the set of all inputs that are accepted by some evasive function), or there may be many different representations for an asset (*e.g.,* different representations of the same control flow graph). Our key insight is that the focus should be less on what is an asset and more on how the software developer recognises that the asset has been successfully found by the adversary.

Definition 2. *Let $\mathcal{C} = (\mathcal{C}_n)$ be a class of programs. An* ***asset space*** *$\mathcal{A}$ for $\mathcal{C}$ is a family of sets $(\mathcal{A}_n)_{n\in\mathbb{N}}$. An* ***asset*** *for $\mathcal{C}$ is a sequence of functions $\mathsf{a}_n : \mathcal{C}_n \to \mathcal{P}(\mathcal{A}_n)$, where $\mathcal{P}(\mathcal{A}_n)$ is the power set of $\mathcal{A}$. (This function does not need to be efficiently computable.)*

An asset is ***efficiently verifiable*** *if there is an algorithm $\mathcal{V}$ (the* ***asset verifier****) and a polynomial $q(x)$ such that for all $n \in \mathbb{N}$ and all $(P, \mathsf{aux}) \leftarrow \mathsf{Gen}_\mathcal{C}(n)$ (so $P \in \mathcal{C}_n$), and all $a \in \mathcal{A}_n$, $\mathcal{V}(P, \mathsf{aux}, a)$ outputs 1 if $a \in \mathsf{a}_n(P)$ and 0 otherwise, the running time of $\mathcal{V}$ on $\mathcal{C}_n$ is bounded by $q(n)$.*

The above definition has a deterministic verifier that always outputs the correct answer. But one can also consider a randomised verifier that is correct with some overwhelming probability; this can arise in some settings. We have given $\mathcal{V}$ the original code of P and the auxiliary information in order to check the asset, but in many applications, oracle access to P would suffice.

Finally, we can discuss the attacker and the security goal. We follow Kerckhoffs's principle and so we assume the attacker knows: the class $\mathcal{C}$ of programs being obfuscated; the asset space $\{(\mathcal{A}_n, \mathsf{a}_n)\}$; and the obfuscation tool being used to protect the asset. Hence, our formalism does not address the question of *stealth*, *i.e.,* can an attacker determine that part of a program has been obfuscated, and if so, using what tool? Indeed, we believe that highly secure obfuscation and stealth are probably incompatible.

Assets should not be guessable or easily *learnable* from the program. If the asset is learnable from executing the program on various inputs (and logging those inputs and corresponding outputs) then it is clearly impossible to protect the asset using any form of obfuscation. Hence, we require $\#\mathcal{A}_n$ to be large.

For example, the class of programs that computes linear functions $\mathcal{C}_n = \{L : \mathbb{F}_q^n \to \mathbb{F}_q^m\}$ is learnable by executing such a program on unit vectors in $\mathbb{F}_q^n$. Hence, it makes no sense to obfuscate $\mathcal{C}_n$. There are many program classes that might be tempting to obfuscate, but are actually efficiently learnable. For example, consider a predicate P that tells whether or not an input $x \in \mathbb{Z}$ satisfies $x < C$ for some secret constant C. One can learn C using black-box access to P by binary search.

It is the software developer's job to define "useful" or "meaningful" assets. There may be some particular programs $P \in \mathcal{C}_n$ that are easily learnable, but we require that a random program in the class is not learnable. More precisely, the output distribution of $\mathsf{Gen}_\mathcal{C}$ needs to be such that, for large n, instances output by $\mathsf{Gen}_\mathcal{C}(n)$ are not learnable with high probability.

C. Secure Obfuscators

We now define what is meant by an obfuscator. First, it is a compiler. So, it takes programs in some language and converts to another form.

Definition 3. *Let $\mathcal{C} = (\mathcal{C}_n)$ be a class of programs defined by a generator $\mathsf{Gen}_\mathcal{C}$, $\mathcal{A}_n$ a sequence of asset sets, $\mathsf{a} = (\mathsf{a}_n)$ an asset and $\mathcal{V}$ an efficiently computable asset verifier. A* ***secure obfuscator*** *for $(\mathcal{C}, \mathcal{A}, \mathsf{a})$ is a randomised algorithm $\mathsf{Obf}(P, \mathsf{aux})$ that transforms a program $(P, \mathsf{aux}) \leftarrow \mathsf{Gen}_\mathcal{C}(n)$ in $\mathcal{C}_n$ to a program $P' = \mathsf{Obf}(P, \mathsf{aux})$ in $\mathcal{C}'_n$ for some class of programs $\mathcal{C}' = (\mathcal{C}'_n)$. Further, we require:*

1) *(Correctness) For all $P \in \mathcal{C}$ and all*[1] *inputs x we have $P'(x) = P(x)$.*
2) *(Efficiency) There is a polynomial $p_1(t) \in \mathbb{R}[t]$ such that $|P'| \leq p_1(|P|)$ for all $P \in \mathcal{C}$ (for some canonical measure $|P|$ and $|P'|$ of program size in the languages of $\mathcal{C}$ and $\mathcal{C}'$). There is a polynomial $p_2(t) \in \mathbb{R}[t]$ such that if $P(x)$ runs in time T for an input x then $P'(x)$ runs in time $p_2(T)$.*

[1] One can relax this to almost all inputs if necessary.

