CS 294. Constrained PRFs and Program Obfuscation

1 Constrained PRF

A constrained PRF is a special type of PRF where the owner of the PRF key K can construct a special key K_f which enables anyone to compute

$$\forall x \text{ s.t. } f(x) = 1: \quad \mathsf{PRF}(K, x)$$

The PRF values at all other values should remain hidden given K_f, the *constrained key*.

We will only consider single-key CPRFs here, that is the adversary gets to see the constrained key for a single function f of her choice. Constructing many-key CPRFs from LWE is another one of my favorite open problems!

More generally, the adversary can get a single constrained key together with oracle access to the PRF (as usual). Her job is to compute $\mathsf{PRF}(K, x^*)$ for some x^* where (a) $f(x^*) = 0$ (false) and (b) she did not make an oracle query on x^*.

2 Private Constrained PRFs

A (single key) private constrained PRF is a constrained PRF where, in addition, the constrained key hides the function that it constrains.

In other words, given a constrained key (denoted as $K\{f\}$) and oracle access to $\mathsf{PRF}_K(\cdot)$, it is computationally hard to determine what f is. There are of course settings where this is impossible to achieve. (Can you think of one?)

Private Constrained PRFs

A (single key) private constrained PRF is a constrained PRF where, in addition, the constrained key hides the function that it constrains.

In other words, given a constrained key (denoted as $K\{f\}$) and oracle access to $\mathsf{PRF}_K(\cdot)$, it is computationally hard to determine what f is. There are of course settings where this is impossible to achieve.

For example, assume that an adversary gets $K\{f\}$ and wants to check if $f(x) = 0$ or 1 for some x in her mind. This can be done: she makes an oracle query to x and gets $\mathsf{PRF}_K(x)$. She also uses $K\{f\}$ to compute something, and she knows that the output matches $\mathsf{PRF}_K(x)$ iff $f(x) = 1$. This reveals some information, so it is not reasonable to expect to hide all information about f given $K\{f\}$ and oracle access to the PRF.

How then shall we define private constrained PRFs?

Private Constrained PRFs

Security is defined in terms of an indistinguishability game where the adversary produces two functions f_0, f_1 and gets oracle access to PRF but only for those x for which $f_0(x) = f_1(x)$.

3 Private Constrained PRF: Construction

We will start with the following PRF (that we will call dual-BLMR).

$$\mathsf{PRF}_{\{\mathbf{S}_{i,b}\}}(x_1 x_2 \ldots x_\ell) = \lfloor \mathbf{S}_{1,x_1}\mathbf{S}_{2,x_2}\ldots\mathbf{S}_{\ell,x_\ell}\mathbf{A}\rceil_p \pmod q$$

where $\mathbf{S}_{i,b}$ are (secret) square matrices with random small entries, and $\mathbf{A}$ is a (public) uniformly random matrix mod q.

(This is closely related to the BLMR construction which if you recall looks as follows:

$$\mathsf{PRF}_{\mathbf{S}}(x_1 x_2 \ldots x_\ell) = \lfloor \mathbf{S}\mathbf{A}_{1,x_1}\mathbf{G}^{-}(\mathbf{A}_{2,x_2})\ldots\mathbf{G}^{-}(\mathbf{A}_{\ell,x_\ell})\rceil_p \pmod q$$

where the $\mathbf{A}_{i,x_i}$ are random matrices. This is actually a slight generalization of BLMR using a secret *matrix* $\mathbf{S}$ and 2ℓ public matrices.)

We'd like to generate constrained keys for dual-BLMR.

For starters, let's try generating the constrained key for the identity function. That is, given the constrained key, one should be able to compute the PRF on *all* inputs. (We want the construction to be non-trivial in the sense that the constrained key should not reveal the PRF key).

Here is a try.

$$\begin{array}{llll} \mathbf{S}_{1,0}\mathbf{A} + \mathbf{E}_{1,0}, & \mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}, & \ldots, & \mathbf{S}_{\ell,0}\mathbf{A} + \mathbf{E}_{\ell,0} \\ \mathbf{S}_{1,1}\mathbf{A} + \mathbf{E}_{1,1}, & \mathbf{S}_{2,1}\mathbf{A} + \mathbf{E}_{2,1}, & \ldots, & \mathbf{S}_{\ell,1}\mathbf{A} + \mathbf{E}_{\ell,1} \end{array}$$

This certainly hides the $\mathbf{S}_{i,b}$ (the PRF key) but it's not clear how to compute the PRF output from it.

