CS 7880 Graduate Cryptography October 20, 2015

Lecture 11: Hash Functions, Merkle-Damgaard, Random Oracle

Lecturer: Daniel Wichs *Scribe: Tanay Mehta*

1 Topics Covered

- Review Collision-Resistant Hash Functions
- Merkle-Damgaard Theorem
- Random Oracle Model
- Number Theory and Algebra

2 Collision-Resistant Hash Functions

Recall the following definition from last time.

A family of functions $H_s : \{0,1\}^{l(n)} \to \{0,1\}^n$, where $l(n) > n$ and $s \in \{0,1\}^n$ are *collison-resistant hash functions* iff

- $H_s(x)$ can be computed in polynomial time.
- $\Pr[(x \neq x') \wedge (H_s(x) = H_s(x')) \ : \ s \leftarrow \{0,1\}^n, (x, x') \leftarrow A(s)] \leq \mathsf{negl}(n)$

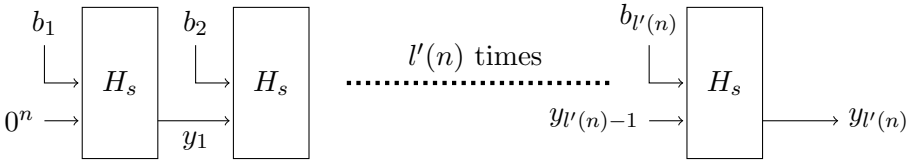
Note that the seed s is public to all parties; there is no secret key. Given access to all this information, an adversary should still be unable to find collisions.

Collision-resistant hash functions seem to be harder to construct than the other cryptographic primitives we've seen so far and we don't know how to build them from one-way functions. In fact, there is a result that shows that a collision-resistant hash function cannot be built from one-way functions in a blackbox manner.

Ideally, we want a high amount of compressability from collision-resistant hash functions (i.e. $l(n) >> n$). This is similar to how pseudorandom generators where we wanted a high amount of stretch (number fo extra pseudorandom bits we get). The following theorem shows that if we can compress by even 1 bit then we can compress by any number of bits.

Theorem 1 (Merkle-Damgaard) *If there exists a family of collision-resistant hash function H_s with $l(n) = n + 1$ bit input, then there exists a collision-resistant hash function H'_s with any polynomial $l'(n)$ bit input.*

Proof: We prove this by giving an explicit construction for H'_s. Let $(b_1, \ldots, b_{l'(n)})$ be the input to H'_s. Then, H'_s can be found by chaining together $l'(n)$ instances of H_s in the following manner.



Each H_s takes $n+1$ bit input. We initially feed an n-bit 0 string and the first bit of our input b_1 into one instance of H_s. Then, we take it's output y_1, which is of n bits, and feed it into another H_s instance along with the second bit of our input b_2. We do this process $l'(n)$ times until we have fed all bits of our input. We take the output of from this instance $y_{l'(n)}$ as the value of our hash.

We will now prove security for this construction. Assume that there is a PPT attacker $A(s)$ on the collision resistance of H'_s. In other words, with some probability $\varepsilon(n)$, the attacker $A(s)$ outputs

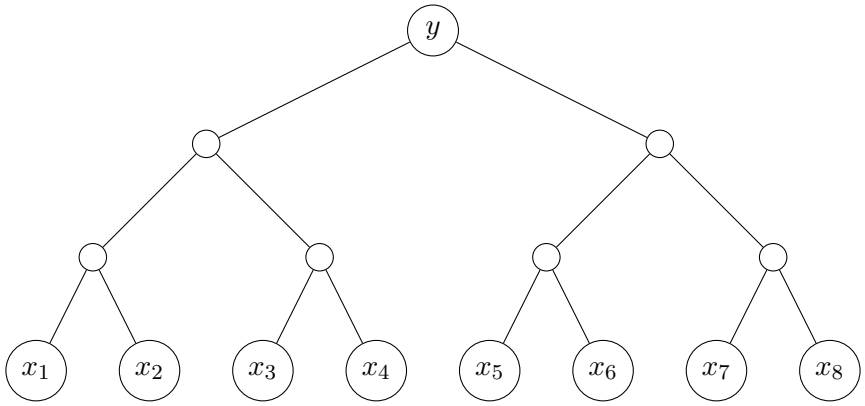
$$x = (b_1, \ldots, b_{l'(n)}) \neq x' = (b'_1, \ldots, b'_{l'(n)})$$