3) *(Security) For all probabilistic polynomial-time algorithms A that know $\mathsf{Gen}_{\mathcal{C}}, \mathsf{Obf}, \mathcal{V}$, such that $A : \mathcal{C}'_n \to \mathcal{A}_n$ the probability*

$$p_n = \Pr\Big(\mathcal{V}(P, \mathsf{aux}, A(P')) = 1 :$$

$$(P, \mathsf{aux}) \leftarrow \mathsf{Gen}_{\mathcal{C}}(1^n), P' = \mathsf{Obf}(P, \mathsf{aux})\Big)$$

over $(P, \mathsf{aux}) \leftarrow \mathsf{Gen}_{\mathcal{C}}$ and the random choices of Obf and A, is negligible in n. By p_n being negligible in n we mean: for all polynomials $q(x) \in \mathbb{Z}[x]$ there is some N such that, for all $n > N$, $p_n \leq 1/|q(n)|$.

One of the main motivations behind our formalism is to be able to write down the equation in item (3) of the definition, which gives falsifiable criteria for security of an obfuscator. The connection between this definition and setting up an obfuscation challenge is given in Section IV. Note that we do not specify the power of the attacker A, apart from that it is provided with the obfuscated program $P' = \mathsf{Obf}(P, \mathsf{aux})$. At a purely formal level, there is no distinction between an attacker who inspects the source of the program, or who runs a symbolic analyser on the program, or who executes the program in a debugger. Hence our formalism includes the strongest possible dynamic attacker.

IV. Connection with Obfuscation Challenges

It may be helpful to the reader to see that the problem of defining obfuscation security is exactly the same as providing automated obfuscation challenges. For example, the Tigress project [15]–[17] provides reverse engineering challenges that are obfuscated programs with a cash prize for the first contestant to successfully reverse-engineer them. Such competitions are a popular way to encourage research into breaking obfuscation schemes, and to give insight into the security of current obfuscation methods. To set up such a competition, one needs a means to generate random programs to be obfuscated, and a means to decide whether a contestant has successfully completed the task.

Banescu, Collberg, and Pretschner [17] build a "code generator that can generate large numbers of arbitrarily complex random C functions". Essentially, their generator produces a random hash function of a certain type, and the program is a licence check. Determining the success of the contestant is clear, *i.e.,* find an input that is accepted by the licence check. This generator has been implemented in Tigress as `RandomFuns`.

The requirements to set up a successful obfuscation challenge are exactly captured by our formalism. First, one needs to have a well-defined class of programs and an algorithm Gen to generate a random program from this class. Second, one needs a well-defined security goal (the asset). Third, one needs an efficient and automated method to judge if a contestant has found the asset (the asset verifier). A potential future application of our formalism is to set up obfuscation challenges for programs other than simple licence checks.

The issue of learnability also arises in this context. If the asset is guessable, or learnable from the input-output behaviour, then an adversary can win the game without reverse-engineering the code. Hence, it only makes sense to offer an obfuscation challenge when the asset is not learnable. This requirement is implicit in the previous work.

Finally, there is an interesting subtlety about determining whether a competitor has successfully determined the asset. In our formalism, the verification algorithm $\mathcal{V}$ takes inputs (P, aux, a), and so requires the original program and the auxiliary data. This seems to be necessary in some contexts; whereas, in other contexts (*e.g.,* the ones studied in [17]), one can verify the asset using only $P' = \mathsf{Obf}(P, \mathsf{aux})$. It follows that obfuscation challenges may come in two flavours: those that are publicly verifiable, and those for which only the problem-setter can decide if a solution is correct.

V. Formalism in Action

The purpose of this section is to illustrate how our formalism is intended to be used. We do this using a very simple example, namely *point functions*, which include licence checks and password checks. Point functions may seem a very primitive class of programs, but they have applications in practice. For example, Sharif *et al.* [18] consider a predicate that triggers malicious behaviour in malware and encrypts the program block using the satisfying input as the key.

