Computational Complexity[*]

Salil Vadhan, Harvard University
salil@eecs.harvard.edu

July 15, 2004

Computational complexity theory is the study of the minimal resources needed to solve computational problems. In particular, it aims to distinguish between those problems that possess efficient algorithms (the "easy" problems) and those that are inherently intractable (the "hard" problems). Thus computational complexity provides a foundation for most of modern cryptography, where the aim is to design cryptosystems that are "easy to use" but "hard to break". (See computational security.)

Running Time. The most basic resource studied in computational complexity is *running time* — the number of basic "steps" taken by an algorithm. (Other resources, such as *space* (i.e., memory usage), are also studied, but we will not discuss them here.) To make this precise, one needs to fix a model of computation (such as the Turing machine), but here we will informally think of it as the number of "bit operations" when the input is given as a string of 0's and 1's. Typically, the running time is measured as a function of the *input length.* For numerical problems, we assume the input is represented in binary, so the length of an integer N is roughly $\log_2 N$. For example, the elementary-school method for adding two n-bit numbers has running time proportional to n. (For each bit of the output, we add the corresponding input bits plus the carry.) More succinctly, we say that addition can be solved in time "order n", denoted $O(n)$ (see O-notation). The elementary-school multiplication algorithm, on the other hand, can be seen to have running

[*] This is an entry to appear in the *Encyclopedia of Cryptography and Security* [7]. Underlined words/phrases refer to other entries in the encyclopedia.

time $O(n^2)$. In these examples (and in much of complexity theory), the running time is measured in the *worst case*. That is, we measure the maximum running time over all inputs of length n.

Polynomial Time. Both the addition and multiplication algorithms are considered to be efficient, because their running time grows only mildly with the input length. More generally, polynomial time (running time $O(n^c)$ for a constant c), is typically adopted as the criterion of efficiency in computational complexity. The class of all computational problems possessing polynomial-time algorithms is denoted $\mathbf{P}$.[1] Thus Addition and Multiplication are in $\mathbf{P}$, and more generally we think of $\mathbf{P}$ as identifying the "easy" computational problems. Even though not all polynomial-time algorithms are fast in practice, this criterion has the advantage of robustness: the class $\mathbf{P}$ seems to be independent of changes in computing technology. $\mathbf{P}$ is an example of a *complexity class* — a class of computational problems defined via some algorithmic constraint, in this case "polynomial time".

In contrast, algorithms that do not run in polynomial time are considered infeasible. For example, consider the *trial division* algorithms for integer factoring or primality testing (see prime number). For an n-bit number, trial division can take time up to $2^{n/2}$, which is exponential time rather than polynomial time in n. Thus, even for moderate values of n (e.g. $n = 200$) trial division of n-bit numbers is completely infeasible for present-day computers, whereas addition and multiplication can be done in a fraction of a second. Computational complexity, however, is not concerned with the efficiency of a particular algorithm (such as trial division), but rather whether a problem has *any* efficient algorithm at all. Indeed, for primality testing, there are polynomial-time algorithms known (see prime number), so Primality is in $\mathbf{P}$. For integer factoring, on the other hand, the fastest known algorithm has running time greater than $2^{n^{1/3}}$, which is far from polynomial. Indeed, it is believed that Factoring is not in $\mathbf{P}$; the RSA and Rabin cryptosystems (see RSA public-key encryption, RSA digital signature scheme, Rabin cryptosystem, Rabin signature scheme) rely on this conjecture. One of the ultimate goals of computational complexity is to rigorously prove such *lower bounds*, i.e. establish theorems stating that there is no polynomial-time algorithm for a given problem. (Unfortunately, to date, such theorems have been elusive, so

[1] Typically, $\mathbf{P}$ is defined as a class of *decision problems* (i.e. problems with a yes/no answer), but here we make no such restriction.

cryptography continues to rest on conjectures, albeit widely believed ones. More on this below.)

Polynomial Security. Given the above association of "polynomial time" with feasible computation, the general goal of cryptography becomes to construct cryptographic protocols that have polynomial efficiency (i.e., can be executed in polynomial time) but super-polynomial security (i.e., cannot be broken in polynomial time). This guarantees that, for a sufficiently large setting of the *security parameter* (which roughly corresponds to the input length in complexity theory), "breaking" the protocol takes much more time than using the protocol. This is referred to as *asymptotic security.*

While polynomial time and asymptotic security are very useful for the theoretical development of the subject, more refined measures are needed to evaluate real-life implementations. Specifically, one needs to consider the complexity of using and breaking the system for fixed values of the input length, e.g. $n = 1000$, in terms of the actual time (e.g. in seconds) taken on current technology (as opposed the "basic steps" taken on an abstract model of computation). Efforts in this direction are referred to as *concrete security.* Almost all results in computational complexity and cryptography, while usually stated asymptotically, can be interpreted in concrete terms. However, they are often not optimized for concrete security (where even constant factors hidden in O-notation are important).

