Cryptography and Complexity

By Tim Henderson (tadh@case.edu)
Dept. of Electrical Engineering and Computer Science
Case Western Reserve University

Modern cryptographic systems are built on problems which are assumed to be computationally infeasible. Computational infeasibility means a computation which although computable would take far too many resources to actually compute. Ideally in cryptography one would like to ensure an infeasible computation's cost is greater than the reward obtained by computing it. At first glance this seems to be an odd notion to base a cryptographic system on. Don't we want our cryptographic systems to be totally secure? They should be unbreakable! "It may take a long time to break it," seems like a poor guarantee of security.

However, it is the best guarantee which can exist in either an ideal world (from a mathematical perspective) or the physical world. As we shall see later in the survey, if several widely held assumptions turn out to be false we can not even make the guarantee of computational infeasibility.

1.1 Classical Security

In classical cryptographic systems, those known to the academic community prior to the publication of Diffie and Hellman's paper [DH76], security assumptions were based on the results of information theory. This approach is sometimes referred to as *information-theoretic* and is concerned with whether there exists information in the *ciphertext* which originated in the *plaintext* or in the *key*. We say a system has *perfect-secrecy* if:

$$\forall_{m \in \mathcal{M}} \forall_{c \in \mathcal{C}} : Pr[\mathcal{M} = m | \mathcal{C} = c] = Pr[\mathcal{M} = m] \tag{1}$$

Intuitively this formula says an attacker gains no information about the contents of a message from the ciphertext of the message. Does this mean the attacker knows nothing about the message? Of course not! However, he doesn't *learn* anything new about the message by closely examining the ciphertext. Therefore, the ciphertext of the message is essentially useless to an attacker. [HP08][1]

However, any system with *perfect-secrecy* requires the length of the key to be at least as large as the sum of the lengths of all messages encrypted with it. Since the key has to be at least as long as the messages sent such a system is of little value in practical modern situations.[Gol01][2]

There are two practical problems with a system with *perfect-secrecy* the first is "Key Distribution." Since the sender and receiver must use the same key they must some how *secretly* agree on a key beforehand. Therefore, there must exist some "second channel" by

[1]See section 4.6.1

[2]See pages 2,3

which the sender and receiver can communicate. The second problem has to do with the length of the key. Since it is as long as the message there seems to be only small utility in the system as the sender and receiver could conceivably securely exchange messages using their secure "second channel" they use for key distribution.

These to problems make the system an unrealistic system for securing (for instance) internet communications. Internet communications do not require the parties to know each other before hand and allow for no secondary secure communications channel to exist. Therefore, some other encryption methodology must be used if one wants to secure communication in this setting.

1.2 Modern Security

In modern system we no longer discuss security in terms of whether a system provides *perfect-secrecy*. Instead, we say if a ciphertext contains information leaked from the plaintext it should be computationally infeasible to extract that information. We can provide this property even in cases where the key is shorter than the message. This property can also be provided in cases where the attacker has access to the *encryption* key (but not of course to the *decryption* key).[Gol01][3]

In following section 4 we shall unpack and rigorously define (using [Gol01]'s definitions) the definition above. In particular we will look at the definition in the context of symmetric key systems. The difficulties of public key systems will also be briefly presented but without detailed exposition. However, before we get to the fun stuff we will first present complexity theory and one way functions.

2 A Tour of Computational Complexity Theory

In many ways Computability Theory, and its daughter field Complexity Theory, began with Gödel's proof of the incompleteness of axiomatic systems in 1931.[G3̈1] The proof is a tremendously important result in meta-mathematics stating: no recursively axiomatized mathematical system can be both complete and consistent. Thus, we cannot prove in a particular theory that the same particular theory is consistent. Indeed, if we did construct such a proof it would prove exactly the opposite. Thus, there are some mathematical sentences which are true for which no algorithm can decided on their truth value.

In a similar result, Alan Turing in 1937 proved that for general programs one could not decide whether the programs would halt for all inputs.[Tur37] During the previous year Alonzo Church proved the exact same thing for evaluations of λ-Calculus expressions.[Chu36] Church and Turing later conjectured that the machine Turing defined (eg. Turing Machines) and Church's lambda calculus were equivalent. All though this is an unprovable conjecture it is largely accepted today.

Once actual computational machines were produced (as opposed to the abstract machines of Turing and Church), programmers became interested in the notion of the *complexity* of an algorithm. The complexity of an algorithm is an expression of how much time or space or other resources the algorithm will use. The representation of time and space is abstract and

[3]See page 3

placed in terms of the size of the parameters to the algorithm. Today, we use asymptotic notation to express complexity assertions. The notation was standardized by Don Knuth in 1976 but in wide (although inconsistent) use before then. It was invented by Bachmann in 1894 for use in a different context.[Knu76]

The interest in the complexity of algorithms and work on linguistics (particularly formal language hierarchies) lead to work on classifying the "hardness" computational problems. For instance all of the language classes in the Chomsky hierarchy have hardness results. Type–0 Languages (all recursively enumerable languages) are recognizable but only non-deterministically. While, Type–3 languages (referred to as regular languages) can be recognized in linear time.[4][RM00]

This leads to defining complexity classes for problems (as opposed to algorithms). A complexity class typically refers to a bound on the amount time or space needed to solve the problem in the worst case. Thus, complexity classes describe how difficult a problem is to solve in general. The first general results in the theory were obtained in 1965 by Hartmanis and Stearns who defined the meaning computation complexity.

