CS 7880 Graduate Cryptography September 24, 2015

Lecture 4: Leftover Hash Lemma and One Way Functions

Lecturer: Daniel Wichs *Scribe: Biswaroop Maiti*

1 Topics Covered

- Finish Proof of Leftover hash Lemma
- Computational Model for Adversary
- One Way Functions and One Way Puzzles

2 Recall, From Last Time...

We recall some definitions and a claim proved in our previous lecture. These will be required to finish the proof for the Leftover Hash Lemma.

DEFINITION 1 $\mathbf{H}_\infty(X) = -\log\left(\max_x \Pr\left[X = x\right]\right)$ ♢

DEFINITION 2 A function $\text{Ext} : \mathcal{U} \times \mathcal{S} \to \mathcal{V}$ is a (k, ε) extractor if for all random variables X with $\mathbf{H}_\infty(X) \geq k$, we have:

$$\mathsf{SD}\left[\left(S, \text{Ext}(X, S)\right), (S, V)\right] \leq \varepsilon$$

where S is uniformly distributed over $\mathcal{S}$ and V is uniformly distributed over $\mathcal{V}$. ♢

DEFINITION 3 A function $H : \mathcal{U} \times \mathcal{S} \to \mathcal{V}$ is a universal hash function if $\forall x \neq x' \in \mathcal{U}$:

$$\Pr\left[\text{H}(x, S) = \text{H}(x', S)\right] = \frac{1}{|\mathcal{V}|}$$

We denote the collision of the hashed elements x, x' by the $*$ operator. ♢

Claim 1 *Let Z be a r.v. over $\mathcal{W}$ and Z' be an identical copy of Z. Define*

$$\text{Col}(Z) := \Pr[Z = Z'] = \sum_{z \in \mathcal{W}} \Pr[Z = z]^2$$

If $\text{Col}(Z) \leq \frac{1}{|\mathcal{W}|}(1 + 4\varepsilon^2)$ *then* $\mathsf{SD}(Z, W) \leq \varepsilon$ *where W is uniform on $\mathcal{W}$.*

3 Proof of Leftover Hash Lemma

Here we conclude the proof of the leftover hash lemma and discuss it in the context of extracting randomness for cryptographic schemes.

Theorem 1 *Every universal hash function is also a (k, ε)-extractor for*

$$k \geq \ell + 2\log(1/\varepsilon) - 2,$$

where $\ell = \log_2 |V|$.

Proof: Let X be a random variable with $\mathbf{H}_\infty \geq k$.

Apply Claim 1 from above with $Z = (S, H(X,S))$ and $\mathcal{W} = \mathcal{S} \times \mathcal{V}$, $Z' = (S', H(X', S'))$.

$$\begin{aligned}
\mathrm{Col}(Z) &= \Pr\left[Z = Z'\right] \\
&= \Pr\left[S = S', H(X,S) = H(X',S')\right] \\
&= \Pr\left[S = S', H(X,S) = H(X',S)\right] && (\text{ Since } S = S') \\
&= \Pr\left[S = S'\right] \cdot \Pr\left[H(X,S) = H(X',S)\right] && (\text{Since the events are independent}) \\
&= \frac{1}{|\mathcal{S}|}\left[\Pr\left[X = X'\right] + \Pr\left[H(X,S) = H(X',S) \wedge X \neq X'\right]\right] \\
& && (\text{Considers two events of collision, including the trivial case}) \\
&\leq \frac{1}{|\mathcal{S}|}\left(2^{-k} + \frac{1}{|V|}\right) && (\text{From the definition of min entropy}) \\
&= \frac{1}{|\mathcal{W}|}\left(1 + |V| \cdot 2^{-k}\right) \\
&= \frac{1}{|\mathcal{W}|}\left(1 + 2^{\ell - k}\right) \\
&= \frac{1}{|\mathcal{W}|}\left(1 + 4\varepsilon^2\right)
\end{aligned}$$

Therefore, by Claim 1, we have: $\mathsf{SD}(Z, W) = \mathsf{SD}((S, H(X,S)),\ (S,V)) \leq \varepsilon$ where $W = (S,V)$ is uniform over $\mathcal{W}$. This gives us the statement of the leftover hash lemma. □

3.1 Discussion

Extractors allow us to convert any imperfect source of randomness X which has entropy but is not uniformly random, into a nearly perfect uniformly random output $Y = \mathsf{Ext}(X, S)$. To do so, we need to "invest" some uniformly random seed S. However, the output Y is close to uniformly random even given S, and therefore the randomness of Y must come from X.

