Notes for Lecture 1

1 Introduction to Obfuscation

Program Obfuscation promises the seemingly impossible: software that can keep a secret — even from the person who owns and is running the software! The "secret" can come in many forms: a new proprietary algorithm, the secret key to unlock and encrypted message, or the password to you bank account. The goal is to distribute software — with the secret embedded in it — all the while guaranteeing that the recipient of the software never learns the secret.

More precisely, a program obfuscator is a program *compiler*: it takes a program P as input, and outputs a program $\hat{P}$. As with any compiler, the input and output programs must be functionally equivalent: $\hat{P}(x) = P(x)$ for all inputs x. An obfuscator is a special kind of compiler where the output program $\hat{P}$ reveals "as little as possible" about the program P. Clearly, if someone has $\hat{P}$, they can learn $P(x)$ for any chosen input simply by computing $\hat{P}(x)$. In other words, the functionality of P is always revealed by the obfuscated program $\hat{P}$. Ideally, this is the only information that can be deciphered. The inner workings of P — that is, *how* the output $P(x)$ was obtained — should be entirely hidden. Any algorithms, cryptographic keys, or passwords used in P should not be revealed.

Another way to think about obfuscation is as a form of encryption for programs. One can always encrypt a program using a standard encryption techniques by encrypting the 1s and 0s that constitute the code for the program. To anyone without the secret decryption key, this encrypted program will look like just a random sequence of 1s and 0s. Hence, any secrets embedded in the program will remain hidden, as desired. However, this encrypted program is also completely useless as a program: the only way to evaluate it is to decrypt (requiring the secret decryption key), and then run the program. But with the secret decryption key, the entire original program, secrets included, are revealed. Thus, with traditional encryption, security and functionality are at odds. Obfuscation gives a way to resolve this conflict by allowing the program to be encrypted (so that internal secrets are hidden), but simultaneously also allowing the encrypted program to be evaluated.

1.1 Motivation

Why study program obfuscation? Here are some potential applications of obfuscation:

Preventing Reverse-Engineering. Suppose you devise an efficient algorithm for multiplying matrices in time $O(n^2)$. You wish to monetize your discovery by selling software packages including your algorithm. To make the most money, you want to offer packages at different price pints with different running times — say one package multiplies matrices in time $O(n^2)$, another in time $O(n^{2.1})$, and so forth. To accomplish this, you have your program churn mindlessly for $O(n^{2+\epsilon})$ steps, and then finally compute the matrices using your fast algorithm.

Unfortunately, someone can buy the cheapest version of your software — say, an $O(n^{2.5})$ version — inspect the code, and remove the part where the algorithm churns mindlessly. In the end, this cheapskate obtains an $O(n^2)$ algorithm, but at a fraction of the price. Obfuscation offers a very simple solution to this problem. Once the $O(n^{2.5})$ algorithm is obfuscated, the cheapskate has no hope of extracting the secret $O(n^2)$ algorithm from the code.

Watermarking Software. You have a program that you wish to sell. However, you are concerned about someone stealing your software and reselling it. Of course, there is no way to prevent someone who actually bought your software from doing this. Instead, you wish to discourage such behavior. Toward that goal, you individualize your program to each customer, embedding some unique identifier into the functionality of their program. You then obfuscate the program to make removing the watermark difficult.

Cryptomania. Obfuscation has become a "central hub" for cryptography, in the sense that almost any cryptographic task imaginable can be accomplished through obfuscation.

For example, consider a setting where Alice and Bob wish to communicate over an insecure channel which may have eavesdroppers. In order to hide their messages, they encrypt every message. In the "traditional" setting, Alice and Bob have a shared secret key k which they established at a prior point in time (say by meeting in person, or by a briefcase handcuffed to a trusted courier). When Alice wants to send a message m to Bob, she "encrypts" m with k, and sends the resulting ciphertext c over the insecure channel. Bob, upon receiving c, "decrypts" using k to obtain the message m. This form of encryption is called "secret key encryption" or "symmetric key encryption" because the same key is used for encrypting and decrypting. This is the kind of encryption that has been used throughout most of history.