In particular Hartmanis and Stearns modeled their definition using the computational model of an N-Tape Turing Machine. Any computational model could have been used, and today others are used. In particular the authors prove facts about the computability of particular binary strings, α in the paper. They say α is in a complexity class S_T if $T : \mathbb{N} \to \mathbb{N}$ is a monotone increasing function and there exists a Turing machine $\mathcal{M}$ such that $\mathcal{M}$ computes the nth term in $T(n)$ steps.

What does this definition mean intuitively? Think of T as a time function, where time is a function of the number bits generated. A string can belongs to those complexity classes which can compute the string according to the complexity class's specified time function. Thus, by specifying some general time functions such as (1, n, n^2 $\ldots$, 2^n) one can begin classifying bit-strings. The bit-strings correspond to problem solutions. For a simple example consider the bit-string which corresponds to all prime numbers. To compute it, one would need to actually compute exactly those numbers which are prime.[HS65]

The type of problem Hartmanis and Stearns classified belongs to the class of problems known as Decision Problems. Informally, decision problems are problems to which there is a "yes/no" answer. For instance, deciding whether the first n bits of a string is in a language, $\mathcal{L} \subseteq \{0,1\}^*$, is a decision problem. Complexity classes are more general than just decision problems however, one can construct complexity classes for any type of computational problem, optimizations problems for instance.

2.1 The Class NP

Of particular importance to mathematics, computer science and this paper in particular is the complexity class NP (Non-Deterministic Polynomial Time). The class of NP is defined (intuitively) as those problems which have easily verifiable solutions. What does it mean for a solution to be "easily verifiable." It means given the problem instance and the solution one can check the validity of the solution in $O(n^k)$ where n is parameterized by the problem and k is a constant.

[4]ie. time proportional to the size of the input

More formally, the class NP is defined in terms of formal languages. Let Σ be an alphabet and Σ_0 be $\Sigma - \{*\}$ where $*$ is the empty symbol. Let Σ_0^* be the closure of all finite strings made up of symbols in Σ_0. We define a language, $\mathcal{L}$, as $\mathcal{L} \subseteq \Sigma_0^*$.

Definition 1. The Class NP[5]

> A language, $\mathcal{L}$, belongs to NP if there exists a Deterministic Turing Machine, $\mathcal{M}$, a polynomial, $p(n)$ – such that $p(n)$ defines the complexity class of $\mathcal{M}$[6] – and on any input $x \in \Sigma_0^*$:
>
> - if $x \in \mathcal{L}$ then there exists a *certificate*, $y \in \Sigma_0^*$ st. $|y| \leq p(|x|)$, and $\mathcal{M}$ accepts the input string xy.
> - if $x \notin \mathcal{L}$ then for any string, $y \in \Sigma_0^*$, $\mathcal{M}$ rejects the input string xy.

The given definition does not discuss Non-Determinism. To see the role of Non-Determinism consider constructing solutions (certificates) to problem instances (the string x in the definition). If certificates can be chosen and examined non-deterministically then it will take only polynomial time to find a solution. However, if we are testing every possible certificate deterministically it will take $|\Sigma_0|^{p(|x|)}$ examinations, a combinatorial explosion. Thus, problems in NP have solutions which are easy to verify but not necessarily easy to construct.

2.2 The Class P

The complexity class P (Polynomial Time) is exactly those problems solvable in deterministic polynomial time. More formally,

Definition 2. The Class P[7]

> Let Π be a decision problem. Let $L_\Pi = \{x \in \Sigma_0^* | \text{x is an encoding of an instance of } \Pi\}$, that is, L_Π is the language of Π. We can then define the class P as:
>
> $$P = \{L \subseteq \Sigma_0^* | \text{ there is a Deterministic Turing Machine, } \mathcal{M}\text{, and a polynomial, } p(n)\text{, such that } T_\mathcal{M} \leq p(n) \text{ for all } n \geq 1\}$$

A language is in P if one can construct a Turing Machine which accepts it (and rejects all non-members) in time less than some polynomial (with respect to the size of the input).

All those problems which belong to P are considered easily solvable, or tractable. While, they are “easy” one should not make the mistake of assuming they are simple. Given a polynomial time algorithm which solves a problem one can easily solve it. However, even if you know a polynomial time algorithm exists for a problem constructing the algorithm may be difficult.

[5]See [TW06] page 41 at the bottom.
[6]That is $\mathcal{M}$ computes in the nth bit of output in $p(n)$ time.
[7]See [TW06] page 23.

2.3 P vs. NP

What is the relationship between P and NP? It is known P is contained in NP (ie. $P \subseteq NP$). However, whether $NP \subseteq P$ is true is one of the greatest open questions in applied mathematics. The class P containment inside of NP is obvious: if we can find a solution in polynomial time it is certainly verifiable in polynomial time. To show NP is contained within P one would need to show every problem in NP can be solved with a polynomial time algorithm.