such that $H'_s(x) = H'_s(x')$. We claim that whenever the above happens, we can use it to find a collision on H_s. Running backwards on two instances of the above construction (one with input x and the other with x'), there must be a largest index j such that $(b_j, y_{j-1}) \neq (b'_j, y'_{j-1})$. Since this is the larges such j, we know that $H_s(b_j, y_{j-1}) = y_j = H_s(b'_j, y'_{j-1})$. Therefore this allows us to efficiently find a collision on H_s whenever A finds a collision on H'_s. Since the former can only happen with negligible probability, this shows that $\varepsilon(n)$ is negligible and therefore H'_s is collision resistant. □

Note that the above allows us to get a collision-resistant hash function for an arbitrarily large $l(n)$ bit input. But still, the input size $l(n)$ needs to be fixed. Suppose we want to create a collision-resistant hash function $H_s : \{0,1\}^* \to \{0,1\}^n$ with variable input length. Then, the proof above doesn't work anymore because x and x' can be of different length. It turns out that there are relatively simple ways to fix this and get a hash function with variable input length as well.

2.1 Merkle Trees

Let us consider a different construction, *Merkle trees*. We start with a hash function $H_s : \{0,1\}^{2n} \to \{0,1\}^n$. Now consider the following complete binary tree of depth h (illustrated for $h = 3$):

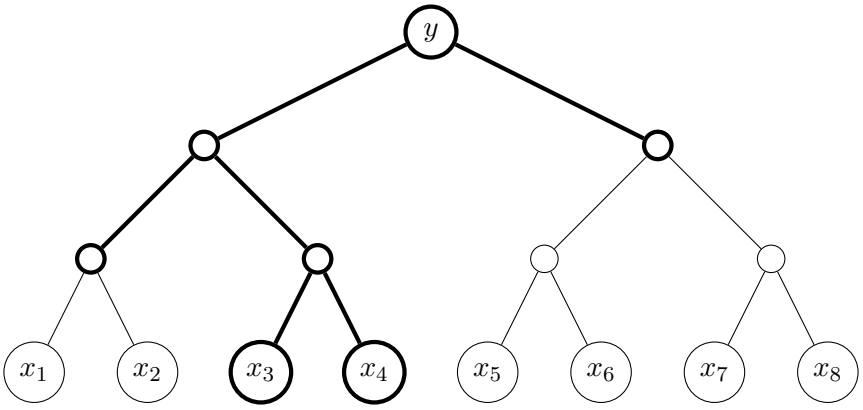


For n-bit blocks $x_1, \ldots, x_{2^h}$, we define the host (parent) of two nodes to have value $H_s(a, b)$ where a and b are the values of the host's two children. Then, the root node has value $y = H'_s(x_1, \ldots, x_{2^h})$ where $H'_s : \{0,1\}^{2^h} \to \{0,1\}^n$.

The proof of security is similar to the one for the Merkle-Damgaard construction. If an adversary comes up with a collision on the function H'_s, then we can use it to find a collision on H_s.

Merkle trees have a couple of useful properties. First, Merkle trees are more efficient to compute when working in parallel processing system than the Merkle-Damgaard construction. This is because all of the hashes at any level of the tree can be computed in parallel.

Second, let's say Alice hashes some long string x and sends the output $y = H_s(x)$ to Bob. Since the hash function is collision resistant, this commits Alice to a single string x which she can send as the pre-image of y. Let's say that Bob later wants to know the i'th block of x where we think of $x = (x_1, \ldots, x_l)$ as having $l = 2^h$ blocks with $x_i \in \{0,1\}^n$. Bob wants to be sure that there is only a single value of x_i that Alice can legitimately send. One way to do this would be for Alice to send the entire string x, and for Bob to very that $H_s(x) = y$ matches the hash he has. But this requires a lot of communication if Bob only wants x_i and doesn't care about the rest of x. The nice thing about the Merkle tree construction is that we can do this with much less communication. Instead of Alice sending the entire string x, she will only send the i'th block x_i along with the hash outputs corresponding to all the sibling nodes along the path from the root of the tree to the i'th bloc. For example, when $i = 3$, the transmitted values are illustrated by the nodes in bold below. Bob can the use the transmitted values to re-compute the value associated with the root of the tree and verify that this matches y.



