CSC 2419: Lattice-based Cryptography **Fall 2025**

Lecture 9: Compute-and-Compare Obfuscation

Instructor: Akshayaram Srinivasan *Scribe: Jason Liu*

Date: 2025-11-17

9.1 Program Obfuscation

The goal of program obfuscation is to encode a program in a way that preserves its functionality while hiding its codes and internal operations.

Consider a family of programs $\mathcal{P}$. An obfuscation scheme consist of a pair of algorithms $(\mathsf{Obf}, \mathsf{Eval})$ (where Obf is a PPT algorithm and Eval is a deterministic polynomial time algorithm). The *obfuscator* Obf takes as input a program $P \in \mathcal{P}$, a security parameter $\lambda \in \mathbb{N}$, and outputs another program $\widetilde{P} \leftarrow \mathsf{Obf}(1^\lambda, P)$, such that

$$\Pr[\forall x \in \{0,1\}^n : P(x) = \mathsf{Eval}(\widetilde{P}, x)] \geq 1 - \mathrm{negl}(\lambda)$$

where the probability is over the choice of randomness used by Obf.

To formulate security, we use the notion of *virtual black-box* security introduced by Barak et al.[BGI+01]. Intuitively, the security requirement states that, given the obfuscated program $\widetilde{P}$, any information that can be learnt by an adversary about the underlying program P can be simulated given oracle access to input-output behaviour of P.

More formally, for every poly-time adversary $\mathcal{A}$, there exists a PPT simulator Sim such that $\forall P \in \mathcal{P}$ and poly-size predicates $\phi : \mathcal{P} \to \{0,1\}$,

$$\left|\Pr[\mathcal{A}(\widetilde{P}) = \phi(P)] - \Pr[\mathsf{Sim}^P(1^\lambda) = \phi(P)]\right| \leq \mathrm{negl}(\lambda)$$

Unfortunately, Barak et al. showed that VBB obfuscation is unachievable for the family of polynomial time programs, by giving a family of poly-time programs that is *inherently unobfuscatable*. More specifically, they described a family of programs $\mathcal{P}$ and a property $\pi : \mathcal{P} \to \{0,1\}$ such that

- for any $P \in \mathcal{P}$, the value of $\pi(P)$ can be efficiently determined from *any* program that computes P;
- no efficient algorithm with black-box access to a oracle $P(\cdot)$ can compute $\pi(P)$ with a higher advantage than simply guessing the output of $\pi(P)$.

In this lecture, we will be looking at VBB obfuscation for one such class of programs, namely *Compare-and-Compute Programs*.

9.2 Compute-and-Compare Programs

A *compute-and-compare* (CC) program is parameterized by a function $f : \{0,1\}^{\ell_{in}} \to \{0,1\}^{\ell_{out}}$ together with a target value $y \in \{0,1\}^{\ell_{out}}$. On an input $x \in \{0,1\}^{\ell_{in}}$,

$$\mathsf{CC}[f,y](x) = \begin{cases} 1 & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases}$$

We focus on obfuscating compute-and-compare programs $\mathsf{CC}[f,y]$ when the target value y is chosen uniformly from $\{0,1\}^{\ell_{out}}$ and the function f is a *Permutation Branching Program*. These programs are a special case of *evasive programs*. Informally speaking, for any input x chosen a-priori, the program outputs 0 with overwhelming probability.

Remark 9.1 *Oracle access to an evasive program can be simulated by simply returning the output of each oracle call as 0. With overwhelming probability, this would simulate the output of the oracle call. This implies that while arguing the simulatability of VBB obfuscation of an evasive program P, the simulator can be defined without oracle access to input-outpu behaviour of P.*

In this lecture, we look at the VBB Obfuscation for *Compute-and-Compare* programs proposed by Wichs and Zirdelis [WZ17], where the security of the construction relied on the hardness of LWE assumption.

It should be noted that the VBB Obfuscation scheme proposed in [WZ17] defines the scheme for a larger class of *Compute-and-Compare Programs*, namely for all poly-time computable function and a target value y with high enough entropy to make it *unpredictable*, instead of being uniformly random. In this lecture, we focus on the more restrictive case.