The methodology for solving P vs NP with the greatest impact relies on the idea of *reduction*. We say problem, Π_1 is *reducible* to another problem, Π_2, if one can find a mapping from every instance of Π_1 to equivalent instances of Π_2 such that the solutions to the constructed instances of Π_2 correspond to solutions of Π_1. A reduction is a *polynomial reduction* if the mapping can be done in polynomial time. A problem, Π_1, is as "hard" as another problem, Π_2, if Π_2 can be *reduced* to Π_2. Thus, hardness is a relational notion. Problems are not intrinsically hard, they are hard with respect to other problems.

To prove P contains NP one could prove the hardest problem in NP is also in P. By the definition of hardness above, if a problem, Π_h, is the hardest problem in NP than every other problem in NP is reducible to Π_h. Problems which are at least as hard as every problem in NP are know as NP-Hard problems. A problem does not need to be in NP to be NP-Hard. However, if a problem is NP-Hard and it is in NP then it is called an NP-Complete problem.

NP-Complete problems exist and their existence is one of the greatest results in complexity theory. It was proved by Stephen Cook in 1971 who found the first NP-Complete problem. The problem he found is known as SAT (for satisfiability of boolean formulas). He proved any problem solvable in polynomial time by a Nondeterministic Turing Machine can be reduced to finding whether or not a boolean formula is satisfiable.[Coo71] Cook's result launched a wave of research. The very next year Richard Karp proved 21 other problems were also NP-Complete.[Kar72]

While there have many hundreds of problems proven to be NP-Complete since Cook proved SAT, there have been only fruitless attempts to prove P does or does not contain NP. The leading consensus in the complexity community is P does not contain NP. Furthermore, since it appears work on proving P contains NP is permanently stalled one can safely assume NP-Hard problems are in some platonic sense actually difficult to solve.

2.4 Infeasibility

A sharper definition of computational infeasibility can now be given with definitions of Complexity Classes, P, and NP in hand. Recall the opening statement on infeasibility, where we defined an infeasible computation to be one requiring too many resources to actually compute. If one has encrypted a message one would ideally like the ciphertext to be unreadable. If the message is a solution to an NP-Complete problem then the ciphertext could be the problem instance and therefore can only be decrypted by solving the NP-Complete problem. However, assuming NP is not contained in P, the NP-Complete problem will take time proportional to $|\Sigma_0^*|^{|x|}$ (where x is the ciphertext) to solve.

Therefore, a new working definition of an infeasible computation is a "hard" instance of an NP-Hard problem of sufficient size. What is sufficient size? Any size which leads to

$|\Sigma_0^*|^{|x|}$ to be so large as to be uncomputable. An example of such as size might be 160 since trying 2^{160} possible solutions is not expected to ever be computable with classical computers in time less than the age of the universe. What is a "hard" instance? A hard instance is one in which there exists no better way to find a solution than trying all possible solutions. Not every instance of a hard problem is hard to solve. Specifying an infeasible computation requires a hard instance is a necessary restriction.

2.5 Probabilistic Infeasibility

In the previous section it was assumed all computations were exact. No computation *sometimes* gave the right answer and sometimes did not. However, with an algorithm which mostly gives right answers could be very useful to the cryptanalyst. Therefore, we briefly turn our attention to probabilistic computations.

2.5.1 Probabilistic Turing Machines

A Probabilistic Turing Machine (PTM) is a Deterministic Turing Machine (DTM) with an extra input tape. The tape is called the "coin flipping tape." The PTM can read one bit of information at a time from the coin flipping tape. Each bit is assured to be a random bit.[8] Computation on the machine proceeds as before except at any time a random choice can be made. This allows us to construct algorithms which will "probably" but not necessarily produce the desired answer.

Analyzing the running time of a PTM is a bit different than a DTM. While a DTM's running time only depends on its program and the initial configuration of the input tape, a PTM also depends on the random bits it reads during the computation. Therefore, the running time of a PTM is a random variable (we denote it as $t_{\mathcal{M}}(x)$). Furthermore, whether a PTM halts or not on a fixed input is also a random variable. A *halting* PTM is one which halts after a finite number of steps for all inputs and all configurations of the coin tossing tape.

With the definition a *halting* PTM in hand we are now prepared to reason about its running time. Worst case running time of a PTM, $T_{\mathcal{M}}(n)$, is:

$$T_{\mathcal{M}}(n) = \max\{t \mid \text{there exists a } x \in \Sigma_0^n \text{ such that } \Pr[t_{\mathcal{M}}(x) = t] > 0\} \tag{2}$$

Informally, this definition states: the worst case running time of a PTM is the maximum running time, $t_{\mathcal{M}}(x)$, for which the machine will run with some probability greater than zero. A polynomial PTM is one in which there exists some positive polynomial, $p(\cdot)$, such

[8] Since a PTM is a theoretical construction rather than a physical construction we can do away with the nasty realities of life and assume these random bits are actually random! A nice change of pace.

that $T_{\mathcal{M}}(n) \leq p(n)$ holds.[TW06][9]

Definition 3. BPP, Bounded Probability Polynomial Time

A language $\mathcal{L}$ is recognized by a polynomial PTM, $\mathcal{M}$, if:

