Lecture 7: Hash functions and random oracles

Boaz Barak

We have seen pseudorandom generators, functions and permutations, as well as Message Authentication codes, CPA and CCA secure encryptions. This week we will talk about cryptographic hash functions and some of their magical properties. We motivate this by the *bitcoin* cryptocurrency. As usual our discussion will be highly abstract and idealized, and any resemblance to real cryptocurrencies, living or dead, is purely coincidental.

The "bitcoin" problem

Using cryptography to create a *centralized* digital-currency is fairly straightforward, and indeed this is what is done by Visa, Mastercard etc.. The main challenge with bitcoin is that it is *decentralized*. There is no trusted server, and there are also no "user accounts" but rather this is a collection of anonymous and autonomous parties that somehow need to agree on what is a valid payment. The basic unit in the system is a *coin*. Each coin has a unique identifier, and a current *owner* .[1] Transactions in the system have either the form of "mint coin with identifier ID and owner P" or "transfer the coin ID from P to Q".

Since there are no user accounts in bitcoin, the "entities" P and Q are not identifier of any person or account. Rather one could think of them as simply "computational puzzles". A *computational puzzle* can be thought of as a string α that specifies some "problem" such that it's easy to verify whether some other string β is a "solution" for α, but it is hard to find such a solution on your own. (Students with complexity background will recognize here the class **NP**.) For example α can encode some large integer N, and a solution β will encode a pair of numbers A, B such that $N = A \cdot B$. Another more generic example (that you can keep in mind as a potential implementation for the puzzles we use here) is that α will be a string in $\{0,1\}^{2n}$ while β will be a string in $\{0,1\}^n$ such that $\alpha = G(\beta)$ where $G : \{0,1\}^n \rightarrow \{0,1\}^{2n}$ is some pseudorandom generator.[2]

[1] This is one of the places where we simplify and deviate from the actual Bitcoin system. In the actual Bitcoin system, the atomic unit is known as a *satoshi* and one bitcoin (abberviated BTC) is 10^8 satoshis. For reasons of efficiency, not every satoshi has an individual identifier and transactions can involve transfer and creation of multiple satoshis.

[2] There are reasons why Bitcoin uses digital signatures and not these puzzles. The main issue is that we want to bind the puzzle not just to the coin but also to the particular transaction, so that if you know the solution to the puzzle P corresponding to the coin ID and want to use

The real Bitcoin system typically uses puzzles based on *digital signatures*, a concept we will learn about later in this course, but you can simply think of P as specifying some abstract puzzle and every person that can solve P perform transactions on the coins owned by P. In particular if you lost the solution to the puzzle then you have no access to the coin, and if someone stole the solution from you, then you have no recourse or way to get your coin back. People have managed to lose millions of dollars in this way.

The bitcoin ledger

The main idea behind bitcoin is that there is a public *ledger* that contains an ordered list of all the transactions that were ever performed and are considered as valid in the system. Given such a ledger, it is easy to answer the question of who owns any particular coin. The main problem is how does a collection of anonymous parties without any central authority agree on this ledger? This is an instance of the *consensus* problem in distributed computing. This seems quite scary, as there are very strong negative results known for this problem; for example the famous Fischer, Lynch Patterson (FLP) result showed that if there is even one party that has a *benign* failure (i.e., it halts and stop responding) then it is impossible to guarantee consensus in an asynchronous network. Things are better if we assume synrhonicity (i.e., a global clock and some bounds on the latency of messages) as well as that a majority or supermajority of the parties behave correctly. The former assumption is typically fine on the Internet, but the latter seems quite suspicious. What does it mean a "majority of parties" in an anonymous network where a single person can create multiple "entities" and cause them to behave arbitrarily ("byzantine" faults in distributed- parlance)? Also, why would we assume that even one party would behave honestly- if it pays to cheat then they would, wouldn't they?

