CS 290G (Fall 2014) – Introduction to Cryptography Oct 9th, 2014

Lecture 2: Computational Security

Instructor: Rachel Lin *Scribe: Binyi Chen*

1 Recap

Last class, we introduced the definition of "*perfect security*" in which any adversary can gain no further information from the *cipher text*. We proposed one method called "*One Time Pad*" to achieve this extremely strong security. However, it suffers from the fact that the *key space* has to be at least as large as the *message space*, which means in order to transmit the message with length n, Alice and Bob have to do the same work for safe key exchange. We further prove that this drawback is inherent in the definition of "perfect security", which makes this setting impractical.

Today, we will propose the new setting of *Computational Security*, which makes encrypting the messages in a large space with short keys to be possible, with the assumption that the adversary's computational power is bounded. To achieve the final goal, we should primarily build three basic infrastructures:

- Construct the model of computation.
- Define the metric of efficiency.
- Define what is a successful attack to the encryption scheme, or what kind of security requirements we have to achieve.

The first task is achieved by describing scheme's computation and adversary's strategy into *probabilistic polynomial time (PPT)* Turing Machine. The second task is achieved by calling a algorithm efficient whenever it always halts in a polynomial time. For the last one, we incorporate the definition of **one-way function** which is easy to compute but hard to invert.

2 Model of Computation

In the previous lecture, we introduce three algorithms for encryption scheme, key generation *Gen*, encryption *Enc* and decryption *Dec*. Several natural questions arise, how to implement an algorithm? How to measure their running time? We can describe an execution of algorithm by Boolean Circut or RAM machine, or even programming code, but those options are relatively complicated to analyze and generalize. Instead we use Turing Machine which is an extremely simple computational model, but achieves the same functionality of previous options. Furthermore we will give a formal definition of

algorithm's running time by Turing Machine.

For readability, we only give an intuitive explanation to Turing Machine. It consists of several tapes of input, a head that can read and write on the tapes and move left or right, a state register that store one state at any time, and a transition function table, that changes state and moves tapes according to the current state and input symbol of tape.

Definition 1 (Deterministic Algorithm) *A deterministic algorithm A is defined to be a Turing Machine M that halts in finite steps, which reads one given input tape x and write one output tape. The running time of M is function $T(\cdot)$, if $\forall x \in \{0,1\}^n, M(x)$ halts in no more than $T(n)$ steps.*

Definition 2 (Randomized Algorithm) *A randomized algorithm A_r is defined to be a Turing Machine M' which reads input x and a given infinite random binary sequence r, write one output tape and halts in finite steps. The running time of M' is function $T(\cdot)$, if $\forall x \in \{0,1\}^n, \forall r \in \{0,1\}^*, M(x,r)$ halts in no more than $T(n)$ steps.*

Definition 3 (Function Implementation) *An algorithm A implement a function f, iff $\forall x \in Dom(f), Pr[y \leftarrow A(x) : y = f(x)] = 1$.*

3 Metric of Efficiency

In order to judge whether an algorithm is efficient or not, we have to set the standard for efficiency. A naive attempt is to consider an Turing Machine efficient whenever it halts in a fixed number (let's say c) of steps. Yet it is not rational, since the running time of a machine towards different scale of problem should be different. For example, intuitively we consider a linear algorithm to be efficient, but when the input length is larger than c, we can never halt in c steps.

Another try is to consider a fixed function, e.g $\log n$, n, n^2, etc. But it seems unfair to say a linear algorithm is the same efficient as a quadratic program, and an algorithm running in cubic time is not efficient. Furthermore, the same algorithm, when implemented differently (e.g with Boolean Circuit or Turing Machine), may halt in different number of step. It is weird to say that an algorithm A is efficient when implemented by Boolean Circuit, but is not when implemented by Turing Machine. A good candidate is to consider all kinds of polynomial functions.

Definition 4 *A Turing Machine M is called efficient, iff $\forall n, \forall x \in \{0,1\}^n, \exists$ constant c, $M(x)$ halts in less than n^c steps.*

The reasons we use the definition above are as follows:

1. (Transitivity). The computation in different models (e.g Turing Machine, Boolean Circuit, RAM) incurs a polynomial-time grow-up. Therefore the standard is transitive in all computational models.