In the cryptographic setting, we usually think of X as some secret randomness (e.g., a password or randomness collected by an operating system from events such as keystrokes,

mouse movement, etc.). The goal is to use this imperfect secret randomness to derive a uniformly random secret key Y to use in cryptosystems. Extractors allow us to accomplish this with the help of a truly random public seed S which is known to all parties including the attacker. In other words extractors allow us to take secret imperfect randomness X and public perfect (uniform) randomness S and output an (essentially) perfect random secret key $Y = \mathsf{Ext}(X, S)$. Even if the attacker knows S but not X, the key $Y = \mathsf{Ext}(X, S)$ can be used as a uniformly random secret key in any cryptosystem.

3.2 Extractors Beyond Leftover Hash Lemma

The Leftover Hash Lemma achieves essentially optimal tradeoff between the entropy k and the output ℓ with $\ell = k - 2\log(1/\varepsilon) - O(1)$. However, it the seed length which is the same as the length of the source X is sub-optimal. There has been much work trying to optimize this.

There has also been much work studying deterministic extractors (without a seed) in setting where we know more about the source X than just that it has entropy. For example, one interesting setting is where $X = (X_1, X_2)$ consists of two independent parts of length n each with entropy k_1, k_2. The goal is to construct an extractor without any additional seed such that $\mathsf{Ext}(X) = \mathsf{Ext}(X_1, X_2)$ is nearly uniform. It was a long-standing open problem to do this when $k_1 + k_2 \ll n$ and a recent breakthrough result from *this year* finally showed how to do this.

4 Computational Security

When it comes to perefectly or statistically secure encryption, Shannon's Theorem tells us that the key size has to be as large as the message size, which means that we can only encrypt some limited amount of data per key. Similarly, for message authentication, the secret key had to grow with the number of messages we want to authenticate. However, just because we know that schemes without these properties aren't perfectly/statistically secure does not mean that they are "broken" in practice. For example, we saw that distinguishing encryptions of different messages was possible when the key size was smaller than the message size, but a generic attack required enumerating all possible keys which is computationally infeasible with today's technology if the key is even, say, 128 bits long or so. We will formalize what it means for a cryptosystem to be secure against *computationally bounded* attackers and, for the rest of the class, we will study different properties that can be achieved in this setting but usually cannot be achieved with perfect/statistical security.

Computational Security at High Level. We'd like to say that a cryptosystem is secure if any adversary that's computationally efficient cannot "break" the cryptosystem. Usually, however, it will be possible for the attacker to break the cryptosystem with at least some small probability via a trivial attack such as simply guessing the secret key. So we really want to say that any *efficient* adversary can only break the cryptosystem with some *tiny* probability.

We could formalize this by defining an efficient adversary as one that runs in time, say, at most $t = 2^{128}$ and we could define *tiny* probability to be $\varepsilon < 2^{-128}$. This would capture

what we want in reality reasonably well, and actually this is essentially the goal that's desired in practice.

The main problem with the above is that it's hard to build a theory around exact values like t, ε. Exact run-times mean different things in different models of computation (Turing Machine vs JAVA) and even on different processors with different instruction sets. As in Algorithms classes or Theory of Computation classes, we know that analyzing exact run-times is cumbersome and often obscures the big picture. Therefore, we will define security in the asymptotic sense.

Asymptotic Security. We will have a parameter n, called the *security parameter*, which dictates the desired security level of the scheme. When we build cryptosystems, the security parameter n will be given as an input to the scheme and will dictate other parameters of the system, such as the length of the key, etc.

We define *efficient* algorithms as Turing Machines that run in polynomial time. We will require that the cryptosystems we build are polynomial time and that security holds against all polynomial time attackers. Since we usually define polynomial time to mean polynomial in the input size, we will give the security parameter 1^n in *unary* as an input to all algorithms. This is to ensure that polynomial time in the input size is polynomial in n. Since it is cumbersome to always write 1^n everywhere, we will eventually stop writing it and just assume this implicitly. The formal definition of polynomial time algorithms refers to *deterministic* algorithms. In cryptography we crucially rely on randomness (e.g., to pick secret key) and therefore we will need to define a notion of *probabilistic polynomial time* (PPT) below.