Perhaps the main idea behind bitcoin is that "majority" will correspond to a "majority of computing power", or as the original bitcoin paper says "one CPU one vote" (or perhaps more accurately, "one cycle one vote"). It might not be immediately clear how to implement this, but at least it means that creating fictitious new entities (sometimes known as a "Sybill" attack after the movie about multiple-personality disorder) cannot help. To implement it we turn to a cryptographic concept known as "proof of work" which was originally suggested by Dwork and Naor in 1991 as a way to combat mass mail email.[3]

Consider a pseudorandom function $\{f_k\}$ mapping n bits to ℓ bits. On average, it will take a party Alice 2^ℓ queries to obtain an input x such that $f_k(x) = 0^\ell$.

that to transfer it to Q, it won't be possible for someone to take your solution and use that to transfer the coin to Q' before your transaction is added to the public ledger. We will come back to this issue after we learn about digital signatures.

[3] This was a rather visionary paper in that it foresaw this issue before the term "spam" was introduced and indeed when email itself was hardly widespread.

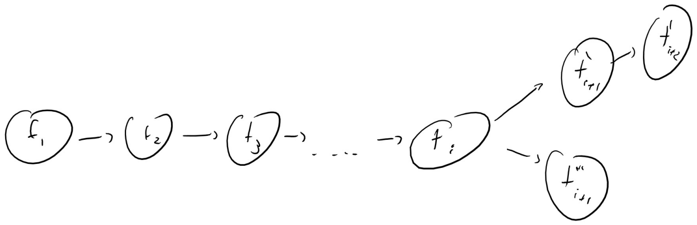

Figure 1: The bitcoin ledger consists of an ordered list of transactions. At any given point in time there might be several "forks" that continue the ledger, and different parties do not necessarily have to agree on them. However, the bitcoin architecture is designed to ensure that the parties corresponding to a majority of computing pwoers will reach consensus on a single ledger.

So, if we're not too careful, we might think of such an input x as a *proof* that Alice spent 2^ℓ time…

However, the question is who is holding the key k for the pseudorandom function? If there is a trusted server holding the key, then sure, such an input x would take on average 2^ℓ queries, but the whole point of bitcoin is to *not* have a trusted server. If we give k to Alice, then can we guarantee that she can't find a "shortcut" to find such an input without running 2^ℓ queries? The answer, in general, is **no**:

Exercise: Show (under the PRF conjecture) that there exists a PRF $\{f_k\}$ mapping n bits to n bits and an efficient algorithm A such that $A(k) = x$ such that $f_k(x) = 0^\ell$.

However, suppose that $\{f_k\}$ was somehow a "super-strong PRF" that would behave like a random function *even to a party that holds the key.* In this case, we would imagine that it would not be feasible to obtain x such that $f_k(x) = 0^\ell$ using less than 2^ℓ cycles, and so sucn an x would serve as a "proof of work" that you've spent 2^ℓ cycles. By adjusting ℓ we would obtain a proof of spending T cycles for a value T of our choice. Now if things would go as usual in this course then I would state a result like the following:

Theorem: Under the PRG conjecture, there exist super strong PRF.

Unfortunately this result is *not* known to be true, and for a very good reason. Most natural ways to define "super strong PRF" will result in properties that can be shown to be *impossible to achieve.* Nevertheless, the intuition behind it still seems useful and so we have the following heuristic:

The random oracle heuristic (aka "Random oracle model", Bellare-Rogaway 1993): If a "natural" protocol is secure when all parties have access to a random function $H : \{0,1\}^n \to \{0,1\}^\ell$, then it remains secure even when we give the parties the *description* of a cryptographic hash function with the same input and outuput lengths.

We don't have a good characterization as to what makes a protocol "natural" and we do have fairly strong counterexamples to this heuristic (though they are arguably "unnatural"). That said, it still seems useful as a way to get intuition for security, and in particular to analyze bitcoin (and many other practical protocols) we do need to assume it, at least given current knowledge.