Obfuscating point functions is already well-understood. The goal is simply to give a clear explanation of how to use our formalism in practice. First, we show how to define security of a point function obfuscator using our formalism, and argue that this agrees with an existing notion in the literature. Second, we show a good point function obfuscator, using our definitions. Then we show how to disprove security of a bad point function obfuscator, using our definitions.

A. Defining Security of Point Function Obfuscation

The class of point functions is defined to be the set of program segments (or functions) that take an input x and output 1 (or execute a certain code segment) if and only if x is equal to some particular value c. The classic example is a licence check or password check.

The first step of putting this into our formalism is to define the class of programs.

Definition 4. *Let $\mathcal{C}_n$ be the set of programs that take an n-bit input x, and are zero on all except one input $c \in \{0,1\}^n$ (the password or the licence code). So, $P(x)$ is the predicate x $==$ c. We call $\mathcal{C}_n$ the class of point function programs.*

Each choice of $c \in \{0,1\}^n$ defines a different point function in $\mathcal{C}_n$, so $|\mathcal{C}_n| = 2^n$. The crucial fact that makes this problem interesting is that when n is large then an attacker cannot guess c with non-negligible probability.

The next step of putting this in our formalism is to define the program generator $\mathsf{Gen}_{\mathcal{C}}$.

Definition 5. *The program generator $\mathsf{Gen}_{\mathcal{C}}$ for point functions takes input n (technically speaking, the input should be written* *in unary as n ones), samples uniformly at random a binary string $c \in \{0,1\}^n$, sets $\mathsf{aux} = c$, and returns a program $P(x)$ given by*

```
if (x==c) then return 1 else return 0
```

Now, we define the assets in this program class. The asset in this case is the secret input c. So, we define $\mathcal{A}_n = \{0,1\}^n$ and $\mathsf{a}_n : \mathcal{C}_n \to \mathcal{A}_n$ so that $\mathsf{a}_n(P) = c$ is the specific value such that $P(\mathsf{a}_n(P)) = 1$.

The formalism also requires us to specify an asset verifier. To verify that an adversary has computed the asset of a program correctly, we simply define $\mathcal{V}(P, \mathsf{aux}, \mathsf{a}_n(P))$ to check that $\mathsf{aux} = \mathsf{a}_n(P)$. Alternatively, in this case, the auxiliary information is not required and we can verify using P alone by computing $P(\mathsf{a}_n(P)) \in \{0,1\}$; since P is computable in polynomial time, $\mathcal{V}$ is computable in polynomial time.

An *obfuscator for point functions* is a program Obf that takes as input a program P from the class $\mathcal{C}_n$ and outputs a program P' in some other class $\mathcal{C}'_n$. The class $\mathcal{C}'$ depends on the details of the obfuscator; we will see an example in the next subsection. As in Definition 3, we require P' to be correct (having the same functionality as P) and be efficient to execute.

The final part of our formalism is to define security of an obfuscator Obf. As we have seen, we consider a probabilistic polynomial-time algorithm A that attempts to computes the asset. The attacker A has full knowledge of $\mathsf{Gen}_{\mathcal{C}}, \mathsf{Obf}$ and $\mathcal{V}$, which is consistent with Kerckhoff's principle. The attacker wins if $\mathcal{V}(P, \mathsf{aux}, A(P')) = 1$. We say that the obfuscator is *insecure* if the probability A wins, over uniformly chosen $P \in \mathcal{C}_n$ and over the random choices of Obf and A, is noticeable (see Definition 3). Conversely, we say that the obfuscator is *secure* if, for all probabilistic polynomial-time algorithms A, the success probability p_n of A is negligible.

In 2014, Barak *et al.* [14] defined *input-hiding security*: It should be hard for an algorithm A, given P' for $P \in \mathcal{C}_n$, to find an $x \in \{0,1\}^n$, such that $P(x) = 1$. Therefore, the security definition above, defining the asset $\mathsf{a}_n(P) = c$, is equivalent to input-hiding security. This shows that our formalism is powerful enough to include existing security definitions from theoretical cryptography.

We also remark that a more general formulation of this problem, as "Constant Hiding" is given by Kuzurin *et al.* [6].

B. A Secure Obfuscator for Point Functions

Obfuscating point functions is well-known. It received its first theoretical treatment by Canetti [19]. The classic obfuscator Obf for point functions is to use a cryptographic hash function H with n-bit output. If the program $P \in \mathcal{C}_n$ is

```
if (x==c) then return 1 else return 0
```

then the obfuscated program $P'(x)$ includes constants r and $c' = H(r \| c)$ and checks whether or not $H(r \| x) = c'$.

