Lecture 3 - Computational Security

Boaz Barak

February 8, 2010

Exponential is big Machine with concrete example.

Ramblings Defense of definitions... why probability is essential for secrecy.

Quick Review Perfect security, impossibility result, statistical security

Definition of statistical security: (E,D) is *ϵ-secure* if for every $A : \{0,1\}^* \to \{0,1\}$, x_0, x_1,

$$\left|\Pr[A(E_{U_n}(x_0)) = 1] - \Pr[A(E_{U_n}(x_1) = 1]\right| < \epsilon$$

Computational Security Unfortunately, we saw that statistical security does not allow us to really break the impossibility result.

We now turn to a closer examination of that impossibility result. In particular, in real life people *are* using encryption schemes with keys shorter than the message size to encrypt all sort of sensitive information including credit card numbers. Could we use the proof of the impossibility result to *break* these schemes and gain notoriety and fortune?

Indeed, the proof of the impossibility result does in fact give a *algorithm* to break any encryption scheme. It's even quite simple (10 lines of C code).

The only problem is that if the key is of size n, then this 10 line C program will run in time roughly 2^n. This is going to take quite a long time even for n that is not too large.

Consider a key that is 1Kbit long (this is not very large note that many cell phones have at least $128,000 * 8$Kbits memory, definition of broadband access is at least 256Kbits/sec). Even if we take Moore's law to its limit, and assume that we have placed a super-computer operating at the speed of light on every atom in the observable universe , we would still not be able to run 2^{1000} operations before the sun collapses. It's a safe bet that any credit cards we manage to steal will be expired by then...

This raises the idea of designing encryptions that are unbreakable within any reasonable time.

Computationally Secure Encryption The main problem we face is that, while the particular C program arising from that proof runs in exponential time, we don't have any guarantee that there is not another program that is actually efficient. In fact, we already saw this is the case for the substitution cipher, where the number of possibilities for the key is huge but still we can break the scheme efficiently.

Another problem is that we want a precise mathematical definition. That is, the previous perfect secrecy definition was a precise statement about the functions (E, D) that can be formulated and proven to hold using the tools of mathematics. We don't want a vague definition such as "breaking E is very hard" since we can't work with such a definition.

This means that we need to give a precise mathematical formulation to statements such as "the problem P can not be solved in reasonable time". However, this arises the question of *how* do we model the adversary's resources. The adversary may use a Windows, Mac, or Unix system, she may use a network of connected computers, she may use a super computer, or a special purpose computer she constructed just for this task, perhaps not made out of silicon but maybe out of analog or biological components, she can also use a mixture of computer and human intelligence, using say particularly gifted mathematicians to help break our encryption.

Can we give a mathematically precise definition that implies that a computational problem cannot be solved in say T years no matter what mixture of these and other resources are used?

It turns out the answer is "Yes". To do so, we need to give a mathematical model that captures the notion of computation in all its forms. The model we will use will be *Boolean circuits* or equivalently *Turing machines*.[1] This model should be familiar to people who have taken any course on computability, complexity, saw the results on NP-completeness, etc.. However, for cryptography it is convenient to make the following two modifications:

- Allow the algorithm to be *probabilistic*: that is, toss coins in the course of its computation.
- We will sometimes allow the algorithm to get a small *advice* string as an additional input. (This is not crucial but simplifies some of the proofs; we note that Katz-Lindell do not follow this approach.)

Computing a function Suppose that f is a function mapping n bit strings into, say, bits. We want to make precise a statement such as "f is computable in T basic computational steps". It turns out this can be done, and if f is computable in T steps according to this definition, then it is also computable in at most $1000T^2$ steps in your favorite programming language and vice versa. So, the exact choice does not really matter.

Some equivalent ways to define this are:

- f can be computed by a size-T Boolean circuit - i.e., by combining at most T, AND, OR and NOT operations.
- f can be computable in T steps by a probabilistic Turing Machine whose description is a string of length at most T.[2]
- f can be computable by a C program of at most T characters that stops within T steps for every input.