When you are stuck, you start by naming things. So, let's do it.

$$\begin{array}{llll} \mathbf{B}_{1,0} := \mathbf{S}_{1,0}\mathbf{A} + \mathbf{E}_{1,0}, & \mathbf{B}_{2,0} := \mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}, & \dots, & \mathbf{B}_{\ell,0} := \mathbf{S}_{\ell,0}\mathbf{A} + \mathbf{E}_{\ell,0} \\ \mathbf{B}_{1,1} := \mathbf{S}_{1,1}\mathbf{A} + \mathbf{E}_{1,1}, & \mathbf{B}_{2,1} := \mathbf{S}_{2,1}\mathbf{A} + \mathbf{E}_{2,1}, & \dots, & \mathbf{B}_{\ell,1} := \mathbf{S}_{\ell,1}\mathbf{A} + \mathbf{E}_{\ell,1} \end{array}$$

It's not clear what you get by multiplying, say, $\mathbf{B}_{1,0}$ with $\mathbf{B}_{2,0}$. What we need is to enable some sort of homomorphic multiplication. Let's take inspiration from GSW13.

$$\begin{array}{llll} \mathbf{B}_{1,0} := \mathbf{S}_{1,0}\mathbf{A} + \mathbf{E}_{1,0}, & \mathbf{B}_{2,0} := \mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}, & \ldots, & \mathbf{B}_{\ell,0} := \mathbf{S}_{\ell,0}\mathbf{A} + \mathbf{E}_{\ell,0} \\ \mathbf{B}_{1,1} := \mathbf{S}_{1,1}\mathbf{A} + \mathbf{E}_{1,1}, & \mathbf{B}_{2,1} := \mathbf{S}_{2,1}\mathbf{A} + \mathbf{E}_{2,1}, & \ldots, & \mathbf{B}_{\ell,1} := \mathbf{S}_{\ell,1}\mathbf{A} + \mathbf{E}_{\ell,1} \end{array}$$

Here is how GSW enables multiplication.

$$\begin{aligned} \mathbf{B}_{1,0} \cdot \mathbf{A}^{-1}(\mathbf{B}_{2,0}) &= \mathbf{S}_{1,0}\mathbf{A} + \mathbf{E}_{1,0} \cdot \mathbf{A}^{-1}(\mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}) \\ &= \mathbf{S}_{1,0}\mathbf{A} \cdot \mathbf{A}^{-1}(\mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}) + \underbrace{\mathbf{E}_{1,0}\mathbf{A}^{-1}(\mathbf{B}_{2,0})}_{\approx 0} \\ &\approx \mathbf{S}_{1,0} \cdot (\mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}) \\ &= \mathbf{S}_{1,0}\mathbf{S}_{2,0}\mathbf{A} + \underbrace{\mathbf{S}_{1,0}\mathbf{E}_{2,0}}_{\approx 0} \\ &\approx \mathbf{S}_{1,0}\mathbf{S}_{2,0}\mathbf{A} \end{aligned}$$

where $\mathbf{A}^{-1}(\mathbf{B})$ is a matrix $\mathbf{R}$ with small random entries such that $\mathbf{A}\mathbf{R} = \mathbf{B} \pmod q$.

Continuing along these lines, you can compute the PRF on all inputs if you can compute $\mathbf{A}^{-1}(\mathbf{B}_{i,b})$.

$$\begin{array}{llll} \mathbf{B}_{1,0} := \mathbf{S}_{1,0}\mathbf{A} + \mathbf{E}_{1,0}, & \mathbf{B}_{2,0} := \mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}, & \ldots, & \mathbf{B}_{\ell,0} := \mathbf{S}_{\ell,0}\mathbf{A} + \mathbf{E}_{\ell,0} \\ \mathbf{B}_{1,1} := \mathbf{S}_{1,1}\mathbf{A} + \mathbf{E}_{1,1}, & \mathbf{B}_{2,1} := \mathbf{S}_{2,1}\mathbf{A} + \mathbf{E}_{2,1}, & \ldots, & \mathbf{B}_{\ell,1} := \mathbf{S}_{\ell,1}\mathbf{A} + \mathbf{E}_{\ell,1} \end{array}$$