In other words, Obf is a randomised algorithm that takes $P \in \mathcal{C}_n$ as input, determines c from examining the code or from the auxiliary input aux, chooses a random $r \in \{0,1\}^n$, sets $c' = H(r \| c)$ and outputs the program $P'(x)$ given by

```
if (H(r || x)==c') then return 1 else return 0
.
```

Lynn, Prabhakaran, and Sahai [20] prove security of obfuscating point functions in the random oracle model. Also see Hofheinz, Malone-Lee, and Stam [13], and Section 3 of Di Crescenzo [21]. In the full version of this paper [22] we also give a proof of security.

C. An Insecure Obfuscator for Point Functions

We now give an example of an insecure obfuscator for point functions, to illustrate how our formalism is falsifiable. This makes explicit what it means to break an obfuscator, and also aligns with the discussion in Section IV.

So, consider the following (obviously trivial) obfuscator Obf for point functions: The input $P(x) \in \mathcal{C}_n$ to Obf is a program

```
if (x==c) then return 1 else return 0 .
```

Then, Obf determines c from examining the code, chooses a random $r \in \{0,1\}^n$, sets $c' = r \oplus c$, and outputs the program $P'(x)$ defined by

```
if ((r^x)==c') then return 1 else return 0
.
```

To show that, this obfuscator is insecure we are required to define an attacker A. The attacker A takes as input the source code of P'. Remember that the algorithm A knows the class $\mathcal{C}_n$ and knows exactly what Obf is doing. Hence, A simply identifies the values r and c' from the code, computes $x = r \oplus c'$, and returns x. It follows that A succeeds with probability 1, since x is exactly the asset in the program.

VI. Real-World Examples

We now consider some classes of programs that are natural targets for obfuscation in real-world applications. Our goal is to show how the security of obfuscation in these cases can be expressed using our formalism. This will demonstrate how our formalism can be used to capture security goals in a wide variety of contexts. The aim of this section is not to give complete obfuscation solutions; however, we do give some sketches and references for how to solve these problems.

A. Biometric Matching

When a user provides a biometric reading (*e.g.,* fingerprint or facial image) then certain features are extracted and represented as a *template* t. When a user submits their biometric reading then there may be measurement errors. The process of determining if the biometric matches the template is called *fuzzy matching*.

Most biometric-based authentication systems provide some privacy of the template biometric data, by storing some kind of *tag* t' that is like an encrypted or obfuscated version of the template. When users submit their biometric reading then the process of determining if the biometric matches the template can be viewed as an *obfuscated fuzzy matching* program. There are a number of solutions in the literature [23].

We show how to express security in our formalism.

Definition 6. *For $n \in \mathbb{N}$, consider the set of all n-bit binary strings (biometric readings) and let T_n be the set of all possible templates t extracted from them. The set $\mathcal{C}_n$ of programs is parameterised by T_n. For each template $t \in T_n$, we have a program $P \in \mathcal{C}_n$ that takes as input an n-bit binary string x and outputs 1 if and only if the extracted template from x matches t within the given error tolerance.*

The algorithm $\mathsf{Gen}_{\mathcal{C}}$ selects uniformly at random an n-bit string and outputs this program P together with $\mathsf{aux} = t$. The asset $\mathsf{a}_n(P)$ is the template $t \in T_n$. So, T_n is the asset space.

An obfuscator Obf for fuzzy matching is secure if no attacker can efficiently compute the asset t when given an obfuscated program P'.

The common security model for secure biometric matching is actually information-theoretic: Given a tag t' that is to be matched by an input, there is still substantial entropy in the possible template t. Solutions to the biometric problem are often based on concepts like *secure sketches* and use coding-theoretic ideas. Instead, our formalism gives an alternative way to define security in the context of fuzzy matching, which is more amenable to solutions based on *computational security* than information-theoretic security. Obfuscation solutions in our model have been given by [24], [25].

B. White-Box Cryptography and Digital Rights Management (DRM)

White-box cryptography was introduced by Chow *et al.* [26], [27] for digital rights applications. DRM is an access control mechanism that aims at restricting access to digital contents. In the context of obfuscation, white-box cryptography and DRM are discussed in [28], [29]

The most common scenario in white-box cryptography is a decryption algorithm for a known symmetric cipher with key k. The idea is that one can provide a user with such a program, and the user cannot determine the key. This is one of the standard security models for white-box cryptography, and we now show how it can be expressed using our formalism.

