Notes for Lecture 18

1 Obfuscation

At an informal level, let us suppose that P is a program that takes some input, and provides some output, and uses some algorithm (or keys) that we don't want to fall in the wrong hands (either for security or commercial purposes). The idea is to *compile* P into a program $\hat{P}$ that preserves functionality but has certain security properties.

$$\text{Program P} \implies \text{new program } \hat{P}$$

$$\text{Goal: } \hat{P} \text{ does same thing as } P$$

We want the code for $\hat{P}$ to hide as much as possible about our program P.

2 Potential Applications

1. **Software Protection** — This has been the motivation to study obfuscation for a long time. Obfuscaton can used to prevent reverse engineering code, preventing IP theft, and enforcing Digital Rights Management (DRM). It can be used to protect software against unwanted use, if you have a "fancy new algorithm" you can obfuscate it to prevent it from getting into anyone's hands.

2. **Fully Homomorphic Encryption** — Use a plain PKE scheme to generate $(\mathsf{pk}, \mathsf{sk})$. Publish the obfuscated code $\hat{P}$ of P defined by

   $$P(c_1, c_2, \odot) = \mathsf{Enc}(\mathsf{pk}, \mathsf{Dec}(\mathsf{sk}, c_1) \odot \mathsf{Dec}(\mathsf{sk}, c_2))$$

   Note that the above should be randomized for security. The hope is that $\hat{P}$ hides sk so that by publishing, the semantic or CPA security is not violated. This has tremendous applications on crypto at large. Most problems in crypto have either a formal or an intuitive solution with obfuscation.

3 History

The study of obfuscaton started in the 1960s. Various tricks can be employed to achieve obfuscation.

1. Renaming variables
2. Unrolling for loops
3. Swapping order of operations
4. Other natural program alterations

Commercial obfuscators today are capable of doing these things super fast. But the problem is that these transformations can be undone with sufficient effort. In particular, commericla obfuscation cannot hide crypto keys.

The rigorous formal study of obfuscation started in 2001, with the celebrated work of Barak-Goldreich-Impagliazzo-Rudich-Sahai-Vardan-Yang. But there still existed no construction for crypto purposes. The first candidate construction came in 2013 with the work of Garg-Gentry-Holevi-Sahai-Raykova-Waters. This opened up the floodgates and since then there have been many more constructions based on different tools as well as many more applications. This has been one of the big advances in cryptography in the last decade — defining a definition of obfuscation that is useful for crypto.

$$\text{Obfuscation} \implies \text{Most of crypto}$$

We now know how to solve nearly anything in crypto with obfuscation. However, the best obfuscators are **incredibly** inefficient. The constants in these algorithms are galactic scale. To obfuscate even the simplest programs, if we pooled all the resources on Earth, we would still not be able to implement one of these obfuscator algorithms.

Current research focuses on realizing applications more efficiently or under better tools. Also, application of obfuscation algorithms to software protection is uncertain.

4 Defining Obfuscation

4.1 Dream Obfuscator

Our ideal obfuscator works as follows.

1. We can run the obfuscated program $\hat{P}$ to get output.

2. Given $\hat{P}$ all we can do is query $\hat{P}$ on various inputs

From this comes the idea of Virtual Black Box Obfuscation. In order to formalize the notion of 'learn nothing', we are going to make use of simulators. Our obfuscator program will be take as inputs the program P and the length of inputs to P and output $\hat{P}$.

$$\mathsf{Obf}(P, 1^\lambda) \to \hat{P}$$

It is also worthwhile to talk about the model of computation that we use when defining P. Here, we will mainly discuss two models - Turing Machines and Circuits.

We want our obfuscator to have the following properties.

1. Obf runs in polynomial time.
2. $\hat{P}$ is at most polynomially slower than P.
3. $\hat{P} \equiv P$ i.e. $\hat{P}(x) = P(x) \forall x$

We also want the security property

$$\text{Looking at code} \equiv \text{Not looking at code}$$

We will formalize the security of Virtual Black Box Obfuscation below.

4.2 Virtual Black Box Obfuscation

We want our Virtual Black Box (VBB) obfuscator to have the following security property.

Security. $\forall$ PPT $\mathcal{A}, \exists$ PPT S *such that for all programs* P,

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

Turns out however, this notion of security is not achievable!

Theorem 1 *VBB Obfuscation is impossible.*

5 Impoosibility of VBB obfuscation

5.1 Starting Observation

Let $P(x) = \mathsf{PRF}(k, x)$ for a chosen key k. Let $\hat{P} = \mathsf{Obf}(P, 1^\lambda)$, then $\hat{P}$ is a short description of the truth table of $\mathsf{PRF}(k, \cdot)$. In other words, $\hat{P}$ is a short program computing $\mathsf{PRF}(k, x)$.

However, given the nature of PRF, given oracle access to $\mathsf{PRF}(k, \cdot)$, it is impossible to produce short desscriptionn or program that comutes the PRF, since the PRF is a random-looking function which admits no short description.