Here is how GSW enables multiplication.

$$\begin{aligned} \mathbf{B}_{1,0} \cdot \mathbf{A}^{-1}(\mathbf{B}_{2,0}) &= \mathbf{S}_{1,0}\mathbf{A} + \mathbf{E}_{1,0} \cdot \mathbf{A}^{-1}(\mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}) \\ &= \mathbf{S}_{1,0}\mathbf{A} \cdot \mathbf{A}^{-1}(\mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}) + \underbrace{\mathbf{E}_{1,0}\mathbf{A}^{-1}(\mathbf{B}_{2,0})}_{\approx 0} \\ &\approx \mathbf{S}_{1,0} \cdot (\mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}) \\ &= \mathbf{S}_{1,0}\mathbf{S}_{2,0}\mathbf{A} + \underbrace{\mathbf{S}_{1,0}\mathbf{E}_{2,0}}_{\approx 0} \\ &\approx \mathbf{S}_{1,0}\mathbf{S}_{2,0}\mathbf{A} \end{aligned}$$

where $\mathbf{A}^{-1}(\mathbf{B})$ is a matrix $\mathbf{R}$ with small random entries such that $\mathbf{AR} = \mathbf{B} \pmod{q}$.

Continuing along these lines, you can compute the PRF on all inputs. Since computing $\mathbf{A}^{-1}(\cdot)$ requires the trapdoor of $\mathbf{A}$, the owner of the PRF key can precompute all these matrices

$$\mathbf{D}_{i,b} := \mathbf{A}^{-1}(\mathbf{S}_{i,b}\mathbf{A} + \mathbf{E}_{i,b})$$

and release them.

$$\begin{array}{llll}\mathbf{D}_{1,0} := \mathbf{A}^{-1}(\mathbf{S}_{1,0}\mathbf{A} + \mathbf{E}_{1,0}), & \mathbf{D}_{2,0} := \mathbf{A}^{-1}(\mathbf{S}_{2,0}\mathbf{A} + \mathbf{E}_{2,0}), & \ldots, & \mathbf{D}_{\ell,0} := \mathbf{A}^{-1}(\mathbf{S}_{\ell,0}\mathbf{A} + \mathbf{E}_{\ell,0}) \\ \mathbf{D}_{1,1} := \mathbf{A}^{-1}(\mathbf{S}_{1,1}\mathbf{A} + \mathbf{E}_{1,1}), & \mathbf{D}_{2,1} := \mathbf{A}^{-1}(\mathbf{S}_{2,1}\mathbf{A} + \mathbf{E}_{2,1}), & \ldots, & \mathbf{D}_{\ell,1} := \mathbf{A}^{-1}(\mathbf{S}_{\ell,1}\mathbf{A} + \mathbf{E}_{\ell,1})\end{array}$$

How about security? This is tricky because we would like to argue that each $\mathbf{S}_{i,b}\mathbf{A} + \mathbf{E}_{i,b}$ is pseudorandom, invoking LWE. But the constrained key depends on the trapdoor for $\mathbf{A}$ in the presence of which LWE is *not* hard.

So, let's modify the construction a bit more.

$$\mathbf{A}_0, \begin{array}{llll} \mathbf{D}_{1,0} := \mathbf{A}_0^{-1}(\mathbf{S}_{1,0}\mathbf{A}_1 + \mathbf{E}_{1,0}), & \mathbf{D}_{2,0} := \mathbf{A}_1^{-1}(\mathbf{S}_{2,0}\mathbf{A}_2 + \mathbf{E}_{2,0}), & \ldots, & \mathbf{D}_{\ell,0} := \mathbf{A}_{\ell-1}^{-1}(\mathbf{S}_{\ell,0}\mathbf{A}_\ell + \mathbf{E}_{\ell,0}) \\ \mathbf{D}_{1,1} := \mathbf{A}_0^{-1}(\mathbf{S}_{1,1}\mathbf{A}_1 + \mathbf{E}_{1,1}), & \mathbf{D}_{2,1} := \mathbf{A}_1^{-1}(\mathbf{S}_{2,1}\mathbf{A}_2 + \mathbf{E}_{2,1}), & \ldots, & \mathbf{D}_{\ell,1} := \mathbf{A}_{\ell-1}^{-1}(\mathbf{S}_{\ell,1}\mathbf{A}_\ell + \mathbf{E}_{\ell,1}) \end{array}$$

where A_i are *distinct* random matrices (chosen during the generation of the constrained key), and we think of $\mathbf{A}_\ell = \mathbf{A}$.