Definition 7. *Let Enc and Dec be the encryption/decryption algorithms for a block cipher with variable key length. Define $\mathcal{C}_n$ as the set of programs $P = Dec_k$ for each possible n-bit key k.*

Choosing $P \in \mathcal{C}_n$ is simply choosing a secret key k. So, the algorithm $\mathsf{Gen}_{\mathcal{C}}(1^n)$ samples an n-bit key k and outputs $P = Dec_k$ and $\mathsf{aux} = k$. Define $\mathsf{a}(P) = k$ to be the secret key corresponding to that instance.

For a candidate asset a, the asset verifier $\mathcal{V}(P, \mathsf{aux}, a)$ just checks whether $a = \mathsf{aux}$.

An obfuscator Obf for white-box encryption is secure if no attacker can efficiently compute the key k when given an obfuscated decryption program.

An alternative verify algorithm $\mathcal{V}$ (which is probabilistic) is to check the correctness of the key by encrypting random messages using the candidate key and seeing if the white-box program decrypts to the correct message.

C. Finite Automata

A finite automaton (state machine) is a program to recognise whether or not an input x is a member of a language L. It has a set of states and, for each symbol in the alphabet, a "next state" transition matrix. One can imagine a scenario where one wants to compute the automaton but keep the transition matrix secret. A related application is regular expressions. The question of obfuscating automata is mentioned in Section 6 of Lynn, Prabhakaran and Sahai [20] and in Kuzurin *et al.* [6].

State machines are used in many applications, such as Cyber-Physical Systems (CPS) and control systems (*e.g.,* Supervisory Control And Data Acquisition – SCADA in short), to encode workflows and other processes in industrial and operational systems [30], [31]. These processes may be finely tuned and optimised using complex mathematical models, or as the result of years of experience. It may arise that these workflows need to be protected against tampering, or that we want to prevent leaking the workflow information to a competitor. Hence, an organisation may wish to prevent theft of the intellectual property represented in such a process, yet the state machine/process itself may be implemented in software throughout the organisation using embedded devices or Internet-of-Things (IoT)-enabled devices. Moctezuma *et al.* [30] discuss the role of state machines in CPS while Iarovyi *et al.* [31] describe the security concerns in this context.

We now explain how to express security using our formalism.

Definition 8. *Define $\mathcal{C}_n$ to be the set of state machines with n states and alphabet $\{0, 1\}$. Let state 1 be the initial state and state n be the accepting state.*

Each element $P \in \mathcal{C}_n$ recognises some language $L \subseteq \{0, 1\}^*$. One can generate a random program in $\mathcal{C}_n$ by choosing randomly a pair of transition matrices (M_0, M_1); hence, we have a program generator $\mathsf{Gen}_{\mathcal{C}}$ and the value aux is the pair (M_0, M_1).

A natural definition for the asset $\mathsf{a}(P)$ is the pair of transition matrices, but note that this depends on the exact ordering of the states; whereas, we might consider two state machines to be isomorphic if each transition matrix of one of them is the conjugate by a permutation matrix of the corresponding transition matrix of the other. Even more problematic is the fact that there may be many quite different state machines that recognise the same language. Hence, we need to be more flexible in our definition of the asset.

Our formalism is designed to handle such issues. In this case, $\mathsf{a}(P)$ is *any* pair of transition matrices for an automaton that recognises the same language. We then require the asset verifier to determine if two state machines recognise the same language. Fortunately, it is known that one can efficiently test equivalence of automata/regular languages (see Hopcroft and Karp [32] or Almeida *et al.* [33]).

Definition 9. *Let notation be as in Definition 8. Then, $\mathsf{a}(P)$ is any transition matrix pair for an automaton that recognises the same language as P. The asset space for $\mathcal{C}_n$ is the set of all pairs of well-formed $n \times n$ transition matrices. For a candidate asset a, the asset verifier $\mathcal{V}(P, \mathsf{aux}, a)$ is an efficient algorithm to determine if the two automata (given by transition matrix pairs aux and a respectively) recognise the same language*

Finally, to ensure that security is possible, we need to consider whether a finite state machine is learnable from its input/output behaviour. Angluin [34] shows that one can learn an automaton when given a way to generate random inputs that are accepted and also random inputs that are not accepted. However, if it is hard to efficiently find such inputs from "black box" execution of the machine, then it seems to be hard to determine the automaton. Hence, the obfuscation problem makes sense for sufficiently complex finite state machines for which either the language or its complement is small.

We therefore need to restrict to the subclass of $\mathcal{C}_n$ of evasive automata. One then can consider an obfuscator Obf that takes as input a transition matrix pair for an automaton P and outputs an obfuscated program P' that takes $x \in \{0,1\}^*$ as input and returns 1 if and only if x is in the language of P. The obfuscator is secure if no adversary takes P' as input, and outputs a transition matrix pair for an automaton P that recognises the same language. Galbraith and Zobernig [35] have proposed an obfuscation scheme for automata that satisfies this definition.