> **Important caveat:** *The random oracle heuristic is very different from all the conjectures we considered before. It is **not** a formal conjecture since we don't have any good way to define "natural" and we do have examples of protocols that are secure when all parties have access to a random function but are **insecure** whenever we replace this random function by **any** efficiently computable function (see the homework exercises).*

We can now specify the "proof of work" protocol for bitcoin. Given some identifier $ID \in \{0,1\}^n$ an integer $T \ll 2^n$ and a hash function $H : \{0,1\}^{2n} \to \{0,1\}^n$, the proof of work corresponding to ID and T will be some $x \in \{0,1\}^*$ such that the first $\lceil \log T \rceil$ bits of $H(ID \| x)$ are zero.[4]

From proof of work to consensus on ledger

How does proof of work help us in achieving consensus? The idea is that every transaction t_i comes up with a proof of work of some T_i time with respect to some identifier that is unique to t_i. The *length* of a ledger $(t_1, \ldots, t_n)$ is the sum of the corresponding T_i's which correspond to the total number of cycles invested in creating this ledger.

An honest party in the bitcoin network will accept the longest valid ledger it is aware of (where a ledger is valid if every transaction in it of the form "transfer the coin ID from P to Q" is preceded by a valid transaction that transferred or minted the coin ID to P). Now you can see that if a ledger L corresponds to the majority of the cycles that were available in this network, then every honest party would accept it, as any alternative ledger would be necessarily shorter. The question is then how do we get to that happy state given that many parties might be non malicious but still *selfish* and might not want to volunteer their computing power for the goal of creating a consensus ledger. Bitcoin achieves this by giving some incentive, in the form of the ability to mint new coins, to any party that adds to the ledger. This means that if we are already in the situation where there is a consensus ledger L, then every party has an interest in continuing this ledger L, and not any alternative, as they want their minting transaction to be part of the new consensus ledger. In contrast if they "fork" the consensus ledger then their work will be for vain. Thus one can hope that the consensus ledger will continue to grow. (This is a rather hand-wavy and

[4] The actual bitcoin protocol is slightly more general, where the proof is some x such that $H(ID \| x)$, when interpreted as a number in $[2^n]$, is at most T. There are also other issues about how exactly x is placed and ID is computed from past history etc.. that we ignore here.

imprecise argument, see this paper for a more in depth analysis; this is also related to the phenomenon known as preferential attachment.)

Cost to mine, mining pools: Generally, if you know that completing a T-cycle proof will get you a single coin, then making a single query (which will succeed with probability $1/T$) is akin to buying a lottery ticket that costs you a single cycle and has probability $1/T$ to win a single coin. One difference over the actual lottery is that there is also some probability that you're working on the wrong fork of the ledger, but this incentivizes people to avoid this as much as possible. Another, perhaps even more major difference, is that things are setup so that this is a *profitable* enterprise and the cost of a cycle is smaller than the value of $1/T$ coins. Just like in the lottery, people can and do gather in groups (known as "mining pools") where they pool together all their computing resources, and then split the award if they win it. Joining a pool doesn't change your expectation of winning but reduces the *variance*. In the extreme case, if everyone is in the same pool, then for every cycle you spend you get exactly $1/T$ coins. The way these pools work in practice is that someone that spent C cycles looking for an output with all zeroes, only has probability C/T of getting it, but is very likely to get an output that begins with $\log C$ zeroes. This output can serve as their own "proof of work" that they spent C cycles and they can send it to the pool management so they get an appropriate share of the reward.

> **The real bitcoin:** There are several aspects in which the protocol described above differs from the real bitcoin protocol. Some of them were already discussed above: Bitcoin typically uses digital signatures for puzzles (though it has a more general scripting language to specify them), and transactions involve a number of satoshis (and the user interface typically displayes currency is in units of BTC which are 10^8 satoshis). The Bitcoin protocol also has a formula designed to factor in the decrease in dollar cost per cycle so that bitcoins become more expensive to mine with time. There is also a fee mechanism apart from the minining to incentivize parties to add to the ledger. (The issue of incentives in bitcoin is quite subtle and interesting and not fully resolved, and it is possible that parties behavior will change with time.) The ledger does not grow by a single transaction at a time but rather by a *block* of transactions, and there is also some timing synchronization mechanism (which is needed, as per the consensus impossiblity results). There are other differences as well; see the Bonneau et al paper as well as the Tschorsch and Scheuermann survey for more.