We have that

$$\mathbf{A}_0\mathbf{D}_{1,x_1}\mathbf{D}_{2,x_2}\ldots\mathbf{D}_{\ell,x_\ell} \approx \mathbf{S}_{1,x_1}\mathbf{S}_{2,x_2}\ldots\mathbf{S}_{\ell,x_\ell}\mathbf{A} \pmod q$$

Thus, we have a construction of a PRF together with a constrained key (and an algorithm to generate it) that can evaluate the PRF at all inputs while (plausibly) hiding the original PRF key.

We now have two questions: (a) allow more expressive constraint functions and (b) show security! We will do these in turn.

3.1 Interlude: Matrix Branching Programs

This is a convenient model of computation for us as we are already working with matrices! In a matrix branching program computing a Boolean function f on k input bits, we have 2ℓ matrices $\mathbf{M}_{i,b} \in \{0,1\}^{w \times w}$ (where think of w as a constant) and a vector $\mathbf{v} \in \{0,1\}^{1 \times w}$ where $\ell \geq k$:

$$\mathbf{v} = (1\ 0\ \ldots 0), \begin{array}{cccc} \mathbf{M}_{1,0} & \mathbf{M}_{2,0} & \ldots & \mathbf{M}_{\ell,0} \\ \mathbf{M}_{1,1} & \mathbf{M}_{2,1} & \ldots & \mathbf{M}_{\ell,1} \end{array}$$

In general, each location i (corresponding to a pair of **full-rank** matrices $\mathbf{M}_{i,0}$ and $\mathbf{M}_{i,1}$) is indexed by an input bit, say $x_j = x_{j(i)}$. To avoid complicating matters, we will let $\ell = k$ and let $j(i) = i$.

To compute the program on an input x, you compute

$$\mathbf{u} := \mathbf{v}\mathbf{M}_{1,x_1}\mathbf{M}_{2,x_2} \ldots \mathbf{M}_{\ell,x_\ell}$$

$\mathbf{u}$ is either $(1\ 0\ \ldots 0)$ or $(0\ 1\ \ldots 0)$. If $\mathbf{u}[1] \neq 0$, output 1 (**true**), otherwise output 0 (**false**).

We know that every function in NC1 (circuits with log depth and poly size) can be computed by poly-size matrix branching programs where each matrix is 5-by-5 permutation matrix (it's in S_5). This is Barrington's theorem. So, matrix branching programs are a pretty powerful computational model.

(Give a simple example of a matrix branching program.)

Let's now incorporate matrix BPs into the constrained key. The construction is due to Canetti and Chen (2017), improved later by Chen-Vaikuntanathan-Wee'18.

Say you want to constrain the PRF key for a constraint function f. Create a length-ℓ matrix branching program for f first.

The constrained key is

$$\hat{\mathbf{v}}\mathbf{A}_0, \begin{array}{llll} \mathbf{D}_{1,0} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,0}\mathbf{A}_1 + \mathbf{E}_{1,0}), & \mathbf{D}_{2,0} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,0}\mathbf{A}_2 + \mathbf{E}_{2,0}), & \ldots, & \mathbf{D}_{\ell,0} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,0}\mathbf{A}_\ell + \mathbf{E}_{\ell,0}) \\ \mathbf{D}_{1,1} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,1}\mathbf{A}_1 + \mathbf{E}_{1,1}), & \mathbf{D}_{2,1} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,1}\mathbf{A}_2 + \mathbf{E}_{2,1}), & \ldots, & \mathbf{D}_{\ell,1} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,1}\mathbf{A}_\ell + \mathbf{E}_{\ell,1}) \end{array}$$

where $\hat{\mathbf{S}}_{i,b} = \mathbf{M}_{i,b} \otimes \mathbf{S}_{i,b}$ is a tensor product of the two matrices and $\hat{\mathbf{v}} = \mathbf{v} \otimes \mathbf{I}$.

The key property of tensor products is the following associative property.

$$(\mathbf{A} \otimes \mathbf{B}) \cdot (\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}$$

where $\otimes$ is the tensor product and $\cdot$ is matrix multiplication.