- *for every* $x \in \mathcal{L}$ *it holds that* $\Pr[\mathcal{M}$ accepts $x] \geq \frac{2}{3}$
- *for every* $x \notin \mathcal{L}$ *it holds that* $\Pr[\mathcal{M}$ does not accept $x] \geq \frac{2}{3}$

BPP is the class of languages recognized by a polynomial PTM.[10,11]

The class Bounded Probability Polynomial Time, sometimes called Bounded-*Error* Probabilistic Polynomial Time, is somewhat analogous to the class P. Computations in BPP are considered feasible computations. The class P is contained within BPP, $P \subseteq BPP$. However, the relationship between NP and BPP has not been established. In practice cryptographers assume $NP \nsubseteq BPP$ which implies $NP \neq P$. All problems unsolvable by a polynomial PTM are considered infeasible, of which NP-Hard problems are a special case. As before, some instances of hard problems may in fact be easy to solve.

Infeasible computations as defined above are nice formalisms but do not seem too useful. To utilize the previous definition one has to answer the following question: Given and instance of a problem is it a *hard* instance? Unfortunately, we don't know how to answer this question.[12] As we will see later, if we could easily find hard instances we could construct a simple and secure crypto-system by sampling hard instances. Therefore, cryptographers need better assurances than *worst-case* assurances; a cryptographer needs to know a typical instance of a problem is hard.[Imp95]

3 One Way Functions

With a firm grounding in Complexity Theory, we turn our attention to cryptography. First, by capturing the notion of exploitable computational difficulty as epitomized in the one way function. A one way function is a function which is *easy* to compute but *hard* to invert. More specifically:

Definition 4. One Way Function

1. $\forall_x$ computing $f(x) = y$ is *easy* to compute.
2. $\forall_y$ computing $f^{-1}(y)$ such that $f(x) \in f^{-1}(y)$ is *hard* to compute.

The one way function in definition 4 is more of a theoretical construct than an actual mathematical construct. Therefore, it uses the notion of *easy* and *hard* computations without grounding itself with exact definitions. One can think of this first definition as an abstract, or ideal, definition.

[9]See section 4.2

[10]Note, any constant greater than $\frac{1}{2}$ can be used here.

[11]Definition is a combination of Definition 1.3.4 from [Gol01] and the definition given in Section 4.5 of [TW06].

[12]From personal discussion with Prof. Harold Connamacher (harold.connamacher@cwru.edu)

Ignoring for the moment the definitional problems, what use is a one way function to the cryptographer? It turns out one can define secure cryptosystems with one way functions. Such a cryptosystem will be discussed in detail in section 4. For now consider this simple example of the power of the idea:

> One day while toiling away, Ian had a flash of insight which would put his mechanical workings to right. A machine danced in his mind, one which would make widgets faster and better than before. So clever his insight he knew no one else would easily come up with the same idea. Thus, he decided not to patent it. Instead, he wrote down his idea and ran it through a one way function producing y his certificate of his idea. He then published y widely, placing it in all the libraries around the country.
>
> Many years passed and Mallory stole Ian's idea. Mallory being very clever sought to undue Ian and patented the idea. Then, he sued Ian for patent infringement. But, since Ian had a certificate of his invention, y, he could prove to the court he had invented and known about the idea long before Mallory had filed for the patent. The court agreed with Ian and invalidated Mallory's patent.

The challenge in section 4 will be transforming the one way function into a workable encryption device. For while a powerful concept, as demonstrated by the story above, it is non-obvious how a crypto-system can be constructed from it. But before crypto-systems, the definition must be tightened. Furthermore, one must be convinced one way functions can be reasonably expected to exist.

3.1 Strong One Way Functions

There are two vague terms used in definition 4, *easy* and *hard* computations. Fortunately, we have already defined what an *easy* computation is: an easy computation is on which can be done in (probabilistic) polynomial time. But what about inversion? What does it mean for a function to be hard to invert? A function, f, is hard to invert if every probabilistic polynomial time algorithms will only invert f with *negligible* probability.

Definition 5. Negligible[13]

> A function, $\mu : \mathbb{N} \to \mathbb{R}$, is negligible if for every positive polynomial, $p(\cdot)$, there exists an N such that for all $n > N$,

$$\mu(n) < \frac{1}{p(n)}$$

The definition of negligible is reminiscent of Asymptotic Notation used in the analysis of algorithms. It concerns itself with the behavior of the function, $\mu(n)$, when n grows large. An additional, and useful, feature of the definition is any negligible function remain negligible after multiplication with any polynomial $q(\cdot)$. Therefore, any event which occurs with

[13]Definition due to [Gol01] see Def. 1.3.5

negligible probability will continue to occur with negligible probability even after polynomial repetitions. Thus, if f is only invertible with polynomial time algorithm, A, with negligible probability than no polynomial repetition of A will be likely to invert f.

Definition 6. Strong One Way Functions[14]

A function, $f : \{0,1\}^* \to \{0,1\}^*$, is **strongly** one way if it is:

Easy to compute There exists a (deterministic) polynomial time algorithm A such that on input x algorithm A outputs $f(x)$ (ie. $A(x) = f(x)$).