However, symmetric key encryption is rather impractical for our current digital age. People may want to communicate securely, even if they have never met in person and do not want to go through the hassle of a trusted courier. The solution is "public key encryption" or "asymmetric key encryption". Here, Alice as a secret key dk that she alone uses to decrypt. To allow Bob to send messages to her, Alice publishes

an encryption key ek to *everyone*, which Bob can use to encrypt m as a ciphertext c. This broadcast gives ek to eavesdroppers as well; even so, an eavesdropper armed with ek (but not dk) should not be able to decrypt c.

How do we achieve public key encryption? With obfuscation, the answer is simple: start with a symmetric key encryption scheme. Alice's secret key sk is just a secret key k for the scheme. Meanwhile, the public key ek is the obfuscation of the encryption program, with k hardcoded. This program allows anyone to encrypt (by just running the program on the message), but because it is obfuscated, the secret key remains hidden. Thus, ek does not give you the ability to decrypt.

The usefulness of obfuscation does not end there. Consider the setting where you wish to store data with a cloud provider. You do not trust the cloud provider with your data, so you encrypt it. Now, you additionally wish to take advantage of the providers computational resources to run computations on your data. Unfortunately, with traditional encryption, the ciphertexts look like a jumble of random 1s and 0s — there is no obvious way to perform computations on this data. Of course, you can decrypt your data for the cloud, but then the cloud can abuse your data.

One solution is something called fully homomorphic encryption (FHE). In such an encryption scheme, it is possible to add and multiply plaintexts, just given the ciphertexts. That is, there is an operation $\oplus$ such that $\mathsf{Enc}(k,x) \oplus \mathsf{Enc}(k,y) = \mathsf{Enc}(k, x+y)$ and an operation $\otimes$ such that $\mathsf{Enc}(k,x) \otimes \mathsf{Enc}(k,y) = \mathsf{Enc}(k, x \times y)$. FHE easily allows the cloud provider to blindly operate on encrypted data by substituting $+$ with $\oplus$ and $\times$ with $\otimes$.

Here's how to obtain FHE from obfuscation. $\oplus$ is an obfuscation of the program that decrypts the input ciphertexts, adds them, and re-encrypts them. $\otimes$ is an obfuscation of the analogous program for multiplication. These programs allow for the computaitons on encrypted data, but hide the secret key.

It turns out that we also have "direct" ways of obtaining public key encryption and fully homomorphic encryption from simple tools that do not go through obfuscation. These examples are still interesting because they demonstrate the intuitive power and broad usefulness of obfuscation in cryptography. Moreover, these two applications are only the start — for many of the applications of obfuscation, the obfuscation-based approach is the *only* approach we currently know.

Broader CS Theory Implications. Obfuscation also has connections to broader CS theory concepts. For example, certain forms of obfuscation imply the hardness of finding Nash equilibria.

1.2 Difficulties

Obfuscation is a very challenging task. Indeed, someone who posses the program has many ways to try to extract information. They can run the program on inputs of their choice, and get to see all intermediate states of the program in addition to the program output. They can even inject faults — changing the internal state — and observing how that affects program behavior. They can do all of this, and yet still should not be able to learn anything about the implementation details of the program. Nonetheless, in this course we will see constructions of obfuscators that are plausible secure against adversaries with even this level of access to the program.

2 Defining Obfuscation

Importance of definitions in cryptography. It will be important for our study of obfuscation to give precise and rigorous definitions for what an obfuscator actually is. Such definitions give us a consistent language in which to study obfuscation. Without definitions, it will be impossible to evaluate how well an obufscator obfuscates and compare different obfuscators. Thus, definitions are crucial to measure our progress on building obfuscation. Since we are also interested in the theoretical implications of obfuscation, definitions are imperative since we cannot prove a theorem about concepts that are not rigorously defined. This need for definitions is by no means unique to obfuscation: indeed, it is now the standard practice in cryptography to give formal definitions, for exactly the reasons above.