Collision resistance and creating short "unique" identifiers

Another issue we "brushed under the carpet" is how do we come up with these unique identifiers per transaction. We want each transaction t_i to be *bound* to the ledger state $(t_1, \ldots, t_{i-1})$, and so the ID of t_i should contain also the ID's all the prior transactions. But yet we want this ID to be only n bits long. Ideally, we would like to find a *one to one* mapping H from $\{0,1\}^N$ to $\{0,1\}^n$ for some very large $N \gg n$. Then the ID corresponding to the task of appending t_i to $(t_1, \ldots, t_{i-1})$ would simply be $H(t_1 \| \cdots \| t_i)$. The only problem is that this is of course clearly impossible- 2^N is *much* bigger than 2^n and there is no one to one map from a large set to a smaller set. Luckily we are in the magical world of crypto where the impossible is routine and the unimaginable is occasional. So, we can actually find a function H that is "essentially" one to one.

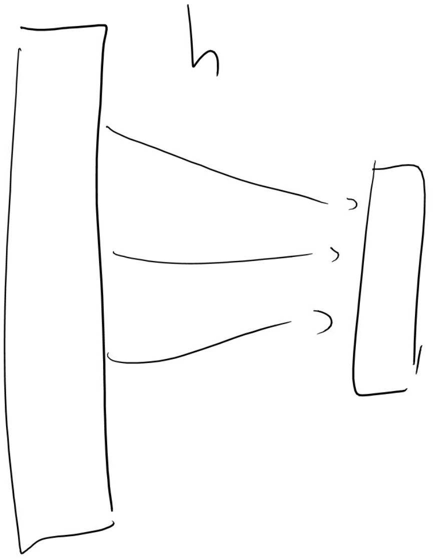

Figure 2: A collision-resistant hash function is a map that from a large universe to a small one that is "practically one to one" in the sense that collisions for the function do exist but are hard to find.

The main idea is the following simple result:

Lemma (One side of birthday bound): If H is a random function from some domain S to $\{0,1\}^n$, then the probabilisty that after T queries an attacker finds $x \neq x'$ such that $H(x) = H(x')$ is at most $T^2/2^n$.

Proof: Let us think of H in the "lazy evaluation" mode where for every query the adversary makes, we choose a random answer in $\{0,1\}^n$ at the time it is made. (We can assume the adversary never makes the same query twice since a repeat query can be simulated by repeating the same answer.) For $i < j$ in $[T]$ let $E_{i,j}$ be the event that $H(x_i) = H(x_j)$. Since $H(x_j)$ is chosen at random and independently from the prior choice of $H(x_i)$, the probability of $E_{i,j}$ is 2^{-n}. Thus the probability of the union of $E_{i,j}$ over all i, j's is less than $T^2/2^n$, but this probability is exactly what we needed to calculate. QED

This means that a random function H is *collision resistant* in the sense that it is hard for an efficient adversary to find two inputs that collide. Thus the random oracle heuristic would suggest that a cryptographic hash function can be used to obtain the following object:

Definition: A collection $\{h_k\}$ of functions where $h_k : \{0,1\}^* \rightarrow \{0,1\}^n$ for $k \in \{0,1\}^n$ is a *collision resistant hash function (CRH) collection* if the map $(k,x) \mapsto h_k(x)$ is efficienty computable and for every efficient adversary A, the probability over k that $A(k) = (x,x')$ such that $x \neq x'$ and $h_k(x) = h_k(x')$ is negligible.[5]

Once more we do *not* know a theorem saying that under the PRG conjecture there exists a collision resistant hash function collection, even though this property is considered as one of the desiderata for cryptographic hash functions. Moreover, we do know how to obtain collections satisfying this condition under various assumptions that we will see later in the course such as the learning with error problem and the factoring and discrete logarithm problems. Furthermore if we consider the weaker notion of security under a *second preimage attack* (also known as being a “universal one way hash function” or UOWHF) then it *is* known how to derive such a function from the PRG assumption.