This has applications to memory checking. If you have a hash of a large database consisting of l records with n bits each (think of l as very large and n as reasonably small) and want to download one small record, you can use this method to check that the downloaded record is correct. The communication is only $n \log l$ bits.

3 Random Oracle Model

Hash functions are used in many different ways in cryptography beyond only for "collision resistance". There is a belief that practical hash functions have many security properties which aren't captured by collision resistance alone.

To capture this intuition, we consider an idealized model of hash functions called the *random oracle model*. In this model, we think of a hash function as a completely random but public function $RO(\cdot)$. Anybody can query this function at arbitrary inputs x and get back $RO(x)$. This means that when we construct a scheme in the random oracle model, we allow the scheme to make such random oracle queries. However, we must also allow the attacker to make such queries as well when attacking the scheme.

Defining the Random Oracle Model. More formally, the *random oracle model* is a model where all parties (e.g. algorithms, adversaries) have oracle access to a (uniformly) random function

$$\mathrm{RO} : \{0,1\}^* \to \{0,1\}^n$$

(we can be flexible about the output length, but for now let's just insist on n bits.)

We can think of this as whenever a fresh value x is queried, the oracle chooses a random output y. The next time that x is queried, the oracle gives back the same y as previously. Note that if two different parties query RO on the same input x, they will receive the same output.

We say that a cryptographic scheme (e.g., PRG, PRF,encryption, etc.) is secure in the random oracle model if it satisfies the standard syntax, correctness and security properties but augmented so that the scheme as well as the attacker in the security definition have oracle access to RO. We give some examples below.

Constructions in the Random Oracle Model. Let us consider how to build various cryptographic primitives in this model.

- Collision-resistant hash functions: We can simply define our CRHF to be

$$H(x) := \mathrm{RO}(x)$$

We claim that

$$\Pr[A^{\mathrm{RO}(\cdot)} \to (x, x') \ : \ x \neq x', \mathrm{RO}(x) = \mathrm{RO}(x')] \leq \frac{q^2}{2^n} = \mathrm{negl}(n)$$

where q is the number of queries that A makes to RO. We can prove this claim unconditionally without assuming that $\mathsf{P} \neq \mathsf{NP}$, and allowing A to run in any amount of time. We will only make a restriction on the number of queries A can make.

Proof: Note that the probability that the i-th and j-th queries collide is $\frac{1}{2^n}$. Then by the union bound, we have that probability of any collision is $\frac{q^2}{2^n}$. This is also known as the Birthday Paradox. Without loss of generality, assume that $x_i \neq x_j$ and A calls RO on x_i. □

- Pseudorandom generators: We define our PRG to be

$$G^{\mathrm{RO}}(x) = \mathrm{RO}(x||1), \mathrm{RO}(x||2)$$

where $x||1$ and $x||2$ are x concatenated with bit-representations of 1 and 2 respectively. Note that each output of each RO is n bits yielding a total of $2n$ bits. The only way to distinguish between this PRG and the uniform distribution is by calling RO on x. The probability of successfully guess x is $\frac{1}{2^{|x|}}$.

- Pseudorandom functions: We initially may think to just define our functions as

$$F_k(x) = RO(x)$$

However, this doesn't work because the adversary A has access to $\mathrm{RO}(x)$. There is no secret key used. Instead, consider

$$F_k(x) = \mathrm{RO}(k||x)$$

The security of this definition follows from considering

$$|\Pr[A^{\mathrm{RO}(\cdot), F_k(-)}(1^n) = 1] - \Pr[A^{\mathrm{RO}(\cdot), R(\cdot)}(1^n) = 1]| \leq \mathrm{negl}(n)$$

where R is a random function independent of RO.

Discussion. As we have seen, it is fairly easy to build cryptographic primitives in this model. The model is mathematically precise and we can give formal definitions and proofs of security in this model. But what does it tell us about the real world, where there is no random oracle?