Additional Considerations for Obfuscation. For obfuscation, there is perhaps even more need for formal definitions. For example, consider the following. From undergraduate CS theory courses, we know it is possible to take a program P, and construct a program P' that outputs its own description. That is, $P'(x) = (P(x), P')$[1]. What happens when we obfuscate P'? We get a program $\hat{P}'$ that is functionally equivalent to P'. By running $\hat{P}'$ on *any* input, one obtains P'. Hence, the obfuscation can easily be reversed.

More generally, consider any program P that is learn-able: given blackbox access to the functionality of P, it is possible to recover P. Then any obfuscation of P, by virtue of being functionally equivalent of P, can be "undone" to recover P. Therefore, informal guarantees like "obfuscation hides the original program" cannot possibly be true in general. We instead need a notion that permits some programs to be reverse-engineered. Moreover, we need the notion to be completely formal in order to verify that there are not any edge cases that make the notion impossible.

[1]Here, we are using P' as both a function (on the left-hand side) and data (on the right-hand side). This will be a common occurrence the course

2.1 Formal Definitions of Obfuscation

What properties should an obfuscator satisfy? Here are the obvious ones:

Functionality. The output of the obfuscator should clearly be a program, and the program should have the same input/output behavior as the input program.

Efficiency of the obfuscator. The obfuscating algorithm should be "efficient." There's no use for an obfuscator that takes a billion years to run. In this course, we will take the "easy" approach and define an obfuscator as "efficient" if its running time is polynomial in the size of the input (that is, the size of the input program being obfuscated).

Efficiency of the obfuscated program. In addition to the obfuscating algorithm being efficient, we ask that the obfuscated code is also efficient. Again, there's no use of an obfuscator that produces programs that take a billion years to run. Again, we will take the "easy" approach, and say the obfuscated code is "efficient" if its running time is polynomially related to the running time of the original program. So there is some polynomial poly such that, if the original program runs in time t on some input, the obfuscated program runs in time $\mathsf{poly}(t)$.

Now we move on to the more difficult aspects of defining obfuscation. In some of the following, we will actually have some choices to make.

Model of computation. We need to formally define the model of computation we will be working with. This is necessary for making sense of the efficiency requirements. Moreover, as we will see in this course, certain notions of obfuscation may be achievable in some models of computation, but not others. There are two natural choices for computational models: **Turing Machines** and **Circuits**. Turing Machine obfuscation has the advantage of capturing programs that run on arbitrary-length inputs, whereas circuits have a fixed input size. However, as we will see when we start to construct obfuscation, circuits are much easier to work with.

We will also discuss obfuscation for restricted models of computation. For example, when building obfuscation, our starting point will be to obfuscate computations that take the form of a "matrix branching program". Matrix branching programs capture programs computable by log-depth circuits.

Security. So far, the identity function satisfies all the requirements of an obfuscator. What's missing is a notion of security. It turns out that defining security for obfuscation is tricky, as hinted at above. After all, what precisely does it mean to

prevent reverse engineering of a program? If the program outputs a description of itself, clearly it can be reverse engineered with zero effort. How should an obfuscator behave in this case? Here are some potential definitions of security:

- **Virtual Black Box Obfuscation (VBBO).** This notion, informally, says that anything learnable from the obfuscated code can be learned just from black box access to the functionality — in other words, there is no point in looking at the code, and instead you might as well just run the code on some inputs. More formally,

  Definition 1 (Virtual Black Box security for Obfuscation) *An obfuscator* Obf *is virtual black box secure if the following holds. For any probabilistic polynomial time (PPT) adversary A, there exists a "simulator" S and a negligible function* negl *such that for any program (Turing Machine/Circuit) P*

  $$\left|\Pr[A(\mathsf{Obf}(P)) = 1] - \Pr[S^P(|P|) = 1]\right| < \mathsf{negl}(|P|)$$

  Here, S is probabilistic and runs in time polynomial in $|P|$. S^P means that S can make oracle queries to the functionality of P. Each query takes unit time. For Turing machines, we also use the convention that, along with the intended output, the program also outputs its running time.