Hard to invert For *every* probabilistic polynomial time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n the probability A' inverts f is negligible. That is:

$$\Pr[A'(f(x)) \in f^{-1}(f(x))] < \frac{1}{p(n)}$$

In the above definition "input x" should be considered as a random variable drawn from a uniform distribution over $\{0,1\}^n$. Thus, the second condition reads: for any random input of size n the probability an arbitrary polynomial time algorithm will find a pre-image is negligible. If such a function could be found or constructed it would offer a strong assurance of computational difficulty.

3.2 Weak One Way Functions

While strong one way functions ensure any efficient inversion algorithm has only a negligible likelihood of succeeding; weak one way functions require efficient inversion algorithms will fail with a non-negligible probability.

Definition 7. Weak One Way Functions[15]

A function, $f : \{0,1\}^* \to \{0,1\}^*$, is **weakly** one way if it is:

Easy to compute There exists a (deterministic) polynomial time algorithm A such that on input x algorithm A outputs $f(x)$ (ie. $A(x) = f(x)$).

Slightly hard to invert There exists a polynomial $p(\cdot)$ such that for every probabilistic polynomial time algorithm, A', and a sufficiently large n's,

$$\Pr[A'(f(x)) \notin f^{-1}(f(x))] > \frac{1}{p(n)}$$

In definition 6 the probability that A' could invert f has an upper bound of $p(\cdot)^{-1}$ for *every* positive polynomial. In definition 7, there is a *single* positive polynomial, $p(\cdot)$, such that $p(\cdot)^{-1}$ is a lower bound on the failure of any efficient inversion algorithm. Unlike strong one way functions, weak one way functions are not hard for typical instances. However, they are hard for some percentage of instances.

[14]Definition due to [Gol01] see Def. 2.2.1. Note, I simplified the definition slightly for clarity.

[15]Definition due to [Gol01] see Def. 2.2.2

3.2.1 Amplification of Weak One Way Functions

Since weak functions are hard for a non-negligible percentage of inputs they can be used to construct strong functions. The proof for this bold assertion is given by Goldreich.[Gol01][16] Since one can convert a weak one way function into a strong one it suffices to find weak ones. While a strongly one way function may yield a more efficient cryptosystem a weak one will still allow a secure system (as discussed in section 4).

3.3 Hard Core Predicates

If Alice has a strong one way function f, computes $y = f(x)$, and sends y to Bob while Eve eavesdrops what can Eve learn about x? Depending on the function f Eve may be able to learn a surprising amount. Since f is hard to invert Eve cannot learn everything about x but she may not need too. Is there some way to quantify which bits of x Eve can learn about and which bits she can't?

There is! The bits which are hard for a polynomial attacker (like Eve) to learn about are called the "Hard Core" of a one way function. A predicate is a yes/no question, for example: Does x end with a 0? If a yes/no question is hard for Eve to answer it is called a Hard Core Predicate. Since a yes/no question only has 2 possible answers Eve can always guess the answer. Therefore, a predicate is only hard for her to answer if she can't do better than get it right about half the time. To be precise:

$$\Pr[\text{EveGuess\_P}(y) = P(x)] \leq \frac{1}{2} + neg(|x|)$$

where $neg(|x|)$ is a negligible function (as defined in definition 5). This description of Eve trying to guess something about x, like whether it starts with 0, leads nicely into a formal definition:

Definition 8. Hard-Core Predicates[17]

> A polynomial time computable predicate, $b : \{0,1\}^* \to \{0,1\}$, is called a **hard-core** of a function, f, if for every probabilistic polynomial time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large $|x|$'s,
>
> $$\Pr[A'(f(x)) = b(x)] < \frac{1}{2} + \frac{1}{p(|x|)}$$

Given a hard to invert function, f, one knows some of the bits in its input must be hard to predict from the output. How does one know which bits are the hard bits? In general deciding what bits are hard for a function is difficult but one can always construct a Hard-Core Predicate for any strong one way function. Since one can always construct a strong one way function from a weak function this poses no limitation to the framework.

[16]See Theorem 2.3.2 for an impractical but demonstrative conversion and Section 2.6 for an efficient conversion in the case of one-way permutations.

[17]Definition due to [Gol01] see Def. 2.5.1

3.3.1 Constructing Hard-Core Predicates

The following result was first proved in 1982 by Yao but we present a simplification due to Goldreich and Levin as presented by Talbot and Welsh.[TW06][18] A detailed proof is available as usual in the Goldreich book.[Gol01][19]

Theorem 1. Hard-Core Predicates from Strong One Way Functions

> Let f be an arbitrary strong one way function. Let g be defined as $g(x, r) = (f(x), r)$, where $|x| = |r|$. Let r be a random bit string. Then define $B(x, r)$ to be a Hard-Core Predicate of g by:
>
> $$\begin{aligned} B(x, r) &= \sum_{i=1}^{|x|} x_i r_i \pmod 2 \\ &= \bar{x} \cdot \bar{r} \pmod 2 \end{aligned}$$

The theorem states, if f is strongly one way then it will be hard to guess the result of taking an exclusive-or of a random subset of x given $f(x)$ and the subset r. If $B(x, r)$ is not a hard-core of g then f is easily invertible. The proof involves constructing an algorithm from the predictor for B. For details on the construction once again see Goldreich.