Definition 3 of Kuzurin *et al.* [6] introduces the notion of a "software protecting obfuscator", which is similar in flavour to iO, and proposes this notion as suitable for the problem of obfuscation DFA (Deterministic Finite State Automata). We believe circuit-hiding and/or input-hiding are more natural security definition for this problem.

VII. Composition of Obfuscations

In this section, we discuss the security of applying different obfuscators iteratively on a program, to protect multiple assets of different types. We call this *composition* of obfuscators. This concept has been discussed by researchers in the software security world and seems to be commonly used. For example, Section 2.1 of Schrittwieser *et al.* [12] note that "many commercial obfuscators employ (and indeed recommend to use) multiple obfuscations at the same time". Xu *et al.* [36] present a variant of composition called "layered obfuscation". But composition of obfuscators has never had a rigorous theoretical treatment.[2]

Be aware that the word "composition" already has several meanings in the context of obfuscation. For example, Lynn, Prabhakaran, and Sahai [20] use the phrase "compositions of obfuscations" to mean applying the same obfuscator Obf to a sequence $P_1, \ldots, P_t$ of programs to get $(\mathsf{Obf}(P_1), \ldots, \mathsf{Obf}(P_t))$. They write "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". Hofheinz, Malone-Lee, and Stam [13] also discuss composability in the context of point function obfuscation: "security may be lost when the obfuscation is used in larger settings in which x is used in several places".

We stress that our interest is in a class of programs $\mathcal{C}_n$ that may have several different assets, and where obfuscators Obf_1 and Obf_2 are each designed to protect a different asset in a different way. We wish to have an assurance that the composition $\mathsf{Obf}_2(\mathsf{Obf}_1(P))$ protects both assets (a formal description is given in Theorem 1).

The following theorem gives conditions under which composition of obfuscators makes sense and maintains security.

Theorem 1. *Let $\mathcal{C}_n$ be a class of program segments defined by a generator $\mathsf{Gen}_\mathcal{C}$. Let $\mathsf{a}^{(1)} : \mathcal{C}_n \to \mathcal{P}(\mathcal{A}_n^{(1)})$ and $\mathsf{a}^{(2)} : \mathcal{C}_n \to \mathcal{P}(\mathcal{A}_n^{(2)})$ be assets with verifiers $\mathcal{V}^{(1)}$ and $\mathcal{V}^{(2)}$. Suppose we have a secure obfuscator $\mathsf{Obf}^{(1)}$ that protects $\mathsf{a}^{(1)}$ and a secure obfuscator $\mathsf{Obf}^{(2)}$ that protects $\mathsf{a}^{(2)}$. Suppose further that the following conditions hold.*

1) *$\mathsf{Obf}^{(1)} : \mathcal{C}_n \to \mathcal{C}_n$ and $\mathsf{Obf}^{(2)} : \mathcal{C}_n \to \mathcal{C}_n$.*
2) *The function $\mathsf{a}^{(2)}$ is defined on $\mathsf{Obf}^{(1)}(P, \mathsf{aux})$ and the function $\mathsf{a}^{(1)}$ is defined on $\mathsf{Obf}^{(2)}(P, \mathsf{aux})$ for $P \in \mathcal{C}_n$. (This is essentially implied by (1) and the definition of an asset.)*
3) *Over $P \in \mathcal{C}_n$ generated by $(P, \mathsf{aux}) \leftarrow \mathsf{Gen}_\mathcal{C}(1^n)$, the output distribution of programs $P' \leftarrow \mathsf{Obf}^{(1)}(\mathsf{Obf}^{(2)}(P, \mathsf{aux}), \mathsf{aux})$ is close to the output distribution of $\mathsf{Obf}^{(2)}(\mathsf{Obf}^{(1)}(P, \mathsf{aux}), \mathsf{aux})$.*

Then, there is no adversary that can compute either $\mathsf{a}^{(1)}(P)$ or $\mathsf{a}^{(2)}(P)$ from $\mathsf{Obf}^{(2)}(\mathsf{Obf}^{(1)}(P, \mathsf{aux}), \mathsf{aux})$.

Proof. Let A be an adversary that takes as input $\mathsf{Obf}^{(2)}(\mathsf{Obf}^{(1)}(P, \mathsf{aux}), \mathsf{aux})$ and outputs a candidate a for $\mathsf{a}^{(1)}(P)$ or $\mathsf{a}^{(2)}(P)$. Suppose A succeeds with non-negligible probability (over the probability distribution of $(P, \mathsf{aux}) \leftarrow \mathsf{Gen}_\mathcal{C}(1^n)$ and the random choices by the obfuscators).

