1 Recap: Perfect Secrecy

Recall from last time:

- Shannon secrecy says that the *a posteriori* distribution over the message, given the ciphertext, is identical to the *a priori* distribution. (See the lecture notes for formal definition.) Informally, "seeing the ciphertext is only as good as seeing nothing at all."
- Perfect secrecy is the property that the distribution of the ciphertext $c \in \mathcal{C}$ (over the choice of key $k \in \mathcal{K}$) is exactly the same, no matter what message $m \in \mathcal{M}$ was encrypted.
- Shannon secrecy and perfect secrecy are equivalent.
- The one-time pad scheme enjoys Shannon/perfect secrecy.

2 Limits of Perfect Secrecy

For reference, here again is the definition of perfect secrecy:

Definition 2.1 (Perfect secrecy)**.** A shared-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ is *perfectly secret* if for all $m_0, m_1 \in \mathcal{M}$ and all $\bar{c} \in \mathcal{C}$,

$$\Pr_{k \leftarrow \mathsf{Gen}}[\mathsf{Enc}_k(m_0) = \bar{c}] = \Pr_{k \leftarrow \mathsf{Gen}}[\mathsf{Enc}_k(m_1) = \bar{c}]. \tag{2.1}$$

Now that we know that the one-time pad is perfectly secret, what more is left to do? (Cancel the course?) Well, first notice that the one-time pad scheme is not necessarily very usable: the key length is as large as the message length, and for long messages it may be unrealistic for Alice and Bob to establish (and keep secret!) a huge key before communicating. (This is not to mention the additional distinct keys needed for when Carol, Dan, Edith, and Fred enter the picture...) Also notice that the key can by used only *one time*: encrypting more than one message with the same key is "abusing the model," and causes the security proof to break down. In fact, using the key more than once often makes the scheme *completely insecure*. (The USSR learned this the hard way as a result of the US/UK-led "Venona" project during the Cold War.)

It turns out that these drawbacks are not really a deficiency of the one-time pad itself, but are *inherent* in the strong requirement of perfect secrecy.

Theorem 2.2 (Shannon's theorem)**.** *If a shared-key encryption scheme (with key space $\mathcal{K}$ and message space $\mathcal{M}$) is perfectly secret, then $|\mathcal{K}| \geq |\mathcal{M}|$.*

Proof. We prove the contrapositive: supposing that $|\mathcal{K}| < |\mathcal{M}|$ for some scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, we will show that the scheme is not perfectly secret. The idea is to show the existence of two messages $m_0, m_1 \in \mathcal{M}$ and a ciphertext $\bar{c} \in \mathcal{C}$ such that m_0 and m_1 are encrypted as $\bar{c}$ with different probabilities, in violation of Definition 2.1.

First let $m_0 \in \mathcal{M}$ be arbitrary, let $\bar{k} \in \mathcal{K}$ be an arbitrary key in the support of Gen (i.e., Gen outputs $\bar{k}$ with probability > 0), and let $\bar{c} = \mathsf{Enc}_{\bar{k}}(m_0)$. (Note: $\bar{c}$ is well-defined because Enc is deterministic in our model). Then we have

$$\Pr_{k \leftarrow \mathsf{Gen}}[\mathsf{Enc}_k(m_0) = \bar{c}] \geq \Pr_{k \leftarrow \mathsf{Gen}}[k = \bar{k}] > 0.$$

Now let $\mathcal{D} = \{\mathsf{Dec}_k(\bar{c}) \ : \ k \in \mathcal{K}\} \subseteq \mathcal{M}$ be the set of all possible decryptions of $\bar{c}$ under all possible keys. (Note: this set is well-defined because Dec is deterministic.) We have $|\mathcal{D}| \leq |\mathcal{K}| < |\mathcal{M}|$, which implies that there is some message $m_1 \in \mathcal{M}$ that is not in $\mathcal{D}$. By the completeness property of the scheme, we have

$$\Pr_{k \leftarrow \mathsf{Gen}}[\mathsf{Enc}_k(m_1) = \bar{c}] = 0. \qquad \square$$

Notice that the proof above is "effective" in the sense that m_0 and m_1 can be computed by explicitly enumerating $\mathsf{Dec}_k(\bar{c})$ for every $k \in \mathcal{K}$. However, if $\mathcal{K}$ is huge, this task may take too long to be feasible. Notice also that the proof depends rather strongly on the assumption that Enc and Dec are *deterministic*, but with a bit more work the proof can be extended to the case where they are allowed to be randomized.

3 Roadmap to Computational Security

We have seen that perfect secrecy is obtainable, but impractical — the key must be as long as the message. Moreover, the proof of this fact provides an actual attack, which enumerates all possible keys. On the one hand, if the key space is huge (say, 2^{1000}), the attack may need to run for a long, long time. On the other hand, maybe there is a faster attack that we just haven't discovered yet?

Today (and throughout most of the rest of course), we will be focused on schemes that are meaningfully secure (though not "perfectly" so) *as long as the adversary does not employ an absurdly large amount of computation*. We have good reason to believe that there are inherent limits on the amount of computation that can be performed in the physical universe — for example, the number of atoms in the observable universe is estimated at around 10^{80}, and we have only about 10^{10} years until our sun runs out of fuel and collapses.