9.3 Permutation Branching Programs

A *boolean permutation branching program* P of length L and width w is a graph containing $(L+1)\cdot w$ vertices that are grouped into $(L+1)$ layers with w vertices each. Each vertex has exactly two outgoing edges into the next layer, labeled with 0 and 1. It is required that the "1-edges" (resp. "0-edges") from one layer to the next layer form a permutation of $\{0,...,w-1\}$. More formally:

- **Vertices:** Vertices in P are of the form (i,j), for $i \in \{1,...,L+1\}$ and $j \in \{0,...,w-1\}$.
- **Transitions:** Each layer $i \leq L$ is associated with two permutations, $\pi_{i,0}$ and $\pi_{i,1}$, over $\{0,...,w-1\}$, that describe the transitions from layer i to $i+1$. A vertex (i,j) in layer i has two outgoing edges, labeled with 0 and 1, that respectively go to vertices $(i+1, \pi_{i,0}(j))$ and $(i+1, \pi_{i,1}(j))$ in layer $i+1$.
- **Computation:** On an input $x \in \{0,1\}^{\ell_{in}}$, the program begins on the vertex $(1,0)$ and proceeds through a sequence of L steps. During the i^{th} step of computation, the program moves from its current vertex (i,j) to vertex output by permutation corresponding to $x_{i \mod \ell_{in}}$ i.e.$(i+1, \pi_{i,x_{i \mod \ell_{in}}}(j))$ in the next layer.
- **Outputs:** A valid program guarantees that the computation ends on a vertex $(L+1,b)$ in the final layer, for some $b \in \{0,1\}$. In this case, the program outputs the value of b.

Barrington [Bar89] proved that the class of $O(\log n)$ depth circuits i.e. NC^1 is equivalent to the class of *Permutation Branching Programs* PBP

Theorem 9.2 ([Bar89]) $\mathsf{NC}^1 = \mathsf{PBP}$

More specifically, an ℓ_{out}-bit output branching program consists of ℓ_{out} separate 1-bit output branching programs $(P^{(k)})_{k\in\{1,...,\ell_{out}\}}$. A *compute-and-compare branching program* $\mathsf{CC}_{P,y}$ is parameterized by an ℓ_{out}-bit output branching program $P = (P^{(k)})_{k\in\{1,...,\ell_{out}\}}$ and a target value $y \in \{0,1\}^{\ell_{out}}$. On an input $x \in \{0,1\}^{\ell_{in}}$,

$$\mathsf{CC}_{P,y}(x) = \begin{cases} 1 & \text{if } P^{(k)}(x) = y_k \text{ for all } k \in \{1,...,\ell_{out}\} \\ 0 & \text{otherwise} \end{cases}$$

9.3.1 Obfuscating Compute-and-Compare Branching Programs

To simplify notation, we will denote that input read in i^{th} step of computation as x_i instead of $x_{i \mod \ell_{in}}$.

9.3.1.1 High Level Idea.

Fix a compute-and-compare branching program $\mathsf{CC}[P, y]$ for $P = (P^{(k)})_{k\in\{1,...,\ell_{out}\}}$ and $y \leftarrow \{0,1\}^{\ell_{out}}$.

Step 1: Encoding program $P^{(k)}$. The first step is to describe an encoded form of the permutation branching program $P^{(k)}$. For each vertex (i, j) in $P^{(k)}$, we associate with it a uniformly sampled public matrix $\mathbf{A}^{(k)}_{i,j} \leftarrow \mathbb{Z}_q^{n\times m}$. For each layer $i \in \{1, ..., L\}$, we sample two random "low-norm" secrets $\mathbf{S}_{i,0}, \mathbf{S}_{i,1}$ [1] and "low-norm" encodings $\mathbf{C}^{(k)}_{i,0}, \mathbf{C}^{(k)}_{i,1}$ such that

$$\underbrace{\begin{bmatrix} \mathbf{A}^{(k)}_{i,0} \\ \vdots \\ \mathbf{A}^{(k)}_{i,w-1} \end{bmatrix}}_{\mathbf{B}^{(k)}_i} \cdot \mathbf{C}^{(k)}_{i,b} = \underbrace{\begin{bmatrix} \mathbf{S}_{i,b} \cdot \mathbf{A}^{(k)}_{i+1,\pi_{i,b}(1)} + \mathbf{E}^{(k)}_{(i+1,0),b} \\ \vdots \\ \mathbf{S}_{i,b} \cdot \mathbf{A}^{(k)}_{i+1,\pi_{i,b}(w-1)} + \mathbf{E}^{(k)}_{(i+1,w-1),b} \end{bmatrix}}_{\mathbf{H}^{(k)}_{i,b}} \tag{9.1}$$

where $\mathbf{E}^{(k)}_{(i+1,j),b}$ is a low-norm error.