The computational model - a user's guide In this course, the only things you'll need to know about the model when trying to show that some computation takes roughly T time are:

[1]We note that the scientific community is still studying whether or not this model of the Turing machine does bound all that can be done efficiently in the physical universe. Although it seems that it captures all mechanical and biological devices that *currently* exist, a fascinating challenge is posed to this model by *quantum computers*. These are hypothetical computing devices that may be built in the future and whose computing power relative to Boolean circuits is still very much an open question (see Scott Aaronson's thesis for more on this). However, for most all of the material of this course the choice of model (e.g., Turing machines, Boolean circuits, or Quantum circuits) does not matter much, and so this debate does not effect us greatly.

[2]An equivalent and more common definition is that the length of the Turing machine's description is at most some absolute constant, say 10^6, but it is given an *advice string* $a \in \{0,1\}^T$ as an additional input.

- You can use T basic operations such as arithmetic operations, conditionals, memory reads and writes, etc...
- You can toss coins or choose random numbers in the course of the computation.
- You can assume that constants are "hardwired" into the algorithm, as long as these constants can be described by a string of length at most T.

Asymptotic notation We use asymptotic (Big Oh) notation in cryptography for reasons similar to the reasons this is used for algorithm, though in cryptography we look at things "backward".

In analyzing algorithms you'd ask a question such as "how large of an input can I handle with this algorithm?". In cryptography you ask the dual question: "how big do I need to make the key so that it will be infeasible to break my system using T steps".

If you have a scheme that cannot be broken with fewer than 2^n steps, then you will be able to use a fairly small key no matter how big you want T (e.g., even if you want to make sure the system cannot be broken by an array of supercomputers the size of the universe running for millions of years). More or less the same conclusion will hold even if 2^n is replaced by $2^{n/10}$ or $2^{n^{1/3}}$ etc..

On the other hand, if your system can be broken in n^2 steps then, although it may still be usable in some setting, it will be a very tight tradeoff between how much effort the "honest parties" are willing to spend in their daily usage of the system, and what bound can we assume on the resources available to an attacker. Again, more or less the same conclusion will hold even if n^2 is replaced by $1000n^2$ or n^3 etc..

For these reasons, and also for pedagogical reasons, it makes sense to have notation that hides away the details of precise constants, and so we use O notation. Much of the time we won't care about the difference between n^2, $100n^2$ and n^3 and we'll call all of these *polynomial time*, and denote this by $\mathrm{poly}(n)$ or $n^{O(1)}$. Similarly, we won't care about the difference between 2^n, $2^{n/10}$ or $2^{n^{1/3}}$ and we'll call these *super-polynomial time*, and denote this by $n^{\omega(1)}$.

There is one price you have to pay when you switch to asymptotic notation: you can never talk about finite functions, cryptosystems, keys, etc.. but you always talk about a sequence of such functions, one for every value of n. We will often treat these sequences implicitly, but you should always think of the key size n as some parameter that is "large enough" and one can always make larger.

Oh-notation: Formally, we use the following set of notations for functions $S, T : \mathbb{N} \to \mathbb{N}$ (they can be used for every function, but we will typically use them for functions denoting running time as function of the input or inverse of probability of success):

- $T(n) = O(S(n))$ if there exists c such that $T(n) \leq cS(n)$ for every sufficiently large n. Example: $T(n) = 100n^2$, $S(n) = n^2$.
- $T(n) = \Omega(S(n))$ if $S(n) = O(T(n))$.
- $T(n) = o(S(n))$ if for every c and sufficiently large n, $T(n) > cS(n)$. Example: $T(n) = n^2$, $S(n) = 1000n$.
- $T(n) = \omega(S(n))$ if $S(n) = o(T(n))$.
- $T(n) = \mathrm{poly}(S(n))$ if $T(n) = S(n)^{O(1)}$. That is, there exists d such that $T(n) \leq S(n)^d$ for every sufficiently large n. Example: $T(n) = 100n^5 \log n$, $S(n) = n^2$.