First, consider the case where A computes $\mathsf{a}^{(1)}(P)$. Then, we turn A into an adversary A' that computes $\mathsf{a}^{(1)}(P)$ from $\mathsf{Obf}^{(1)}(P, \mathsf{aux})$, thus violating the claim that $\mathsf{Obf}^{(1)}$ is a secure obfuscator. The construction is quite simple: Given a challenge $P' = \mathsf{Obf}^{(1)}(P, \mathsf{aux})$ the new adversary A' simply applies $\mathsf{Obf}^{(2)}$ to P' and then passes this to A, which computes $\mathsf{a}^{(1)}(P)$ with non-negligible probability.

Next, consider the case where A computes $\mathsf{a}^{(2)}(P)$. We turn A into an adversary A'' that computes $\mathsf{a}^{(2)}(P)$ from $P'' = \mathsf{Obf}^{(2)}(P, \mathsf{aux})$. The main idea is to use property (3). We apply $\mathsf{Obf}^{(1)}$ to P'', to get a program P'''. Now if the distribution of $P''' = \mathsf{Obf}^{(1)}(\mathsf{Obf}^{(2)}(P, \mathsf{aux}), \mathsf{aux})$ has small statistical distance from the distribution of $\mathsf{Obf}^{(2)}(\mathsf{Obf}^{(1)}(P, \mathsf{aux}), \mathsf{aux})$ then the algorithm A also succeeds on P''' with non-negligible probability. Once again, A'' simply returns the value computed

[2]Note that the topic does not arise in the theoretical obfuscation community, since VBB already protects everything in a program so there is no need to compose VBB obfuscators.

by A on this obfuscated program. Since A computes $\mathsf{a}^{(2)}(P)$ with non-negligible probability, it follows that A'' also succeeds with non-negligible probability. $\square$

VIII. Conclusion

We have introduced a new formalism that allows to precisely define security of obfuscation and to rigorously prove the security of an obfuscator. We believe our formalism will be useful for the design (and cryptanalysis) of obfuscation tools.

Central to our formalism is the notion of an asset. Our contributions include clarifying the need for a precise definition of program class (via a program generator) and the requirement to have an asset verifier. We have shown that these requirements are exactly the same requirements for setting up an automated obfuscation challenge.

In the full version of this paper [22] we discuss some limitations of our formalism.

Acknowledgment

We thank Christian Collberg for several suggestions. This research is funded in part by the Royal Society of New Zealand Marsden fund project 16-UOA-144.

References

[1] O. Goldreich and R. Ostrovsky, "Software protection and simulation on Oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[2] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Computer Science Technical Reports 148, 1997.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *CRYPTO*, ser. LNCS, J. Kilian, Ed., vol. 2139. Springer, 2001, pp. 1–18.

[4] ——, "On the (im)possibility of obfuscating programs," *J. ACM*, vol. 59, no. 2, pp. 6:1–6:48, 2012.

[5] C. S. Collberg, "Code obfuscation: Why is this still a thing?" in *CODASPY*, Z. Zhao, G. Ahn, R. Krishnan, and G. Ghinita, Eds. ACM, 2018, pp. 173–174.

[6] N. Kuzurin, A. V. Shokurov, N. P. Varnovsky, and V. A. Zakharov, "On the concept of software obfuscation in computer security," in *ISC*, ser. LNCS, J. A. Garay, A. K. Lenstra, M. Mambo, and R. Peralta, Eds., vol. 4779. Springer, 2007, pp. 281–298.

[7] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "On secure and usable program obfuscation: A survey," CoRR abs/1710.01139, 2017.

[8] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel, "Program obfuscation: A quantitative approach," in *QoP*, G. Karjoth and K. Stølen, Eds. ACM, 2007, pp. 15–20.

[9] C. Basile, S. D. Carlo, T. Herlea, J. Nagra, and B. Wyseur, "Towards a formal model for software tamper resistance," Technical Report, COSIC, 2009.

[10] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[11] M. Ahmadvand, A. Pretschner, and F. Kelbert, "A taxonomy of software integrity protection techniques," *Advances in Computers*, vol. 112, pp. 413–486, 2019.

[12] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. R. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys*, vol. 49, no. 1, pp. 4:1–4:40, 2016.

[13] D. Hofheinz, J. Malone-Lee, and M. Stam, "Obfuscation for cryptographic purposes," in *TCC*, ser. LNCS, S. P. Vadhan, Ed., vol. 4392. Springer, 2007, pp. 214–232.