Remark 9.3 *In the actual protocol, the low norm matrix $\mathbf{C}^{(k)}_{i,b}$ is sampled using Pre-Image Sampling using Trapdoors (Lecture 4) i.e.*

$$\begin{aligned} (\mathbf{B}^{(k)}_i, \mathbf{T}^{(k)}_i) &\leftarrow \mathsf{TrapGen}(1^\lambda) \\ \mathbf{C}_{i,b} &\leftarrow \mathsf{TrapPreSamp}(1^\lambda \mathbf{T}^{(k)}_i, \mathbf{H}^{(k)}_{i,b}) \end{aligned}$$

Intuitively, the matrix $\mathbf{C}^{(k)}_{i,b}$ encodes the secret $\mathbf{S}_{i,b}$ along with the permutation $\pi_{i,b}$.

[1] The same secrets $\mathbf{S}_{i,b}$ are shared across ℓ_{out} instances of encodings.

Evaluating Encoding. To evaluate the encoding corresponding to program P on input $x \in \{0,1\}^{\ell_{in}}$, we compute

$$\mathbf{A}^{(k)}_{1,0} \cdot \mathbf{C}^{(k)}_{1,x_1} \cdots \mathbf{C}^{(k)}_{L,x_L}$$

We note that the following relation holds true:

$$\mathbf{A}^{(k)}_{1,0} \cdot \mathbf{C}^{(k)}_{1,x_1} \cdots \mathbf{C}^{(k)}_{L,x_L} = \mathbf{S}_{1,x_1} \cdots \mathbf{S}_{L,x_L} \cdot \mathbf{A}^{(k)}_{L+1,P(x)} + \widehat{\mathbf{E}}^{(k)} \tag{9.2}$$

where $\widehat{\mathbf{E}}^{(k)}$ is a low-norm noise. To see why the equality holds true, note that from Equation 9.1, note that

$$\begin{aligned}
\mathbf{A}^{(k)}_{1,0} \cdot \mathbf{C}^{(k)}_{1,x_1} &= \mathbf{S}_{1,x_1} \cdot \mathbf{A}^{(k)}_{2,\pi_{i,x_1}(1)} + \mathbf{E}^{(k)}_{(2,0),x_1} \\
\implies \mathbf{A}^{(k)}_{1,0} \cdot \mathbf{C}^{(k)}_{1,x_1} \cdot \mathbf{C}^{(k)}_{2,x_2} &= (\mathbf{S}_{1,x_1} \cdot \mathbf{A}^{(k)}_{2,\pi_{1,x_1}(1)} + \mathbf{E}^{(k)}_{(2,0),x_1})\mathbf{C}^{(k)}_{2,x_2} \\
&= \mathbf{S}_{1,x_1} \cdot \mathbf{S}_{2,x_2} \cdot \mathbf{A}^{(k)}_{3,\pi_{2,x_2}(\pi_{1,x_1}(0))} + \widehat{\mathbf{E}}'
\end{aligned}$$

From the computation above, we observe that

$$\begin{aligned}
\mathbf{A}^{(k)}_{1,0} \cdot \mathbf{C}^{(k)}_{1,x_1} \cdots \mathbf{C}^{(k)}_{L,x_L} =& \mathbf{S}_{1,x_1} \cdots \mathbf{S}_{L,x_L} \cdot \mathbf{A}^{(k)}_{L+1,\pi_{L,x_L}(\pi_{L_1,x_{L_1}} \cdots (\pi_{1,x_1}(0)))} + \widehat{\mathbf{E}}^{(k)} \\
=& \mathbf{S}_{1,x_1} \cdots \mathbf{S}_{L,x_L} \cdot \mathbf{A}^{(k)}_{L+1,P(x)} + \widehat{\mathbf{E}}^{(k)}
\end{aligned}$$

where the last equality follows from correctness of permutation branching programs.

Step 2: Obfuscating $\mathsf{CC}[P,y]$. Given the encoding for program $P^{(k)}$ generated in Step 1, we will now briefly describe VBB Obfuscation for $\mathsf{CC}[P,y]$ where $P = \{P^{(k)}\}_{k \in \ell_{out}}$. We proceed in an identical fashion as before, where we generate encodings as before with an additional constraint: the public matrices $\{\mathbf{A}_{L+1,y_k}\}_{k \in \ell_{out}}$ are sampled such that

$$\sum_{k=1}^{\ell_{out}} \mathbf{A}^{(k)}_{L+1,y_k} = 0 \tag{9.3}$$

where $\mathbf{A}^{(k)}_{L+1,y_k}$ refers to the matrix that corresponds to vertex $(L+1, y_k)$ in the k^{th} branching program $P^{(k)}$.