Practical constructions of cryptographic hash functions

While we discussed hash functions as *keyed* collections, in practice people often think of a hash function as having being a *fixed keyless function*. However, this is because most practical constructions involve some hardwired standardized constants (often known as IV) that can be thought of as a choice of the key.

Practical constructions of cryptographic hash functions start with a basic block which is known as a *compression function* $h : \{0,1\}^{2n} \rightarrow \{0,1\}^n$. The function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ is defined as $H(m_1,\ldots,m_t) = h(h(h(m_1,IV),m_2),\cdots,m_t)$ when the message is composed of t blocks (and we can pad it otherwise).

This construction is known as the Merkle-Damgard construction and we know that it does preserve collision resistance:

Theorem: From two messages $m \neq m' \in \{0,1\}^{tn}$ such that $H(m) = H(m')$ we can find two messages $x \neq x' \in \{0,1\}^{2n}$ such that $h(x) = h(x')$.

Proof: The intuition behind the proof is that if h was invertible then we could invert H by simply going backwards. Thus in principle if a collision for H exists then so does a collision for h. Now of course this is a vacuous statement since both h and H shrink their inputs and hence clearly have collisions. But we want to show a *constructive* proof for this statement that allows to transform a

[5] Note that the other side of the birthday bound shows that you can always find a collision in h_k using roughly $2^{n/2}$ queries. For this reason we typically want the output length of hash functions to be longer than standard cryptographic key length (e.g., 256 bits as opposed to 128 bits).

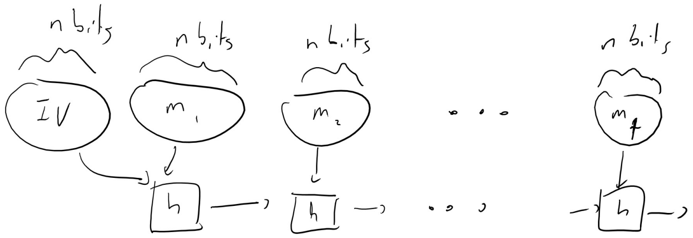

Figure 3: The Merkle-Damgard construction converts a compression function $h : \{0,1\}^{2n} \rightarrow \{0,1\}^n$ into a hash function that maps strings of arbitrary length into $\{0,1\}^n$. The transformation preserves collision resistance but does not yield a PRF even if h was pseudorandom. Hence for many applications it should not be used directly but rather composed with a transformation such as HMAC.

collision in H to a collision in h. This is very simple. We look at the computation of $H(m)$ and $H(m')$ and at the first block in which the inputs differ but the output is the same (there must be such a block). This block will yield a collision for h. QED

Unfortunately, the Merkle-Damgard construction is *not* a PRF even when IV is random and secret. This is because we can perform a *length extension attack* on it. Even if we don't know IV, given $y = H_{IV}(m_1, \ldots, m_t)$ and a block m_{t+1} we can compute $y' = h(y, m_{t+1})$ which equals $H_{IV}(m_1, \ldots, m_{t+1})$. One fix for this is to use a different IV' in the *end* of the encryption. That is, we define:

$H_{IV,IV'}(m_1, \ldots, m_t) = h(IV', H_{IV}(m_1, \ldots, m_t))$

A variant of this construction (where IV' is obtained as some simple function of IV) is known as HMAC and it can be shown to be a pseudorandom function under some pseudorandomness assumptions on the compression function h. It is very widely implemented. In many cases where I say "use a cryptographic hash function" in this course I actually mean to use an HMAC like construction that can be conjectured to give at least a PRF if not stronger "random oracle"-like properties.