(Note that the $\hat{\mathbf{S}}_{i,b}$ are now $nw \times nw$ matrices and $\mathbf{A}_i$ have to be corresponding larger, i.e., $nw \times m$ for a large enough m.)

The constrained key is

$$\hat{\mathbf{v}}\mathbf{A}_0, \begin{array}{llll} \mathbf{D}_{1,0} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,0}\mathbf{A}_1 + \mathbf{E}_{1,0}), & \mathbf{D}_{2,0} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,0}\mathbf{A}_2 + \mathbf{E}_{2,0}), & \ldots, & \mathbf{D}_{\ell,0} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,0}\mathbf{A}_\ell + \mathbf{E}_{\ell,0}) \\ \mathbf{D}_{1,1} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,1}\mathbf{A}_1 + \mathbf{E}_{1,1}), & \mathbf{D}_{2,1} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,1}\mathbf{A}_2 + \mathbf{E}_{2,1}), & \ldots, & \mathbf{D}_{\ell,1} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,1}\mathbf{A}_\ell + \mathbf{E}_{\ell,1}) \end{array}$$

where $\hat{\mathbf{S}}_{i,b} = \mathbf{M}_{i,b} \otimes \mathbf{S}_{i,b}$ is a tensor product of the two matrices and $\hat{\mathbf{v}} = \mathbf{v} \otimes \mathbf{I}$.
The key property of tensor products is the following associative property.

$$(\mathbf{A} \otimes \mathbf{B}) \cdot (\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}$$

We have that

$$\hat{\mathbf{v}}\mathbf{A}_0\mathbf{D}_{1,x_1}\mathbf{D}_{2,x_2}\ldots\mathbf{D}_{\ell,x_\ell} \approx \hat{\mathbf{v}}\hat{\mathbf{S}}_{1,x_1}\hat{\mathbf{S}}_{2,x_2}\ldots\hat{\mathbf{S}}_{\ell,x_\ell}\mathbf{A}_\ell \pmod{q} = \Big((\mathbf{v}\prod \mathbf{M}_{i,x_i}) \otimes (\prod \mathbf{S}_{i,x_i})\Big)\mathbf{A}_\ell$$

The constrained key is

$$\hat{\mathbf{v}}\mathbf{A}_0, \begin{array}{llll} \mathbf{D}_{1,0} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,0}\mathbf{A}_1 + \mathbf{E}_{1,0}), & \mathbf{D}_{2,0} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,0}\mathbf{A}_2 + \mathbf{E}_{2,0}), & \ldots, & \mathbf{D}_{\ell,0} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,0}\mathbf{A}_\ell + \mathbf{E}_{\ell,0}) \\ \mathbf{D}_{1,1} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,1}\mathbf{A}_1 + \mathbf{E}_{1,1}), & \mathbf{D}_{2,1} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,1}\mathbf{A}_2 + \mathbf{E}_{2,1}), & \ldots, & \mathbf{D}_{\ell,1} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,1}\mathbf{A}_\ell + \mathbf{E}_{\ell,1}) \end{array}$$

where $\hat{\mathbf{S}}_{i,b} = \mathbf{M}_{i,b} \otimes \mathbf{S}_{i,b}$ is a tensor product of the two matrices and $\hat{\mathbf{v}} = \mathbf{v} \otimes \mathbf{I}$.
We have that

$$\hat{\mathbf{v}}\mathbf{A}_0\mathbf{D}_{1,x_1}\mathbf{D}_{2,x_2}\ldots\mathbf{D}_{\ell,x_\ell} \approx \hat{\mathbf{v}}\hat{\mathbf{S}}_{1,x_1}\hat{\mathbf{S}}_{2,x_2}\ldots\hat{\mathbf{S}}_{\ell,x_\ell}\mathbf{A}_\ell \pmod q = \Big((\mathbf{v}\prod\mathbf{M}_{i,x_i}) \otimes (\prod\mathbf{S}_{i,x_i})\Big)\mathbf{A}_\ell$$