Correctness. If $f(x) = y$, the the encoding evaluated on input x would result in the output

$$\sum_{k=1}^{\ell_{out}} \left(\mathbf{A}^{(k)}_{1,0} \cdot \mathbf{C}^{(k)}_{1,x_1} \cdots \mathbf{C}^{(k)}_{L,x_L} \right) = \sum_{k=1}^{\ell_{out}} \left(\mathbf{S}_{1,x_1} \cdots \mathbf{S}_{L,x_L} \cdot \mathbf{A}^{(k)}_{L+1,P(x)} + \widehat{\mathbf{E}}^{(k)} \right) = \sum_{k=1}^{\ell_{out}} \hat{\mathbf{E}}^{(k)}$$

. Therefore, if $f(x) = y$, the output will be a low-norm matrix

Security. The simulator simulates the VBB obfuscation of $\mathsf{CC}[P, y]$ as follows:

- For $j = 0$, the simulator samples a uniform random matrix $\mathbf{B}_i^{(k)}$.
- For each layer, the simulator samples the encodings $\mathbf{C}_{i,b}^{(k)}$ from a low-norm gaussian distribution. (This follows from the properties of $\mathsf{TrapPreSamp}()$ algorithm.

Remark 9.4 *It should be noted that the simulator doesn't place any constraints similar to Equation 9.3. This is because w.h.p, the output of* $\mathsf{CC}[P, y](x) = 0$. *Intuitively, this means that in most scenarios, this simulation would be indistinguishable from the actual VBB obfuscation since the output of obfuscated program would not be a low-norm matrix.*

The security argument proceeds as a sequence of hybrids. Intuitively, the hybrids proceed as follows:

- *Hybrid 0:* This hybrid corresponds to the encoding generated in the VBB Obfuscation algorithm.
- *Hybrid 1:* Instead of sampling the matrices $\{\mathbf{A}_{L+1,y_k}^{(k)}\}_{k\in[\ell_{out}]}$ satisfying equation 9.3, the matrices are sampled from a uniform random distribution. This hybrid is statistically indistinguishable from the previous hybrid from *Leftover Hash lemma*. Notably, we can write equation 9.3 as:

$$\mathbf{A}_{L+1,y_{\ell_{out}}}^{\ell_{out}} = -\sum_{k=1}^{\ell_{out}-1} \mathbf{A}_{L+1,y_k}^{(k)} = \sum_{k=1}^{\ell_{out}-1} y_k(\mathbf{A}_{L+1,0}^{(k)} - \mathbf{A}_{L+1,1}^{(k)}) - \mathbf{A}_{L+1,0}^{(k)}$$

  We can define a corresponding hash function

$$h(y_0, \ldots, y_{\ell_{out}-1}) = \sum_{k=1}^{\ell_{out}-1} y_k(\mathbf{A}_{L+1,0}^{(k)} - \mathbf{A}_{L+1,1}^{(k)}) - \mathbf{A}_{L+1,0}^{(k)}$$

  The output of this hash function can be argued to be statistically indistinguishable from random (given that $(y_0, \ldots, y_{\ell_{out}-1})$ are sampled randomly) by invoking *Leftover Hash Lemma*.

- *Hybrid (2,0,k):* replace the final layer $\mathbf{H}_{L,b}^{(k)}$ with a uniformly sampled matrix. The indistinguishability follows from hardness of LWE.
- *Hybrid (2,1,k)* Instead of generating $\mathbf{H}_{L,b}^{(k)} \leftarrow \mathbb{Z}_q^{*\times *}$ and using Trapdoor Pre-Sampling algorithm to generate $\mathbf{C}_{L,b}^{(k)}$, the simulator samples $\mathbf{C}_{L,b}^{(k)}$ from a low-norm gaussian distribution. The indistinguishability of this hybrid follows from the properties of Trapdoor-Sampling algorithm

The argument behind hybrid (*Hybrid (2,0,k)*, *Hybrid (2,1,k)*) can be repeated for each layer from $L \to L-1 \to L-2 \to \ldots \to 1$.