The simplest implementation for a compression function is to take a *block cipher* with an n bit key and an n bit message and then simply define $h(x_1, \ldots, x_{2n}) = E_{x_{n+1},\ldots,x_{2n}}(x_1, \ldots, x_n)$. A more common variant is known as Davies-Meyer where we also XOR the output with $x_{n+1}, \ldots x_{2n}$. In practice people often use *tailor made* block ciphers that are designed for some efficiency or security concerns.

Some history

Almost all practically used hash functions are based on the Merkle-Damgard paradigm. Hash functions are designed to be extremely efficient[6] which also means that they are often at the "edge of insecurity" and indeed have fallen over the edge.

In 1990 Ron Rivest proposed MD4, which was already shown weaknesses in 1991, and a full collision has been found in 1995. Even faster attacks have been since found and this is considered completely insecure.

In response to these weaknesses, Rivest designed MD5 in 1991. A weakness was shown for it in 1996 and a full collision was shown in 2004. Hence it is now also considered insecure.

In 1993 the National Institute of Standards proposed a standard for a hash function known as the *Secure Hash Algorithm (SHA)*, which has quite a few similarities with the MD4 and MD5 functions. This function is known as SHA-0, and the standard was replaced in 1995 with SHA-1 that includes an extra "mixing" (i.e., bit rotation) operation. At the time no explanation was given for this change but SHA-0 was later found to be insecure. In 2002 a variant with longer output, known as SHA-256, was added (as well as some others). In 2005, following the MD5 collision, significant weaknesses were shown in SHA-1 and as a result it is considered insecure and SHA-256 is recommended.

Given these results, NIST started in 2006 a competition for a new hashing standard, based on functions that seem sufficiently different from the MD5/SHA-0/SHA-1 family. (SHA-256 is unbroken but it seems sufficiently close to those other systems.) The hash function Keccak was selected as the new standard SHA-3 in August of 2015.

The NSA and hash functions.

The NSA is the world's largest employer of mathematicians, and is very heavily invested in cryptographic research. It seems quite possible that they devote far more resources to analyzing symmetric primitives such as block ciphers and hash functions than the open research community. Indeed, the history above suggests that the NSA has consistently discovered attacks on hash functions before the cryptographic community (and the same holds for the differential cryptanalysis technique for block ciphers). That said, despite the "mythic" powers that are sometimes ascribed to the NSA, this history suggests that they are ahead of the open community but not so much ahead, discovering attacks on hash functions about 5 years or so ahead.

[6] For example, the Boneh-Shoup book quotes processing times of up to 255MB/sec on a 1.83 Ghz Intel Core 2 processor, which is more than enough to handle not just Harvard's network but even Lamar college's.

There are a few ways we can get “insider views” to the NSA’s thinking. Some such insights can be obtained from the Snowden documents. The Flame malware has been discovered in Iran in 2012 after operating since at least 2010. It used an MD5 collision to achieve its goals. Such a collision was known in the open literature since 2008, but Flame used a different variant that was unknown in the literature. For this reason it is suspected that it was designed by a western intelligence agency.

Another insight into NSA’s thoughts can be found in pages 12-19 of NSA’s internal Cryptolog magazine which has been recently declassified; one can find there a rather entertaining and opinionated (or obnoxious, depending on your point of view) review of the CRYPTO 1992 conference. In page 14 the author remarks that certain weaknesses of MD5 demonstrated in the conference are unlikely to be extended to the full version, which suggests that the NSA (or at least the author) was not aware of the MD5 collisions at the time.

Cryptographic vs non cryptographic hash functions:

Hash functions are of course also widely used for *non cryptographic* applications such as building hash tables and load balancing. For these applications people often use *linear* hash functions known as *cyclic redundancy codes (CRC)*. Note however that even in those seemingly non cryptographic applications, an adversary might cause significant slowdown to the system if he can generate many collisions. This can and has been used to obtain denial of service attacks. As a rule of thumb, if the inputs to your system might be generated by someone who does not have your best interests at heart, you’re better off using a cryptographic hash function.