We know that $\mathbf{v}\prod\mathbf{M}_{i,x_i}$ is either $(1\ 0\ 0\ \ldots 0)$ or $(0\ 1\ 0\ \ldots 0)$. So the entire product is close to $\prod\mathbf{S}_{i,x_i}\mathbf{A}_\ell^{(1)}$ if $f(x) = 1$ or $\prod\mathbf{S}_{i,x_i}\mathbf{A}_\ell^{(2)}$ if $f(x) = 0$. where

$$\mathbf{A}_\ell := \begin{bmatrix} \mathbf{A}_\ell^{(1)} \\ \mathbf{A}_\ell^{(2)} \\ \ldots \\ \mathbf{A}_\ell^{(w)} \end{bmatrix}$$

So, letting $\mathbf{A}_\ell^{(1)} := \mathbf{A}$ in the definition of the PRF finishes the construction.

We will give this monster a compact name:

$$\hat{\mathbf{v}}\mathbf{A}_0, \begin{array}{llll} \mathbf{D}_{1,0} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,0}\mathbf{A}_1 + \mathbf{E}_{1,0}), & \mathbf{D}_{2,0} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,0}\mathbf{A}_2 + \mathbf{E}_{2,0}), & \dots, & \mathbf{D}_{\ell,0} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,0}\mathbf{A}_\ell + \mathbf{E}_{\ell,0}) \\ \mathbf{D}_{1,1} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,1}\mathbf{A}_1 + \mathbf{E}_{1,1}), & \mathbf{D}_{2,1} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,1}\mathbf{A}_2 + \mathbf{E}_{2,1}), & \dots, & \mathbf{D}_{\ell,1} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,1}\mathbf{A}_\ell + \mathbf{E}_{\ell,1}) \end{array}$$

Call this a **GGH15 chain** (after the inventors Gentry, Gorbunov and Halevi) for the program f, secrets $\mathbf{S}_{i,b}$ and the final matrices $\mathbf{A}_\ell^{(1)} := \mathbf{A}^{(1)}$ and $\mathbf{A}_\ell^{(2)} := \mathbf{A}^{(2)}$.

On input x, we can compute

$$\approx \mathbf{S}_x \mathbf{A}^{(2-f(x))}$$

where now and henceforth $\mathbf{S}_x := \prod \mathbf{S}_{i,x_i}$.

We will now sketch the proof that the scheme is constraint-hiding. We start by showing that the constrained key is pseudorandom.

$$\hat{\mathbf{v}}\mathbf{A}_0, \begin{array}{llll} \mathbf{D}_{1,0} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,0}\mathbf{A}_1 + \mathbf{E}_{1,0}), & \mathbf{D}_{2,0} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,0}\mathbf{A}_2 + \mathbf{E}_{2,0}), & \ldots, & \mathbf{D}_{\ell,0} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,0}\mathbf{A}_\ell + \mathbf{E}_{\ell,0}) \\ \mathbf{D}_{1,1} := \mathbf{A}_0^{-1}(\hat{\mathbf{S}}_{1,1}\mathbf{A}_1 + \mathbf{E}_{1,1}), & \mathbf{D}_{2,1} := \mathbf{A}_1^{-1}(\hat{\mathbf{S}}_{2,1}\mathbf{A}_2 + \mathbf{E}_{2,1}), & \ldots, & \mathbf{D}_{\ell,1} := \mathbf{A}_{\ell-1}^{-1}(\hat{\mathbf{S}}_{\ell,1}\mathbf{A}_\ell + \mathbf{E}_{\ell,1}) \end{array}$$

1. Observe first that
$$\hat{\mathbf{S}}\mathbf{A} + \mathbf{E}$$
is pseudorandom by LWE where $\hat{\mathbf{S}} = \mathbf{M} \otimes \mathbf{S}$ where $\mathbf{S}$ is a random small matrix and $\mathbf{M}$ is any **full-rank** matrix. (Can you see what happens if $\mathbf{M}$ is not full-rank?)

2. The trapdoor issue still rears its head, that is, the constrained key is a function of trapdoors for various matrices $\mathbf{A}$, in the presence of which LWE for those matrices does not hold!

   However, and this is the thing that saves us, observe that the trapdoor for $\mathbf{A}_\ell$ is **never used**!!