The hope is that we take a construction which is secure in the random oracle model, and replace the random oracle $RO(\cdot)$ with some real cryptographic hash function $H_s(\cdot)$ (where the seed is chosen randomly but given to the adversary) then the schemes will still be secure. This is not a well defined security property for the hash function $H_s(\cdot)$. We are simply hoping that H_s is "as good" as a random oracle even though we know that H_s is not a truly random function (it has a short description and can be evaluated in polynomial time). We do not have a theorem saying that if a scheme is secure in the random oracle model than it is secure when we replace the random oracle with a real hash function H_s. It's therefore only a heuristic way of arguing about security. It seems to work in real life and allows us to construct very simple schemes. But we know that it does not work in general and can lead to incorrect incorrect conclusions. In fact, we know that there are (contrived) schemes which are secure in the random oracle model, but are always insecure when you replace the random oracle by *any potential* hash function $H_s(\cdot)$.

On the other hand, we can think of the random oracle model as defining the limits of what we can even hope to construct using (only) tools like one-way function or collision-resistant hash functions. If there is a primitive that we can't even construct in the Random Oracle model, then there is no hope to construct it (in a black-box way) from any of the primitives we've studied so far. An in fact, it turns out that there is such a primitive: public-key encryption. A beautiful result by Impagliazzo-Rudich shows that we can't build public key encryption in the random oracle model (without making some other hardness assumptions). Intuitively, public-key encryption requires hard problems that have more structure which the random oracle does not provide. Luckily, we will see (candidates) for such hard problems in number theory and algebra.

4 Number Theory and Algebra

We now move onto a new part of the course. We're going to build the primitives we've been studying (OWFs, PRGs, PRFs) using hardness assumptions from number theory and algebra. These constructions are never used in practice since they're far less efficient than the "ad-hoc" constructions out there. We will also go beyond the primitives we've seen so far and show how to build more advanced cryptosystems, most importantly public-key encryption. These are the schemes used in practice since there aren't more efficient "ad-hoc" alternatives.

We state some facts of number theory and algebra.

Modular arithmetic: Consider the group over the set $\mathbb{Z}_n$ (integers modulo n). The canonical representatives of $\mathbb{Z}_n = \{0, \ldots, n-1\}$.

Along with the operation $+$, $(\mathbb{Z}_n, +)$ forms a group. This means we satisfy

1. Identity: There exists $0 \in \mathbb{Z}_n$ such that for all $a \in \mathbb{Z}_n$ we have that $a + 0 = a$.

2. Add: For all $a, b \in \mathbb{Z}_n$, $a + b \in \mathbb{Z}_n$.

3. Inverse: For all $a \in \mathbb{Z}_n$, there exists $-a \in \mathbb{Z}_n$ such that $a + (-a) = 0$.

Note that $(\mathbb{Z}_n, \cdot)$ does not form a group because we lack multiplicative inverses (i.e. we cannot divide every element by another to get 1.

Fact: $a \neq 0$ has inverse in $(\mathbb{Z}_n, \cdot)$ if $gcd(a, n) = 1$.

Proof: This is equivalent to saying there exists x, y such that $xa + yn = 1$. From this, we have

$$xa \equiv 1 \pmod{n} \Rightarrow x \equiv a^{-1} \pmod{n}$$

□

Consider $\mathbb{Z}_n^* = \{a \in \{1, \ldots, n\}\}$ such that $gcd(a, n) = 1$. This forms a group under multiplication since all elements have inverses.

What is the size of this group? Define the Euler totient function $\varphi(n) = |\mathbb{Z}_n^*|$.

If p is prime, then $\mathbb{Z}_p^* = \mathbb{Z}_p \backslash \{0\} = \{1, \ldots, p-1\}$. Therefore, $\varphi(p) = p - 1$.

Let us now look at the computational complexity of various algebraic operations.

- $+, \cdot$ are polynomial time. $+$ is, in fact, linear.
- $(-)^{-1} \pmod{n}$ computing the inverse. Existence of the inverse can be found using Euclid's algorithm. The actual inverse can be found using the extended Euclidean algorithm. Both of these run in polynomial time.
- a^b exponentiation cannot be computed in polynomial time because the output is exponential in the input length.
- $a^b \pmod{n}$ can be computed in polynomial time. Write b in binary.

$$\begin{aligned} a^b &\equiv a^{\sum b_i \cdot 2^i} \pmod{n} \\ &\equiv \prod_i a^{b_i \cdot 2^i} \\ &\equiv \prod_{i, b_i = 1} a^{2^i} \pmod{n} \end{aligned}$$

The final value can be computed by repeated squaring i times.