With the result of theorem 1 and the ability to construct strong one way functions from weak one way functions one will always be able to construct a function where at least one predicate on x is hard to compute. If one bit is not enough it turns out *hard-core functions* are also constructable. However, their specific details are well out of the scope of this paper.

3.4 Constructing One Way Functions

It one is going to build a crypto-system based on hard computational problems (specifically strong one way functions) one should have some way of identifying such problems. From a practical perspective there are three number theoretic based problems which are assumed to be one way functions. The first is the discrete log problem: $g^x \equiv y \pmod p$, second finding square roots mod $N = pq$, and third the "RSA" problem $c \equiv x^e \pmod N$. While these problems are likely to be used in practice none of them are suspected to be in the class NP-Hard. While, instances of problems in NP-Hard may be efficiently solvable there is good evidence they are not. In contrast these problems are potentially vulnerable to good approximation algorithms.

Thus, an open problem for the aspiring cryptographer to tackle is to suggest a novel one way function. However, serious care needs to be exercised when suggesting such a function. It is not good enough for the function to be difficult in the *worst-case* it must be difficult in the typical case. Average case complexity analysis relies heavily on the input distribution. Thus, the input distribution must be carefully characterized and uniform sampling techniques must be developed. Without exercising such care the aspiring cryptographer may fall into the trap of defining something which appears secure from a cursory theoretical glance but on close inspection is quite vulnerable.

[18]Theorem 10.8
[19]Section 2.5.2

4 Secure Encryption

Secure encryption schemes are naturally built on top of strong one way functions with hard-core predicates. However, before the encryption schemes can be defined a formal definition of security must be stated. Until now, our definition has been colloquial: information in the ciphertext should be computationally infeasible to extract. The informal definition is too vague for use in defining an encryption system because the security definition is more important than the cryptographic system itself. A proper definition ensures systems conforming to the definition will be more difficult to attack.

4.1 Security Definitions

Before rigorously defining a modern definition of security let us turn once again to classical security and *perfect-secrecy.* Recall perfect secrecy says an attackers *uncertainty* about a message should not be reduced when in possession of a corresponding ciphertext. As noted in the introduction, the obvious criticism of *perfect-secrecy* is the implied key length. In such a system, the length of the key must be at least as long as the message. Making the definition impractical for most modern uses of cryptography. Therefore, a new definition is indeed necessary.

4.1.1 Polynomial Indistinguishability

The first definition we will consider is *polynomial-indistinguishability.* Informally, if Alice has two messages, M_1 and M_2 and she sends Bob a ciphertext, C, Eve who has been given both messages and the ciphertext will have no easy way to determine which message it corresponds to. Something is easy for Eve if she can do it in probabilistic polynomial time. Indeed, it is assumed none of our characters can do any computations except easy ones. Formally,

Definition 9. Polynomial Indistinguishability of Encryptions[20]

> An encryption scheme, (G, E, D), where G generates keys, E encrypts messages, and D decrypts messages has *indistinguishable encryptions* if for every probabilistic polynomial time algorithm, A', every polynomial $p(\cdot)$, all sufficiently large n, and every $x, y \in \{0,1\}^{\text{poly}(n)}$ with $|x| = |y|$,
>
> $$|\Pr[A'(E_{G(1^n)}(x)) = 1] - \Pr[A'(E_{G(1^n)}(y)) = 1]| < \frac{1}{p(n)}$$

The above definition was written with a symmetric encryption and decryption keys. However, the public key version only has minor and unimportant complications. The importance of the definition is in the intuition. Eve, the attacker, knows both messages and she has a ciphertext. The only thing she does not know is the key used to create the ciphertext. If the system is polynomially indistinguishable then Eve can only guess which message the ciphertext corresponds to. Since there are two messages she will only get it right half the

[20]Definition due to [Gol04] see Def. 5.2.3, simplified.

time. If she can get it right better than half the time then the system is *not* polynomially indistinguishable.

The *security* of the definition is perhaps non-obvious but consider the case were Eve can distinguish which message the ciphertext corresponds too. If the system was supposed to have *perfect-secrecy* then clearly the secrecy would have been violated. Some bit of information would be leaking from the message to the ciphertext. Therefore, what the definition is saying is no information is leaking from the message to the ciphertext which can be extracted in polynomial time.

4.1.2 Semantic Security

The intuitive explanation of polynomial indistinguishability is captured in an alternative definition: *semantic-security*. A crypto-system is semantically secure if any piece of information Eve can compute given a ciphertext she could just as easily compute without the ciphertext. That is, the ciphertext provides Eve with no advantage for computing any piece of information of interest to her. Formally,

Definition 10. Semantic Security[21]

> An encryption scheme, (G, E, D), where G generates keys, E encrypts messages, and D decrypts messages is *semantically secure* if for every probabilistic polynomial time algorithm, A, there exists another probabilistic polynomial time algorithm, A', such that for every message $\mathcal{M}$ of length n, every pair of functions with polynomially bounded output $f, h : \{0,1\}^* \to \{0,1\}^*$, every polynomial $p(\cdot)$, and all sufficiently large n,
>
> $$\begin{aligned}&\Pr[A(1^n, E_{G_1(1^n)}(\mathcal{M}), h(1^n, \mathcal{M})] = f(1^n, \mathcal{M})] \\ &< \Pr[A'(1^n, h(1^n, \mathcal{M})] = f(1^n, \mathcal{M})] + \frac{1}{p(n)}\end{aligned}$$

In the above definition, f represents the information Eve would like to compute. The information Eve wants, f, is a function of the message and the length of the message (encoded for technical reasons in unary). The output of f is polynomial however it is not necessary for f to be a *computable* function. The algorithm A guesses f using the ciphertext, the length of the message, and h. The algorithm A' guesses f using only the length of the message and h. The function h represents a polynomial amount of *a-priori* knowledge about the output of f.