3. So, we can do a *right-to-left* proof where we replace matrices by random small matrices in two mini-steps:

   - First replace $\hat{\mathbf{S}}_{\ell,b}\mathbf{A}_\ell + \mathbf{E}_{\ell,b}$ by uniformly random and independent matrices $\mathbf{U}_{\ell,b}$. This is by an invocation of LWE.
   - Second, replace $\mathbf{A}_{\ell-1}^{-1}(\mathbf{U}_{\ell,b})$ by a Gaussian matrix $\mathbf{D}_{\ell,b}$. This is by an invocation of the GPV theorem (the same thing we used to construct digital signatures via Gaussian sampling.)
   - Now, we are at a hybrid experiment where we *never use the trapdoor for* $\mathbf{A}_{\ell-1}$! Rinse and repeat.

Of course, we need to prove more. That it is constraint-hiding even with oracle access to the PRF (in the sense that we defined) and that the constraint key does not enable evaluation of PRF on x such that $f(x) = 0$. We will leave these as a (non-trivial but doable) exercise.

An advanced comment: Before we move on, let's ask if we can use this to release more than one constrained key. Ie, can we plausibly conjecture security?

4 Program Obfuscation and Other Beasts

How much more useful is it to have the *code* of a program than merely *oracle* access (or input/output access) to it? This is a foundational question in cryptography, and indeed in theoretical computer science and even all of computing.

In a cryptographic context, we ask: can we transform a program into another (*obfuscated*) program which has the same input/output functionality but is no more revealing than having black-box access? This is the problem of *program obfuscation.*

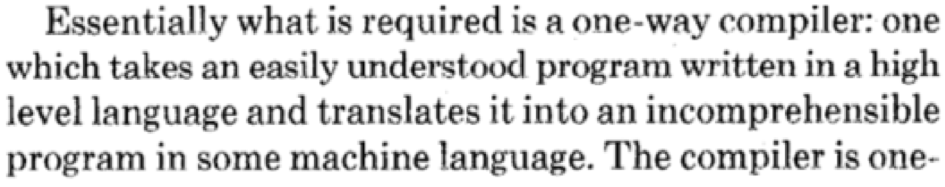
On the one hand, given a program P, it is hard to even say whether it halts on input 0^n (this is the Halting problem). Indeed, any "non-trivial" property of a program is undecidable (this is Rice's theorem). So, worst-case programs seem naturally obfuscated. Yet, these are programs we do not necessarily care about. This brings into sharper focus the problem of program obfuscation, the problem of transforming *arbitrary* programs, ones that we do care about, into their obfuscated versions.

Aside from their obvious uses in software protection, program obfuscation is of fundamental importance to cryptography. Let us illustrate two of their uses.

4.1 Applications of Program Obfuscation (Informally)

In the early 1970s, the big problem in cryptography was whether there is a method of encrypting messages from A to B which does not require A and B to have met beforehand and come up with a common private key. In other words, is public-key cryptography possible?

In a landmark paper that kickstarted the field, Diffie and Hellman wondered about the following possibility. Take the encryption program of a secret-key encryption scheme and obfuscate it!



Essentially what is required is a one-way compiler: one which takes an easily understood program written in a high level language and translates it into an incomprehensible program in some machine language. The compiler is one-

They didn't quite manage to make it work, and went a different route, but it's a fascinating route.

Indeed, being able to obfuscate programs (in an appropriate sense) makes nearly every cryptographic task trivial. Can you see how to achieve fully homomorphic encryption if I gave you a way to obfuscate programs?

4.2 Defining Program Obfuscation

What does it mathematically mean to obfuscate programs?

One way to define it leads to the notion of *virtual black-box obfuscation* due to Hada and Barak et al. In a nutshell, it defines an obfuscator $\mathcal{O}$ to be a probabilistic algorithm that takes programs (or circuits or Turing machines...) and converts them into other programs that are:

- *functionally equivalent* and *nearly as fast* (asymptotically!); and
- *virtual black-box.* That is, for every PPT adversary A that tries to learn a predicate of the original program given its obfuscation, there is a black-box PPT adversary S (also called the simulator) that does the same thing given oracle access. That is,

$$\forall A, \exists S, \forall \pi : \{0,1\}^* \to \{0,1\} \text{ and } \forall \text{ programs } P :$$
$$\Pr[A(\mathcal{O}(P) = \pi(P)] \approx \Pr[S^P(1^n) = \pi(P)]$$

This is a pretty strong definition and formalizes the idea that the obfuscated program should be *no more revealing* than black-box access to it. Unfortunately, it is also impossible to construct a universal program obfuscator. (can you see, perhaps informally, why?)