The *computationally secure* schemes we develop will have keys much shorter than the message length, but this is really just the beginning: they will also have mind-bendingly strong (seemingly even paradoxical) security and functionality properties that go far beyond just keeping a message secret.

To get there from here, there are many questions to which we must give precise mathematical answers. here are just a few:

- How do we *model* "bounded computation"?
- What does it mean for a problem to be *hard* for bounded computation?
- Where might we hope to *find* such hard problems?
- How do we *define* security against computationally bounded adversaries?
- How do we *use* hard problems to design schemes that satisfy our security definitions?

To keep a concrete example in mind, perhaps you have heard the (informal) conjecture that "factoring is hard." We will see how to state this conjecture (and others like it) precisely, and start to use it in cryptography.

4 Computational Model

4.1 Algorithms

Informally, we all know what an algorithm is: something that you can implement in your favorite programming language, and run on any kind of computer (or network of computers) that you might like. As a formal

mathematical object, an algorithm is a *Turing machine*. In this course we will not need the formal definition too often, and the informal notion will usually suffice.

The *running time* of an algorithm on an input is the number of "basic" steps it performs before terminating. Basic steps include reading or writing to a location in memory, performing an arithmetic operation (on fixed-size pieces of data), etc. For our purposes, the exact set of basic operations will not be too important, because it is possible to translate between different sets of basic operations with only polynomial slowdown: an algorithm (using one set of basic ops) with running time T can be converted into an algorithm (using another set) with running time dT^c, for some fixed constants c, d. The running time $T(n)$ of an algorithm as a function of input length n is its *maximum* running time over all inputs of length n. We say that an algorithm is *polynomial-time* if its running time $T(n) = O(n^c)$ for some fixed constant $c \geq 0$.

Two aspects of our model of an algorithm require a bit more precision. The first is *randomization*, that is, we grant algorithms the ability to "flip coins." In the Turing machine view, the machine is augmented with a read-only "random tape," which is initialized with an infinite string of uniformly random and independent bits. Alternatively, we can view the algorithm's randomness as an extra argument $r \in \{0,1\}^*$, and we denote the output of $\mathcal{A}$ on input x with "coins" r as $\mathcal{A}(x;r)$. (When omitted, r is taken to be a uniformly random string.) Note that $\mathcal{A}(x)$ defines a probability distribution over its outputs, where the probability is taken over the random coins. The running time T of a probabilistic algorithm $\mathcal{A}$ on an input x is the *maximum* number of steps performed by $\mathcal{A}(x;r)$, over all random strings r. The notions of running time as a function of the input length $n = |x|$, and *probabilistic polynomial-time* (PPT), are defined similarly. Note that a PPT algorithm can only "look at" polynomially many bits of its random string.

In this course, PPT algorithms will be our main model of "efficient computation" for cryptographic schemes. This is primarily to give us a *useful, well-behaved abstraction*, which allows us not to get too caught up in the details of implementations. Of course, an algorithm with running time n^{100} is not very useful in practice, and at times we will take a more precise view of running times.

The second aspect of our model, which we use only for modelling *adversaries*, is *non-uniformity*. The idea is that we allow an adversary to have some extra "advice" that depends only on the *length* of its input. This advice can only increase what the algorithm is capable of computing, so security against non-uniform adversaries is potentially stronger than against only uniform ones. The ability to have advice will also simplify some of our security proofs.[1] More formally, we say that $\mathcal{A}$ is a non-uniform algorithm if there exists an infinite sequence $w_1, w_2, \ldots \in \{0,1\}^*$ so that on input x, $\mathcal{A}$ is also given the extra argument $w_{|x|}$. The notion of PPT for non-uniform algorithms is as above; in particular, note that a non-uniform PPT machine can only "look at" polynomially many bits of its advice string, so without loss of generality we can assume that $|w_n| = O(n^c)$ for some constant $c \geq 0$.

4.2 Asymptotics

We have defined running time as a function of input length n, and we will also frequently do so for probabilities. The length n is frequently called the *security parameter* of a cryptographic scheme, because it (informally) lets us specify "how much" security we want.

We say that a function $T(n)$ is *polynomial*, written $T(n) = \mathrm{poly}(n)$, if $T(n) = O(n^c)$ for some fixed constant c. A nonnegative function $\nu(n)$ is *negligible*, written $\nu(n) = \mathrm{negl}(n)$, if it vanishes faster than the inverse of any polynomial: $\nu(n) = o(n^{-c})$ for every constant $c > 0$, or equivalently, $\lim_{n\to\infty} \frac{\nu}{(}n) \cdot n^c = 0$. We usually apply the concept of a negligible function to quantify the probability of an "extremely rare" event, i.e., one that will effectively "never occur" in a polynomial-time universe. As with polynomial time, the

[1] It also makes Turing machines equivalent to "circuit families," which leads to more robust security definitions.

concept of a negligible function is a *useful* and *well-behaved* abstraction, though at times we will need to be more precise in our manipulations of probabilities.