We define "tiny" probabilities as ones that are asymptotically smaller than every inverse polynomial. We call these *negligible* and give a formal definition below.

Polynomial, Negligible. For a function $f : \mathbb{N} \to \mathbb{N}$ we say that f is polynomially bounded, denoted $f(n) = \mathsf{poly}(n)$, if there exists some constant $c \in \mathbb{N}$ such that $f(n) = O(n^c)$. This means that there exists some $c, c', n_0 \in \mathbb{N}$ such that for all $n > n_0$ we have $f(n) \leq c' n^c$. Alternatively, we can write this as $f(n) = n^{O(1)}$.

For a function $\varepsilon : \mathbb{N} \to \mathbb{R}$ we that ε is negligible, denoted $\varepsilon(n) = \mathsf{negl}(n)$, if for all $f(n) = \mathsf{poly}(n)$ there exists some $n_0 \in \mathbb{N}$ such that for all $n > n_0$ we have $\varepsilon(n) \leq 1/f(n)$. Alternatively, this is equivalent to saying that for every constant c, we have $\varepsilon(n) = 1/\Omega(n^c)$ or that $\varepsilon(n) = 1/n^{\omega(1)}$.

Here are some useful properties of polynomial and negligible functions that are simple exercises.

If $f(n), f'(n) = \mathsf{poly}(n)$ and $\varepsilon(n), \varepsilon'(n) = \mathsf{negl}(n)$ and c is a constant, then:

- $f(n) \cdot f'(n) = \mathsf{poly}(n)$
- $f'(f(n)) = \mathsf{poly}(n)$
- $\varepsilon(n) + \varepsilon'(n) = \mathsf{negl}(n)$
- $f(n) \cdot \varepsilon(n) = \mathsf{negl}(n)$
- $\varepsilon(\sqrt[c]{n}) = \mathsf{negl}(n)$

Probabilistic Polynomial Time (PPT). Our cryptosystems and our adversary are going to be probabilistic Turing Machines. We model this as a Turing Machine which, in addition to its input tape, has an infinitely long "random tape" r. This means that it can use as many random bits as it wants during the computation (but of course it can only read at most as many bits of r as its run time).

For a probabilistic Turing Machine A we write

- $\mathrm{A}(x; r)$ to denote the outputs of A on input x with random tape containing r.
- $\mathrm{A}(x)$ to denote a random variable for the output of $\mathrm{A}(x; r)$ over a random r.

We define a probabilistic Turing Machine A to be *probabilistic polynomial time* (PPT) if the worst case runtime is polynomial i.e. $\exists f(n) = \mathrm{poly}(n)$ such that $\forall x \in \{0,1\}^n$ and $\forall r \in \{0,1\}^*$, $\mathrm{A}(x, r)$ runs in time $f(n)$.

Notation for Random Experiments. We will need to analyze complicated probability distributions and it will be too cumbersome to define all of the random variables and how they are related to each other. Instead we will define a *random experiment* by describing the process of how different values are sampled. If $\mathcal{X}$ is a set we write $x \leftarrow \mathcal{X}$ to denote that x is sampled uniformly at random from $\mathcal{X}$. If X is a probability distribution or a random variable, we write $x \leftarrow X$ to denote that x is sampled according to the distribution X. For example, if A is a probabilistic TM then we write $x \leftarrow \mathrm{A}(z)$ to denote that x is sampled according to the distribution $\mathrm{A}(z)$.

From now on, we will express statements on probability of events using the notation

$$\Pr[< \text{Event} >:< \text{Random Experiment} >]$$

where the random experiment describes how various values are sampled. For example we can write

$$\Pr[x = y \ : \ x \leftarrow \{0,1\}^n, y \leftarrow \{0,1\}^n] = 2^{-n}$$

to denote the probability of the event $x = y$ when x, y are chosen uniformly at random from $\{0,1\}^n$.

5 One Way Functions

Before we define computational security for encryption and other useful cryptographic primitives, we start with a very simple primitive called a *one-way function*. We will see that every interesting cryptographic primitive will, at the very least, implicitly require us to construct a one-way function. Therefore, if we have any hope of building interesting cryptographic primitives, we need to understand one-way functions first.