4.3 Defining Program Obfuscation: Take 2

Nevertheless, researchers have shown multiple paths to circumvent this (one!) impossibility. The most well-studied is to relax the *definition* to indistinguishability obfuscation. We will explore a different route today, relaxing the *class of functions* we plan to obfuscate. In particular, we will look at the following class of functions.

$$F_{f,\alpha,\beta}(x) = \begin{cases} \beta & \text{if } f(x) = \alpha \\ 0 & \text{otherwise} \end{cases}$$

where f is some function and α, β are strings (of length at least the security parameter).

We call this *lockable obfuscation*, a terminology due to Wichs-Zirdelis'17 and Goyal-Koppula-Waters'17. (A special case of this is obfuscating point functions).

(A slightly weaker version we will look at is

$$F_{f,\alpha}(x)$$

which outputs 1 if $f(x) = \alpha$ and 0 otherwise.)

We will obfuscate this class when f is pretty complex (all we need is that there is a matrix branching program that computes it) and α is a uniformly random string (really, all we need is that it has large min-entropy.) In fact, in this world, we require that the lockable obfuscation for any function f, random α and arbitrary β is *pseudorandom* or *simulatable with no other information.*

5 Lockable Obfuscation: An Application

As we mentioned in the last class, it is easy to construct an example of an encryption scheme which is CPA-secure but releasing an encryption of the secret key under the matching public key is completely insecure (let's try!)

However, what if the encryption algorithm is restricted to encrypting bits? That is, when we say "encrypt the secret key", we will encrypt the bits of the secret key one by one. The counterexample we just constructed falls apart. So maybe every *bit encryption scheme* is circular-secure?! For a while, we did not have counterexamples for this under plausible conjectures, but now we do, thanks to lockable obfuscation.

Take any CPA-secure encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$. Modify it to $(\mathsf{KeyGen}', \mathsf{Enc}', \mathsf{Dec}')$ that works as follows.

- KeyGen' generates a (pk, sk) pair from KeyGen and lets

  $$pk' = (pk, \mathsf{LO}(F_{f_{sk},\alpha,sk})) \text{ and } sk' = (sk, \alpha)$$

  for a random α. Here, f_{sk} takes a bunch of ciphertexts and decrypts them using sk. That is, if you feed an encryption of α to the lockable obfuscation, it reveals sk. You see where this is going!

- Enc' is the same as Enc.

- Dec' is the same as Dec.

This scheme remains CPA-secure (using the security of lockable obfuscation) but it is not circular-secure.

6 Lockable Obfuscation: Construction

The final item for the day is a construction of lockable obfuscation using the machinery of GGH15 chains. (We will show the slightly weaker construction today, but it's easy to modify it to get the stronger version.) To do lockable obfuscation $F_{f,\alpha}$ of a function f with lock $\alpha \in \{0,1\}^{2\lambda}$, do:

- generate 2λ matrices $\mathbf{A}^{(j,b)}$ with $j \in [\lambda]$ and $b \in \{0,1\}$ such that

$$\sum_j \mathbf{A}^{(j,\alpha_j)} = 0 \pmod q$$

- generate λ GGH15 chains, one for the function f_j that outputs the j-th bit of f, and the final matrix pair $(\mathbf{A}^{(j,0)}, \mathbf{A}^{(j,1)})$. All GGH15 chains *share the same* $\mathbf{S}$ *matrices.*

For correctness, note that we can compute

$$\approx \mathbf{S}_x \mathbf{A}^{(j,f_j(x))}$$

for all j. Sum them up to get

$$\approx \mathbf{S}_x \sum_j \mathbf{A}^{(j,f_j(x))}$$

This sum is ≈ 0 if each $f_j(x) = \alpha_j$, in other words if $f(x) = \alpha$.

We will only say two words about security, which we argue in two steps.

1. First, the fact that α is random (or has min-entropy) can be used to show using Leftover hash lemma that the matrices $\mathbf{A}^{(j,b)}$ are truly random (as if there were no additive constraint on them.)

2. Now, we have ℓ GGH15 chains with the same $\mathbf{S}$ matrices but random and independent $\mathbf{A}$ matrices. The same proof that we did before can be argued to show security in this case as well.