5 One-Way Functions

There are many different notions of computational "hardness" in computer science, e.g.: undecidability (exemplified by the Halting Problem), NP-completeness (circuit satisfiability), time/space hierarchies, and others.

In cryptography, we have a few notions of hardness, the most basic of which is *one-wayness*. A *one-way function* (OWF) is often called the "minimal" cryptographic object, because a scheme satisfying almost any interesting computational security notion will imply the existence of a OWF. (This is obviously an informal statement, but a good rule of thumb. We will see some examples later on.) In other words, we can't have (computational) cryptography without one-way functions.

Definition 5.1. A function $f : \{0,1\}^* \to \{0,1\}^*$ is *one-way* if it satisfies the following conditions.

- *Easy to compute.* There is an efficient algorithm computing f. Formally, there exists a (uniform, deterministic) poly-time algorithm F such that $\mathsf{F}(x) = f(x)$ for all $x \in \{0,1\}^*$.
- *Hard to invert.* An efficient algorithm inverts f on a random input with only negligible probability. Formally, for any non-uniform PPT algorithm $\mathcal{I}$, the *advantage* of $\mathcal{I}$

$$\mathbf{Adv}_f(\mathcal{I}) := \Pr_{x \leftarrow \{0,1\}^n}\left[\mathcal{I}(1^n, f(x)) \in f^{-1}(f(x))\right] = \mathrm{negl}(n)$$

  is a negligible function (in the security parameter n).

Some remarks on this definition:

1. On $f^{-1}(f(x))$ versus x: notice that the inverter $\mathcal{I}$ "wins" if it outputs *any* preimage of $f(x)$, i.e., any element in the set $f^{-1}(f(x))$. (The preimage need not even have length n.) This is mainly to rule out trivial function, such as the function f that just maps every input to 0. If we required the inverter to output the *original* $x \in \{0,1\}^n$, then the best any inverter could do against such f would be to guess randomly, which succeeds with only negligible probability 2^{-n}. But such f does not satisfy our intuitive idea of a function that is "hard to invert."

2. The choice of $\{0,1\}^*$ as the domain and range of f is mainly for syntactic convenience, and it may be replaced by any finite sets, one for each value of the security parameter n. There should also be an efficient algorithm for sampling from the domain, so that the entire inversion experiment can be executed efficiently.

3. Ideally, we would like to require $\mathcal{I}$'s advantage to be 0, i.e., the function can *never* be inverted. But this is too strong: after all, $\mathcal{I}$ can always guess x with probability 2^{-n} (without even looking at $f(x)$)! In cryptography, when we can't achieve the "ideal," we can often do the next best thing by using a negligible function to mean "effectively zero."

4. In contrast to some other notions of hardness you may have seen, this definition is inherently *average-case* rather than *worst-case*. A worst-case definition would say that every $\mathcal{I}$ fails to invert $f(x)$ for *some* (possibly rare) value of x. Our definition is potentially much stronger: it says that $\mathcal{I}$ fails to invert $f(x)$ for *almost all* values of $x \in \{0,1\}^n$ (for large enough n).

5. Notice that in addition to the value $f(x)$ to be inverted, the input to $\mathcal{I}$ includes the string 1^n. This is simply a technicality that always allows the inverter to run in time $\mathrm{poly}(n)$, even if $f(x)$ has length much less than n. Usually we will omit this extra argument, with the implicit understanding that all algorithms are given the security parameter (in unary), and may run in time polynomial in it.

5.1 Candidates

We have defined the concept of a one-way function, but does such an object exist? First, it can be shown that if a one-way function exists, then $\mathsf{P} \neq \mathsf{NP}$. Since we have no idea how to show that $\mathsf{P} \neq \mathsf{NP}$, *proving* that a one-way functions exis is far beyond our reach. But we have many *candidate* functions that we believe to be one-way. Consider these two examples:

1. Subset-sum: define $f_{\text{ss}} : (\mathbb{Z}_N)^n \times \{0,1\}^n \to (\mathbb{Z}_N)^n \times \mathbb{Z}_N$, where $N = 2^n$, as

$$f_{\text{ss}}(a_1, \ldots, a_n, b_1, \ldots, b_n) = (a_1, \ldots, a_n, S = \sum_{i\,:\,b_i=1} a_i \bmod N).$$

   Notice that because the a_is are part of the output, inverting f means finding a subset of $\{1, \ldots, n\}$ (corresponding to those b_is equaling 1) that induces the given sum S modulo N.

2. Multiplication: define $f_{\text{mult}} : \mathbb{N}^2 \to \mathbb{N}$ as[2]

$$f_{\text{mult}}(x, y) = \begin{cases} 1 & \text{if } x = 1 \vee y = 1 \\ x \cdot y & \text{otherwise.} \end{cases}$$

Are these one-way functions, according to our definition? This is something for you to think about until the next lecture.

[2]For security parameter n, we use the domain $[1, 2^n] \times [1, 2^n]$. The cases $x = 1$ and $y = 1$ are special to rule out the trivial preimages $(1, xy)$ and $(xy, 1)$ for the output $xy \neq 1$.