The definition of semantic security intuitive says the probability Eve can guess f utilizing the ciphertext is at most negligibly greater than guessing f without the ciphertext. The definition places no restrictions on what Eve might be guessing (other than an upper bound on its size). Eve could be guessing whether the message is an order to move troops, or the message is a bank account number; it makes no difference to the definition.

Semantic-security is therefore the complexity theory analog of *perfect-secrecy*. It provides assurance to the cryptographer that a polynomially bound cryptanalyst will be able to gain no information from the ciphertext. In practice, one only cares about polynomially bound adversaries since exponential adversaries do not exist.

[21]Definition due to [Gol04] see Def. 5.2.1, simplified.

4.1.3 Equivalence of Definitions

In a potentially surprising result it turns out it doesn't matter which security definition one uses, they imply each other:

Theorem 2. Equivalence of Definitions[22]

> An encryption scheme is semantically secure if and only if it has indistinguishable encryptions.

In practice, it is usually far easier to prove a scheme has indistinguishable ciphertexts. However, from a security perspective the property one actually wants is *semantic-security.* Thus, theorem 2 provides the cryptographer with an incredibly useful result.

4.2 A Secure Symmetric Key Encryption Scheme

To construct a perfectly secret symmetric key encryption scheme from an information theory perspective one first obtains a large amount of random information. One then takes a random bit for each bit of message and exclusive-ors them together. One now has the perfect cryptographic system. The construction of a semantically secure system is quite similar (in the case of stream ciphers). One takes a bit of random information, referred alternately as the seed of the key, stretches it to create a pseudo-random sequence the same length as the message. The message and the pseudo-random sequence are then xored together. This encryption scheme will clearly be semantically secure if no adversary can distinguish between the pseudo-random sequence and a truly random sequence.

4.2.1 Pseudo-Random Sequence Generators

A pseudo-random bit generator, $G(x)$ is defined as a deterministic polynomial time algorithm taking a bit-string, $x \in \{0,1\}^k$, and outputting a longer string $G(x)$. In other words, the generator stretches the input. For the generator to be pseudo-random in nature, the output must be unpredictable if the input is random. Luckily, we already know how to produce bits which are essentially unguessable by a polynomial adversary. Hard-core predicates by construction cannot be guessed correctly better than half the time.

Theorem 3. A Pseudo-Random Generator can be Constructed from any One Way Permutation.[23,24]

> Let $f : \{0,1\}^* \to \{0,1\}^*$ be a one-way function length preserving permutation with a hard core predicate $B : \{0,1\}^* \to \{0,1\}$ then,
>
> $$G : \{0,1\}^k \to \{0,1\}^{k+1}$$
> $$G(x) = (f(x), B(x))$$
>
> is a pseudo-random generator.

[22]Theorem (and proof) due to [Gol04] Theorem 5.2.5.

[23]Theorem and proof due to [TW06] see theorem 10.9. My proof is a summary of Talbot and Welsh's main argument

[24]A one way permutation is simply a one way function which is a bijection from the domain to the range. The existence of one way functions implies the existence of one way permutations.

Proof. If x is a random string, and therefore drawn from a uniform distribution over $\{0,1\}^k$, then $f(x)$ is also a random string. Therefore, if there is some test, T, which can distinguish $G(x)$ from a random string of length $k+1$ it must be distinguishing the last bit, $B(x)$. Since, it can distinguish $B(x)$ from a random bit then one must be able to guess it significantly better than half the time. However, this contradicts $B(x)$ being a hard-core predicate of $f(x)$. Therefore, f is either not a one-way function or $G(x)$ is a pseudo-random generator. $\square$

While, theorem 3 certainly constructs a pseudo-random number generator it is hardly a useful one. Recall, the issue with perfect secrecy was the key size. If one constructed a stream cipher from using theorem 3 one would only save 1 bit of key size over a one time pad. Luckily, the following extension also holds:

Theorem 4. An $l(k)$ Pseudo-Random Generator[25]

Let $f : \{0,1\}^* \to \{0,1\}^*$ be a one-way function length preserving permutation with a hard core predicate $B : \{0,1\}^* \to \{0,1\}$. If $l(\cdot)$ is a positive polynomial then,

$$G : \{0,1\}^k \to \{0,1\}^{l(k)}$$
$$G(x) = (B(x), B(f(x)), B(f^2(x)), ..., B(f^{l(k)-1}(x)))$$

is a pseudo-random generator.