[14] B. Barak, N. Bitansky, R. Canetti, Y. T. Kalai, O. Paneth, and A. Sahai, "Obfuscation for evasive functions," in *TCC*, ser. LNCS, Y. Lindell, Ed., vol. 8349. Springer, 2014, pp. 26–51.

[15] C. Collberg, "The tigress c diversifier/obfuscator," http://tigress.cs.arizona.edu/challenges.html, 2016.

[16] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *ACSAC*. ACM, 2016, pp. 189–200.

[17] S. Banescu, C. S. Collberg, and A. Pretschner, "Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning," in *USENIX Security*, E. Kirda and T. Ristenpart, Eds., 2017, pp. 661–678.

[18] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *NDSS*. The Internet Society, 2008.

[19] R. Canetti, "Towards realizing random oracles: Hash functions that hide all partial information," in *CRYPTO*, ser. LNCS, B. S. Kaliski Jr., Ed., vol. 1294. Springer, 1997, pp. 455–469.

[20] B. Lynn, M. Prabhakaran, and A. Sahai, "Positive results and techniques for obfuscation," in *EUROCRYPT*, ser. LNCS, C. Cachin and J. Camenisch, Eds., vol. 3027. Springer, 2004, pp. 20–39.

[21] G. D. Crescenzo, "Cryptographic program obfuscation: Practical solutions and application-driven models," in *Versatile Cybersecurity*, ser. Advances in Information Security, P. R. Conti M., Somani G., Ed., vol. 72. Springer, 2018.

[22] M. R. Asghar, S. D. Galbraith, A. Lanzi, . G. Russello, and L. Zobernig, "Towards a theory of special-purpose program obfuscation," arXiv:2011.02607, 2020.

[23] P. Tuyls, A. H. M. Akkermans, T. A. M. Kevenaar, G. J. Schrijen, A. M. Bazen, and R. N. J. Veldhuis, "Practical biometric authentication with template protection," in *AVBPA*, ser. LNCS, T. Kanade, A. K. Jain, and N. K. Ratha, Eds., vol. 3546. Springer, 2005, pp. 436–446.

[24] K. Karabina and O. Canpolat, "A new cryptographic primitive for noise tolerant template security," *Pattern Recognition Letters*, vol. 80, pp. 70–75, 2016.

[25] S. D. Galbraith and L. Zobernig, "Obfuscating fuzzy matching for hamming distance," in *TCC*, ser. LNCS, D. Hofheinz and A. Rosen, Eds., vol. 11891. Springer, 2019, pp. 81–110.

[26] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot, "White-box cryptography and an AES implementation," in *Selected Areas in Cryptography 2002*, ser. LNCS, K. Nyberg and H. M. Heys, Eds., vol. 2595. Springer, 2003, pp. 250–270.

[27] ——, "A white-box DES implementation for DRM applications," in *CCS*, ser. LNCS, J. Feigenbaum, Ed., vol. 2696. Springer, 2003, pp. 1–15.

[28] M. Joye, "On white-box cryptography," in *Security of Information and Networks*, A. Elçi, S. Ors, and B. Preneel, Eds. Trafford Publishing, 2008, pp. 7–12.

[29] A. Sethi, "Digital rights management and code obfuscation," MS thesis, University of Waterloo, 2004. [Online]. Available: http://hdl.handle.net/10012/1012

[30] L. E. G. Moctezuma, B. R. Ferrer, X. Xu, A. Lobov, and J. L. M. Lastra, "Knowledge-driven finite-state machines. study case in monitoring industrial equipment," in *INDIN*. IEEE, 2015, pp. 1056–1062.

[31] S. Iarovyi, W. M. Mohammed, A. Lobov, B. R. Ferrer, and J. L. M. Lastra, "Cyber-physical systems for open-knowledge-driven manufacturing execution systems," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 1142–1154, 2016.

[32] J. E. Hopcroft and R. M. Karp, "A linear algorithm for testing equivalence of finite automata," Cornell University, Tech. Rep., 1971.

[33] M. Almeida, N. Moreira, and R. Reis, "Testing the equivalence of regular languages," *Journal of Automata, Languages and Combinatorics*, vol. 15, no. 1–2, pp. 7–25, 2010.

[34] D. Angluin, "Learning regaular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, 1987.

[35] S. D. Galbraith and L. Zobernig, "Obfuscating finite automata," in *to appear in Selected Areas in Cryptology (SAC 2020)*. Springer, 2020.

[36] H. Xu, Y. Zhou, J. Ming, and M. Lyu, "Layered obfuscation: a taxonomy of software obfuscation techniques for layered security," *Cybersecurity*, vol. 3, no. 9, 2020.