2. (Portability). There exists a universal Turing Machine U, $\forall x \in \{0,1\}^n, \forall M$ runs in $T(n)$ steps, $U(M,x) = M(x)$ and U runs in $poly(T(n))$.

3. (Composability). When Turing Machine A with oracle function f is efficient, and Turing Machine B which implement oracle function f is efficient, then Turing Machine A^B which invokes B whenever A call oracle function f, is efficient.

4. (Practicality) In real experience, a polynomial time algorithm is a good start to make the implementation practical. Since there will be more optimizations (e.g from n^8 to quadratic and to linear time) as time goes by.

4 Private Key Encryption Scheme

A private key encryption algorithm with keyspace $K = \{0,1\}^n$ and message space $M = \{0,1\}^m$ consists of three parts *Gen, Enc, Dec*.

- $Gen(1^n)$ is a Probabilistic Polynomial Time (PPT) Turing Machine that samples a key $k \in \{0,1\}^n$ given input 1^n.
- $Enc(k,m)$ is a Probabilistic Polynomial Time (PPT) Turing Machine that generates ciphertext c given key k and message m.
- $Dec(k,c)$ is a Deterministic Polynomial Time (DPT) Turing Machine that generates plaintext m given key k and ciphertext c.

4.1 Power of Adversaries

Though we assume the adversary is not computational unbounded, we give it a little bit more power. Intead of only to be a single Turing Machine, we define the adversary to be a *non-uniform PPT algorithm* which consists of a collection of Turing Machines. This enables the adversary to tailor its strategy for different input length.

The *non-uniform PPT algorithm* of the adversary is a collection of an infinite sequence of machines $M_1, M_2, ...$ Whenever the input length $|x| = n$, the adversary invokes the machine M_n. Machines also satisfy the following:

1. There exists a polynomial $p(\cdot)$ such that for every n, the description of machine M_n has length bounded by $p(n)$.

2. There exists a polynomial $q(\cdot)$ such that for every n, the running time of machine M_n on each input of length n is bounded by $q(n)$.

5 Security Goal

5.1 One-Way Function

The necessary requirement of security in computational model is that it is easy to encrypt the message with the key, but hard to invert the ciphertext to the original message without the key. Before formally generalize the security definition of computational setting, we first propose a tool that is essential for implementing a secure private key encryption scheme.

The tool is called **One-Way Function (OWF)**. To put it simply, it is a kind of function that easy to compute but hard to invert. The first attempt to define one-way function is as follows:

- There exists a PPT algorithm M, s.t, M implement f. (E.g $\forall x, Pr[y \leftarrow M(x) : y = f(x)] = 1$)
- There is no non-uniform PPT algorithm A, s.t, $\forall n, x \in \{0,1\}^n, Pr[A(n, f(x)) = x] = 1$.

The definition above suffers from several flaws. Firstly, if the function f is not injection (e.g there are more than one pre-image of $f(x)$), whenever there exists an algorithm that find another pre-image of $f(x)$, we still cannot say that the scheme is secure. Secondly, it is possible that the size $|f(x)|$ rapidly shrink, e.g, $|f(x)| \ll |x|$, this makes A even impossible to output x. Thirdly, whenever we find an algorithm that, for half of the elements x in $\{0,1\}^n$, we can find the pre-image of $f(x)$ with high probability, we still cannot ensure the security of the function.

Therefore, we give the modified definition of one-way function as follows:

Definition 5 (Negligible Function) *A function $\mu(n)$ is negligible, iff $\forall c, \exists n_0, \forall n > n_0, \mu(n) < n^{-c}$.*

Definition 6 (Strongly One-Way Function) *A function f is strongly one-way function iff it satisfies the following:*

- *(Easy to compute). There exists a PPT algorithm M, s.t, M implement f. (E.g $\forall x \in Dom(f), Pr[y \leftarrow M(x) : y = f(x)] = 1$)*
- *(Hard to invert). $\forall$ non-uniform PPT algorithm $M = \{M_1, M_2, ...\}$, $\forall n$,*

$$Pr[x \leftarrow \{0,1\}^n; y = f(x); x' \leftarrow M_n(1^n, y) : f(x') = y] < \mu(n)$$

Where $\mu(n)$ is a negligible function.