We say that a function T is *super-polynomial* if $T(n) = n^{\omega(1)}$. Examples: $T(n) = n^{\log n}$, $T(n) = 2^{\sqrt{n}}$.

For example, if f is a function mapping $\{0,1\}^*$ to $\{0,1\}$, then:

- We say that f has a *linear time* (i.e., $O(n)$) algorithm if there is a constant c such that f's restriction to $\{0,1\}^n$ can be computed in at most cn steps for every $n \in \mathbb{N}$.
- We say that f has a *polynomial time* (i.e., $n^{O(1)}$) algorithm if there are constants c, d such that f's restriction to $\{0,1\}^n$ can be computed in at most cn^d steps for every $n \in \mathbb{N}$.
- We say that f has *super-polynomial complexity* (i.e., $n^{\omega(1)}$) if for every constants c, d and sufficiently large n,[3] f's restriction to $\{0,1\}^n$ can *not* be computed in cn^d steps.

Probabilities Another parameter that comes up in cryptography is the probability that an adversary succeeds in an attack. Again for simplicity we will mostly care if this probability is extremely tiny (e.g., 2^{-n}, $2^{-n/10}$, $2^{-n^{1/3}}$) or relatively large (e.g., $1/10$, $1/n$, $1/n^2$). Often we'll have a bound on this probability as a function of the key size, and use the following notation:

We say that a function $\epsilon : \mathbb{N} \to [0,1]$ is *polynomially bounded* if $\epsilon(n) \geq \frac{1}{n^{O(1)}}$. Examples: $\epsilon(n) = 1/10$, $\epsilon(n) = 1/n^2$, $\epsilon(n) = 1/n^5 \log n$.

A function $\epsilon : \mathbb{N} \to [0,1]$ is *negligible* if $\epsilon(n) < \frac{1}{n^{\omega(1)}}$. Examples: $\epsilon(n) = 2^{-n}$, $\mu(n) = 2^{-\sqrt{n}}$, $\epsilon(n) = n^{-\log n}$,

Convention: In this course we use the convention that efficient computation is equal to polynomial-time. This convention is used for *pedagogical* purposes only - for simplicity of description and notations. In practice we deal with finite inputs, and so one would need to work out the precise quantitative meaning of a proof of a statement such as "breaking this encryption has super-polynomial complexity". However, equating polynomial-time with efficient computation is an extremely useful way to describe the high level ideas behind many proofs of security, without getting bogged down with the low-level details.

Polynomial-time algorithm A polynomial-time algorithm is an algorithm A that maps any input $x \in \{0,1\}^*$ into an output y within $\mathrm{poly}(|x|)$ number of steps (steps can include probability and advice).

Computational security We can now define computational security:

C/S Definition Let (E, D) be an encryption scheme that uses n-bit keys to *encrypt* $\ell(n)$-length messages. (E, D) is *computationally secure* if for every polynomial-time algorithm $A : \{0,1\}^* \to \{0,1\}$, polynomially bounded $\epsilon : \{0,1\}^* \to [0,1]$, n, and $x_0, x_1 \in \{0,1\}^{\ell(n)}$,

$$\left|\Pr[A(E_{U_n}(x_0)) = 1] - \Pr[A(E_{U_n}(x_1) = 1]\right| < \epsilon(n) \quad \textbf{(*)}$$

Conjecture 1: There exists an efficiently computable and computationally secure encryption scheme satisfying $\ell(n) = n^{100}$. (In fact, most people believe this is true even for $\ell(n) = 2^{0.9n}$.)

Game view We can also define computational security using a game between the Evesdropper and encrypter, as we did in the case of statistical security.