This hints that in general, the security of VBB should be impossible, but doesn't contradict VBB since the simulator S only needs to simulate 1 bit.

5.2 Turing Machines

We will first prove impossibility of VBB on Turing Machines.

Idea — *Run $\hat{P}$ on itself.*

To start off, we will define a pair of programs that cannot be obfuscated jointly. So define

$$\underset{\alpha,\beta\in\{0,1\}^\lambda}{\mathrm{C}_{\alpha\beta}}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$$

$$\underset{\gamma\in\{0,1\}}{\mathrm{D}_{\alpha\beta\gamma}}(C) = \begin{cases} \gamma & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

There are some finer points here to handle, such as whether C halts or not. But we will sweep these details under the rug.

Informal Claim. $\mathrm{C}_{\alpha\beta}, \mathrm{D}_{\alpha\beta\gamma}$ *cannot simultaneously be obfuscated.*

We argue this as follows. Suppose $\hat{C}$ and $\hat{D}$ are the obfuscations of $\mathrm{C}_{\alpha\beta}$ and $\mathrm{D}_{\alpha\beta\gamma}$ respectively.

Suppose $\mathcal{A}$ has been given these obfuscations. It can then compute γ by running $\hat{C}$ on $\hat{D}$

$$\mathcal{A}(\hat{C}, \hat{D}) = \hat{D}(\hat{C}) = \gamma$$

However, a simulator $S^{C,D}$ which has oracle access to $\mathrm{C}_{\alpha\beta}$ and $\mathrm{D}_{\alpha\beta\gamma}$ cannot learn γ with high probability. To formalize this note that

1. All queries to $\mathrm{C}_{\alpha\beta}$ result in 0 with high probability.
2. $S^{C,D}$ cannot learn what α and β are with high probability, since it does not have access to α. This implies that all queries to D result in 0 with high probability.

Everything we have done until now also works for circuits! But note that the proof is not over yet, we still want to combine $\mathrm{C}_{\alpha\beta}$ and $\mathrm{D}_{\alpha\beta\gamma}$ into a single program and show that obfuscating it is impossible. Here, we will make use of the Turing Machine Example.

We will now define $\mathrm{E}_{\alpha\beta\gamma}$ as

$$\mathrm{E}_{\alpha\beta\gamma}(\mu, x) = \begin{cases} \mathrm{C}_{\alpha\beta}(x) & \text{if } \mu = 0 \\ \mathrm{D}_{\alpha\beta\gamma}(x) & \text{if } \mu = 1 \end{cases}$$

By the same reasoning, given $\hat{E}$ to $\mathcal{A}$, it can get γ by hardcoding 0 and 1 into μ order to get programs that run $\hat{C}$ and $\hat{(D)}$, and then we can get $\hat{D}(\hat{C}) = \gamma$. However, similar to before, $S^{C,D}$ cannot learn γ.

5.3 Circuits

So why does the argument above with $\mathrm{E}_{\alpha\beta\gamma}$ not work with circuits? The main reason is that we cannot run a circuit on itself, because of input sizes. Sicne we get $\hat{C}$ and $\hat{D}$ from $\hat{E}$ as

$$\hat{C}(\cdot) = \hat{E}(0, \cdot)$$
$$\hat{D}(\cdot) = \hat{E}(1, \cdot)$$

when we want to run $\hat{D}$ on the input $\hat{C}$ we will essentially be inputting $\hat{E}$ into itself. Since we obtain $\hat{C}$ and $\hat{D}$ from $\hat{E}$, we can informally see that $|\hat{C}|, |\hat{D}| \approx |\hat{E}|$. But then we have

$$|D| > |\hat{C}| \approx |\hat{E}| \approx |\hat{D}| \geq |D|$$

So now we will try to extend this to circuits.

Idea — *Assume a Fully Homomorphic Encryption Scheme (which can be obtained with OWF + VBB).*

Assume the same α, β, γ as before.

$$C(x) = \begin{cases} \mathsf{Enc}(k, \alpha) & \text{if } x = 0 \\ \beta & \text{if } x = \alpha \\ \gamma & \text{if } \beta = \mathsf{Dec}(k, x) \\ 0 & \text{otherwise} \end{cases}$$

Now given an adversary $\mathcal{A}$ that is given $\hat{C}$, it first gets a ciphertext $c = \hat{C}(0)$ (this encrypts α) and uses it to get a new ciphertext $d = \mathrm{Eval}(\hat{C}, c)$ (this encrypts β) and finally can obtain γ by evaluating $\hat{D}(d)$.

On the other hand, the simulator S^C that is only given oracle access to C can only can learn $\mathsf{Enc}(k, \alpha)$ but no way to learn $\mathsf{Enc}(k, \beta)$ since it cannot do the homomorphic operations, and therefore has no way to learn γ. This concludes the case with circuits.