  We give the simulator $|P|$ since the length of the program cannot be entirely hidden by the obfuscation.

  Remark 2 *Notice that S actually runs in time exponential in the bit-length of its input, since the bit-length of $|P|$ is $\log |P|$. Many authors will actually represent $|P|$ in unary, giving S the input $1^{|P|}$. In this case, S is actually a polynomial time algorithm.*

  Remark 3 *One could ask if restricting to polynomial time adversaries and simulators is necessary. Indeed, this has been the study of some works. However, security against "efficient" adversaries is sufficient, and already a very difficult goal. Moreover, as we will see later in the course, security against unbounded adversaries is impossible under widely believed complexity assumptions. Therefore, for the bulk of this course, we will only consider security against polynomial-time adversaries.*

  One limitation of this definition is that security depends on having large programs. What if the program is constant size? To enable a meaningful notion in this case, we do the following. First, the obfuscator Obf takes as input both a

program P, and a "security parameter" λ. Increasing λ gives a "more secure" obfuscation of the program. We will assume that it is possible to recover λ from the obfuscated program, so that everyone knows what level of security was used to obfuscate. The running time of Obf, and the running time of the obfuscated program, are allowed to grow polynomially with λ.

Definition 4 (VBB security for Obfuscation, with Security Parameter) *An obfuscator* Obf *is virtual black box secure if the following holds. For any probabilistic polynomial time (PPT) adversary A, there exists a "simulator" S and a negligible function* negl *such that for any program (Turing Machine/Circuit) P*

$$\left|\Pr[A(\mathsf{Obf}(P,\lambda))=1]-\Pr[S^P(|P|,\lambda)=1]\right|<\mathsf{negl}(\lambda)$$

Here, S is probabilistic and runs in time polynomial in $|P|$ and λ

Remark 5 *Notice that* Obf *and S run in time exponential in the bit-length of λ. Many authors will give λ to the algorithms in unary so that all algorithms run in polynomial time.*

As we will see next time, VBB obfuscation is in a sense "too strong." This is because there are actually programs where having access to the code of the program (even if obfuscated) actually does give you more power that back-box access. For this reason, we will need alternative definitions of security.

- **Indistinguishability Obfuscation (iO).** This is perhaps the most natural alternative to VBB obfuscation. It roughly says that, if I have two programs P_1 and P_2 that are equivalent ($P_1(x) = P_2(x)$ for all x), and if I give you an obfuscation of one of them, then you cannot tell which program was obfuscated. We will take the approach above of separating out the security parameter from the size of the obfuscated program.

Definition 6 (Indistinguishability Security for Obfuscation) *An obfuscator* Obf *is indistinguishability secure if the following holds. For any two programs (Turing Machines/Circuits) P_0, P_1 that are equivalent ($P_0(x) = P_1(x) \forall x$) and have the same size, for any PPT adversary A, there exists a negligible function* negl *such that*

$$|\Pr[A(\mathsf{Obf}(P_0,\lambda))=1]-\Pr[A(\mathsf{Obf}(P_1,\lambda))=1]|<\mathsf{negl}(\lambda)$$

Taking a look at the definition of iO, we unfortunately seem to be moving away from the intuition that obfuscation prevents reverse engineering. Indeed, iO is only meaningful if you can give a second program with the exact same

functionality. What if all programs that computed the functionality leaked the secret you are trying to hide?

As we will see in a couple of lectures, nonetheless it is still possible to use iO to solve almost any cryptographic task imaginable. And, perhaps just as surprisingly, we will see constructions of obfuscation that plausibly satisfy this notion. This is one of the reasons that iO has become the "industry accepted" security definition for obfuscation.