[3]That is, there exists N_0 such that this condition holds for every $n > N_0$.

Advanced note There is a subtle difference between the game definition and the C/S definition above: in the C/S definition we assumed security *for every* pair of messages x_0, x_1, and in the game-based definition the adversary Adv has to find the two messages x_0, x_1 herself. A-priori this seems to imply that the encryption scheme is not required to guarantee secrecy for pairs of messages that are very hard to find (e.g., strings that encode the answers to age-old riddles). However, because we allow *advice* (i.e., hardwired constants), if there exists a single pair x_0, x_1 on which one can distinguish $\mathsf{E}_{U_n}(x_0)$ from $\mathsf{E}_{U_n}(x_1)$, then the adversary can have this pair "hardwired" into its description, and because of this the two definitions are equivalent.

In contrast, Katz-Lindell do not allow advice in their definition of efficient computation, and for this reason their analog of the C/S definition only talks about pairs of messages that are *efficiently samplable.*

In our context, the difference between allowing advice and not allowing it is very small, but as in this case, allowing advice sometimes slightly simplifies definitions and proofs, which is why we do it.

Plan for next few lectures We will show that Conjecture 1 is true assuming a certain Axiom that we will name as "The PRG Axiom". Then, we will work in two directions:

1. Give evidence for the validity of the PRG Axiom, showing how we can base it on weaker and more reasonable-sounding conjectures, although at the moment it is not known how to prove it from scratch.
2. Show how assuming the PRG Axiom is true, we can get better and better cryptographic constructions: encryptions that can withstand stronger attacks such as chosen plaintext and chosen ciphertext attacks, and other constructs such as message authentication codes, digital signatures, and more.

Proofs by reduction The main tool we will use is the notion of proof by reduction. For example, it is possible to prove the following theorem:

Theorem 1. *Suppose that for every polynomial-time algorithm B and polynomially bounded $\epsilon : \mathbb{N} \to [0,1]$, $\Pr[B(p \cdot q) = \langle p, q \rangle] < \epsilon(n)$, where p and q are chosen as random n-bit long primes.*

Then Conjecture 1 is true: there exists a computationally secure encryption scheme E, D with $\ell(n) = n^{100}$.

(A variant of this theorem will follow from the results we'll see in this course.)

How do you prove something like that? The way the proof goes is by showing the following:

Theorem 2. *There exists an encryption scheme E, D and a polynomial-time algorithm B with the following property:*

*Assume that the algorithm B is given as input an number x and has access to a black-box that we denote by A. Then, if $x = pq$ for random n-bit primes p, q and A is an algorithm violating **(*)** with probability ϵ, then B will output the pair p, q with probability* $\mathrm{poly}(\epsilon)$.

Why does Theorem 2 imply Theorem 1? Because it means that if there was a T-time algorithm A that breaks the scheme (E, D) (in the sense of **(*)**) with probability ϵ, then there would be a $\text{poly}(n) \cdot T(\text{poly}(n))$-time algorithm B that factors the $2n$-bit number x with probability at least $\text{poly}(\epsilon)$. Since we assume the latter is impossible, it follows that so is the former.

Of course, the hard thing is to prove Theorem 2. Still sometimes half of the job of proving a theorem such as Theorem 1 is understanding what kind of reduction we need to prove, or in other words, understanding the statement of the corresponding Theorem 2.

All of the security proofs in this course will be proofs by reductions. (Specifically black-box reductions.)

Advanced note: The fact that all these security proofs are by reduction means that they often supply us more than just the theorem statement. For example, Theorem 2 implies that if even, say $2^{n^{1/4}}$-time algorithms cannot factor n-bit integers with, say, $2^{-n^{1/4}}$ probability then the resulting encryption scheme would have *exponential* security, and it is even possible to work out the precise security from the parameters of the reduction. Such quantitative or *concrete* analysis is needed to better understand the implications of the security proofs to the practical security of implemented schemes.