Intuitively, a one-way function is easy to compute but hard to invert.

Definition 4 A deterministic function $f : \{0,1\}^* \to \{0,1\}^*$ is a one way function if it satisfies the following.

- Easy to compute: f can be computed in polynomial time

- Hard to invert: For all PPT Turing Machines A there exists a function $\varepsilon(n) = \mathsf{negl}(n)$, such that:

$$\Pr\left[f(x') = y : x \leftarrow \{0,1\}^n, y = f(x), x' \leftarrow \mathrm{A}(1^n, y)\right] \leq \varepsilon(n)$$

$\diamondsuit$

Remarks. The above says that any potential PPT inverter A that is given $f(x)$ must fail to come up with a pre-image $x' \in f^{-1}(f(x))$ with all but negigible probability. The probability above is over a random choice of x and the randomness of the algorithm A. Note that we do not require A to come up with the original value x but any pre-image of $f(x)$ will do. This means that an inefficient algorithm $\mathrm{A}(1^n, y)$ could always succeed with probability 1 by trying all possible values $x' \in \{0,1\}^n$ and outputting the first one such that $f(x') = y$. Alternatively, there is also an efficient algorithm $\mathrm{A}(1^n, y)$ that succeeds with probability 2^{-n} by picking a random $x' \leftarrow \{0,1\}^n$ and outputting it.

Lastly, we mention that it is crucial that we give 1^n to A to make sure it can run in polynomial time in n as otherwise the definition could be met with trivial (cryptographically uninteresting) functions. For example consider the function $f(x) = |x|$ which outputs the length of x in binary. In particular, if $|x| = n$ then $|f(x)| = \log n$ where $|\cdot|$ denotes the length of a string. Intuitively, this function is very easy to invert since any string of length n is a good inverse. However, an algorithm A that gets as input only $f(x)$ but not 1^n and needs to run in time polynomial in the size of its input $|f(x)| = \log n$ does not even have enough time to write down a pre-image x of length n.

5.1 A Simple Application of OWFs

It might not be immediate that the notion of one way functions has any connection with encryption of a message or any form of cryptography. As we will see, every interesting cryptographic primitive will, at the very least, implicitly require us to construct a one-way function. Therefore, one-way functions are *necessary* for cryptography. We will also later see that one-way functions are *sufficient* for much of symmetric-key cryptography. In other words, given a one-way function we can use it to construct many interesting cryptosystems. However, these constructions are fairly complex and not very direct.

Is there something interesting that we can do with one-way functions directly? A very simple use can be elaborated as follows. Suppose you have a number of websites to login through a password x, which we model as a uniformly random n bit string. We cannot be certain whether these websites are safe to keep the password x, perhaps they may be hacked. We want to make sure that, even if one website is hacked, the attacker does not learn enough information to log in to the other websites. A solution to this problem is to compute a one way function $y = f(x)$ and only store the output y on every website. Every time, a user wants to log in to a wesbite, he sends x and the website checks that $f(x) = y$. Even if some of the websites are hacked and the adversary learns y, this will not let him compute any value x' such that $f(x') = y$ which would let him log in to the other websites.

5.2 One Way Functions and Complexity

It's easy to see that if $P = NP$ then one-way functions don't exist. This is because we can efficiently verify if a candidate solution x' is a pre-image $f(x') = y$ and therefore if $P = NP$ we can also efficiently find x'.

However, we do not know if $P \neq NP$ is sufficient to imply the existence of one-way functions. In particular, we seem to need more than just $P \neq NP$ to get one-way functions. On a high level, we now show that one-way functions are equivalent to saying that there are NP problems that are hard on average (whereas $P \neq NP$ only says that there are problems that are hard in the worst-case) and we can sample a random problem along with a solution. To elucidate further, we show an alternate, slightly non-standard view of one way functions by defining the concept of a *one-way NP puzzle* and showing that it is equivalent to OWFs.