With the construction in theorem 4 one can now generate a strong pseudo-random sequence. If f is a strong one way function with a hard-core then no polynomial adversary can discern between the output of the generator above and a truly random string.

Definition 11. A Symmetric Key Stream Cipher[26]

Setup Alice chooses a short random key $x \in_R \{0,1\}^k$

Key Distribution Alice secretly shares x with Bob.

Encryption Alice encrypts an m-bit message, M, by generating a pseudo-random string:

$$G(x) = (B(f(x)), B(f^2(x)), ..., B(f^m(x)))$$

and forming the cryptogram $C = G(x) \otimes M$

Decryption Bob creates the same string $G(x)$ an recovers the message via $M = G(x) \otimes C$.

The strength of the cipher relies on the strength of $G(x)$ and the strength of $G(x)$ relies on the underlying properties of the one way function, f. The above stream cipher is clearly semantically secure since the ciphertexts are indistinguishable by Eve. If Eve could distinguish the ciphertexts than she could predict $G(x)$. If Eve can predict $G(x)$ than f must not be a one way function.

While the above stream cipher is semantically secure, it is not necessarily the construction one would use in practice. Often, one would instead want to use a block cipher. Luckily, one can also construct block based ciphers from pseudo-random generators. For these an many other complications I refer you to Oded Goldreich's 2004 book.[Gol04]

[25]Theorem due to [TW06] see Theorem 10.10

[26]Definition due to [TW06] see page 216.

4.3 Public Key Schemes

I will not discuss the public key schemes in detail. The definitions for security setup in section 4.1 are implicitly for symmetric key systems. While, the modifications are fairly trivial they should be given proper treatment. In addition the public key systems deserve a thorough explanation. I will settle for some brief remarks.

The RSA cryptographic system does not satisfy the property of polynomial indistinguishability. In particular, if Eve wants to tell whether C corresponds to M_1 or to M_2 all she has to do is encrypt both messages can compare their ciphertexts. Eve and easily do this since the encryption algorithm in a public key system is public and therefore available to Eve.

The encryption algorithm being publicly available seems to be an insurmountable obstacle at first, but it turns out to be possible to overcome it. In the case of the RSA algorithm one needs to introduce randomness (and thus uncertainty) into the encryption process. One such suggestion *Randomized RSA* introduces random data into each encryption thus ensuring polynomial indistinguishability. However, Randomized RSA comes with a cost: one must believe a different strong assumption. One must assume RSA has a “large” hard-core of bits in the input. While, this may be a reasonable assumption it is a *different* assumption and not implied by the usual RSA assumption. For details I once again commend you to Goldreich’s 2004 book.[Gol04]

5 Concluding Remarks

Basing cryptographic security on computation complexity is a sound practice. It yields systems with strong and extensible security guarantees. However, it also requires strong assumptions. In particular, we must believe in “one way functions.” While, there is good evidence they exist, and several candidate functions appear to work, we do not *know* they exist. But, until a better formalism comes along complexity theory is secure in its position as the basis of modern cryptography.

References

[Chu36] Alonzo Church, *An Unsolvable Problem of Elementary Number Theory*, American Journal of Mathematics **58** (1936), no. 2, 345.

[Coo71] Stephen A. Cook, *The complexity of theorem-proving procedures*, Proceedings of the third annual ACM symposium on Theory of computing - STOC ’71 (New York, New York, USA), ACM Press, 1971, pp. 151–158.

[DH76] Whitfield Diffie and M. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **22** (1976), no. 6, 644–654.

[G3̈1] Kurt Gödel, *On Formally Undecidable Propositions of Principia Mathematica and Related Systems (translated by B. Meltzer 1962)*, Monatshefte für Mathematik und Physik **38** (1931), 173–198.

[Gol01] Oded Goldreich, *The Foundations of Cryptography : Basic Tools*, Cambridge University Press, Cambridge, 2001.

[Gol04] ______, *The Foundations of Cryptography : Basic Applications*, Cambridge University Press, Cambridge, 2004.

[HP08] Jeffery Hoffstein and JC Pipher, *An Introduction to Mathematical Cryptography*, Springer, New York, 2008.

[HS65] J Hartmanis and R E Stearns, *On the computational complexity of algorithms*, Transactions of the American Mathematical Society **117** (1965), 285–285.

[Imp95] R. Impagliazzo, *A personal view of average-case complexity*, Structure in Complexity Theory Conference, 1995., Proceedings of Tenth Annual IEEE, IEEE, 1995, pp. 134–147.

[Kar72] R.M. Karp, *Reducibility among combinatorial problems*, 50 Years of Integer Programming 1958-2008 (1972), 219–241.

[Knu76] Donald E Knuth, *Big Omicron and big Omega and big Theta*, ACM SIGACT News **8** (1976), no. 2, 18–24.

[RM00] Kenneth H Rosen, John G Michaels, and ..., *Handbook of discrete and combinatorial mathematics*, CRC Press, Boca Raton, 2000.

[Tur37] A. M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society **s2-42** (1937), no. 1, 230–265.

[TW06] John Talbot and Dominic Welsh, *Complexity and Cryptography: An Introduction*, Cambridge University Press, Cambridge, 2006.