DEFINITION 5 **One Way NP Puzzle:** A one-way NP puzzle consists of a PPT Algorithm called PGen that generates the puzzle and its solution and a deterministic polynomial time algorithm PVer that can verify the solution.

$$(y, x) \leftarrow \text{PGen}(1^n)$$

$$\{0, 1\} \leftarrow \text{PVer}(y, x)$$

1. Correctness: $\Pr\left[\text{PVer}(y, x) = 1 : (y, x) \leftarrow \text{PGen}(1^n)\right] = 1$
2. Hard to Solve: For all PPT Turing Machine A, there exists a function $\varepsilon(n) = \mathsf{negl}(n)$ such that:

$$\Pr\left[\text{PVer}(y, x') = 1 : (y, x) \leftarrow \text{PGen}(1^n), x' \leftarrow A(1^n, y)\right] \leq \varepsilon(n)$$

♢

We can think of $\text{PGen}(1^n)$ as a puzzle generator which generates a puzzle y together with a solution x, and we think of $\text{PVer}(y, x)$ as puzzle verifier that checks if x is a valid solution to puzzle y (note that there may be many valid solutions to any puzzle). Given a random puzzle y sampled via $(y, x) \leftarrow \text{PGen}(1^n)$ it should be hard to find any valid solutions x'.

We can now show that one-way NP puzzles exist if and only if OWFs do.

5.2.1 OWF $\Rightarrow$ One-Way NP Puzzle

This is the easy direction. Given a one way function, we construct a one way puzzle. We define PGen and PVer as below, in terms of an OWF $f(\cdot)$:

$$\text{PGen}(1^n) : \{\text{choose } x \leftarrow \{0, 1\}^n ; \text{ output } f(x)\}$$

$$\text{PVer}(y, x) : \{\text{output } 1 \iff y = f(x)\}$$

5.2.2 One-Way NP Puzzle $\Rightarrow$ OWF

Given an One-Way NP Puzzle $(\text{PGen}, \text{PVer})$, we construct a one way function $f(\cdot)$. Recall that $\text{PGen}(1^n)$ is a PPT algorithm and we can write $\text{PGen}(1^n; r)$ to denote the execution of the algorithm with randomness r. Furthermore, since PGen is polynomial time, it only relies on randomness r of length at most $p(n)$ for some polynomial p. For a start, let's make a simplifying assumptions that $p(n) = n$ so that $\text{PGen}(1^n)$ only uses n bits of randomness. Then we can define a OWF $f(\cdot)$ as follows.

$$f(r) : \{ \text{ run } (y, x) \leftarrow \text{PGen}(1^n; r); \text{output } y\}$$

It's easy to see that this is a OWF since inverting this function is no easier than solving the puzzle. In particular, assume that PPT adversary A manages to invert $y = f(r)$ by finding r' such that $f(r') = y$ with probability ε. Then we can use A to solve the one-way puzzle y generated via $(y, x) \leftarrow \text{PGen}(1^n)$ by running $r' \leftarrow \text{A}(1^n, y)$, $(y', x') = PG(1^n; r')$ and outputting x'. Whenever A manages to invert correctly and $f(r') = y$ we know that $(y', x') = \text{PGen}(1^n; r')$ has $y' = y$ and, by the correctness property of the one-way puzzle, this means that $\text{PVer}(y, x') = 1$. Therefore the above attack solves the one-way puzzle with probability ε.

We made a simplifying assumption that $\text{PGen}(1^n)$ uses exactly n bits of randomness. In reality it may be that $\text{PGen}(1^n)$ uses any polynomial $p(n)$ bits of randomness. Since $p(n)$ is polynomial we known $p(n) \leq n^c$ for some constant c and all sufficiently large n. In this case we define

$$f(r) : \left\{\text{for } |r| = n \text{ run } (y, x) \leftarrow \text{PGen}(1^m; r) \text{ where } m = \sqrt[c]{n}; \text{output } y\right\}$$

In other words, we run $\text{PGen}(1^m; r)$ on a smaller security parameter $m = \sqrt[c]{n}$ to ensure that it only uses $p(m) \leq m^c \leq n$ bits of randomness for sufficiently large n. The same argument as above shows that this is a OWF where we rely on the fact that if $m = \sqrt[c]{n}$ then $\mathsf{negl}(m) = \mathsf{negl}(n)$.

More on OWFs and Complexity. Next time we will continue the discussion of where one-way functions fit in the complexity landscape and the relation between one-way functions and P vs NP.