Even with asymptotic security, it is sometimes preferable to demand that the gap between the efficiency and security of cryptographic protocols grows even more than polynomially fast. For example, instead of asking simply for super-polynomial security, one may ask for *exponential security* (i.e. cannot be broken in time 2^{n^ϵ} for some $\epsilon > 0$). Based on the current best known algorithms, it seems that FACTORING may have exponential hardness and hence the cryptographic protocols based on its hardness may have exponential security. [2]

Complexity-Based Cryptography. As described above, a major aim of complexity theory is to identify problems that cannot be solved in polynomial

[2] In cryptography, a slightly different definition of exponential hardness is typically employed, with *exponential security* (compare exponential time) only referring to protocols that cannot be broken in time $2^{\epsilon n}$ for some $\epsilon > 0$. Accordingly, in cryptography, FACTORING is typically considered to provide *subexponential security* (compare subexponential time).

time and a major aim of cryptography is to construct protocols that cannot be broken in polynomial time. These two goals are clearly well-matched. However, since proving lower bounds (at least for the kinds of problems arising in cryptography) seems beyond the reach of current techniques in complexity theory, an alternative approach is needed.

Present-day complexity-based cryptography therefore takes a *reductionist approach*: it attempts to relate the wide variety of complicated and subtle computational problems arising in cryptography (forging a signature, computing partial information about an encrypted message, etc.) to a few, simply stated assumptions about the complexity of various computational problems. For example, under the assumption that there is no polynomial-time algorithm for Factoring (that succeeds on a significant fraction of composites of the form $n = pq$), it has been demonstrated (through a large body of research) that it is possible to construct algorithms for almost all cryptographic tasks of interest (e.g., asymmetric cryptosystems, digital signature schemes, secure multiparty computation, etc.). However, since the assumption that Factoring is not in $\mathbf{P}$ is only a conjecture and could very well turn out to be false, it is not desirable to have all of modern cryptography to rest on this single assumption. Thus another major goal of complexity-based cryptography is to abstract the properties of computational problems that enable us to build cryptographic protocols from them. This way, even if one problem turns out to be in $\mathbf{P}$, any other problem satisfying those properties can be used without changing any of the theory. In other words, the aim is to base cryptography on assumptions that are as weak and general as possible.

Modern cryptography has had tremendous success with this reductionist approach. Indeed, it is now known how to base almost all basic cryptographic tasks on a few simple and general complexity assumptions (that do not rely on the intractability of a single computational problem, but may be realized by any of several candidate problems). Among other things, the text below discusses the notion of a *reduction* from complexity theory that is central to this reductionist approach, and the types of general assumptions, such as the existence of *one-way functions*, on which cryptography can be based.

Reductions. One of the most important notions in computational complexity, which has been inherited by cryptography, is that of a *reduction* between computational problems. We say that problem Π reduces to problem Γ if Π can be solved in polynomial time given access to an "oracle" that solves Γ (i.e. a hypothetical black box that will solve Γ on instances of our choosing in a single time step). Intuitively, this captures the idea that problem Π is no harder than problem Γ. For a simple example, let us see that PRIMALITY reduces to FACTORING.[3] Suppose we have an oracle that, when fed any integer, returns its prime factorization in one time step. Then we could solve PRIMALITY in polynomial time as follows: on input N, feed the oracle with N, output "`prime`" if the only factor returned by the oracle is N itself, and output "`composite`" otherwise.

It is easy to see that if problem Π reduces to problem Γ, and $\Gamma \in \mathbf{P}$, then $\Pi \in \mathbf{P}$: if we substitute the oracle queries with the actual polynomial-algorithm for Γ, we obtain a polynomial-time algorithm for Π. Turning this around, $\Pi \notin \mathbf{P}$ implies that $\Gamma \notin \mathbf{P}$. Thus, reductions give a way to use an assumption that one problem is intractable to deduce that other problems are intractable. Much work in cryptography is based on this paradigm: for example, one may take a complexity assumption such as "there is no polynomial-time algorithm for FACTORING" and use reductions to deduce statements such as "there is no polynomial-time algorithm for breaking encryption scheme X". (As discussed later, for cryptography, the formalizations of such statements and the notions of reduction in cryptography are more involved than suggested here.)

NP. Another important complexity class is **NP**. Roughly speaking, this is the class of all computational problems for which solutions can be *verified* in polynomial time.[4] For example, given that PRIMALITY is in **P**, we can easily see that FACTORING is in **NP**: to verify that a supposed prime factorization of a number N is correct, we can simply test each of the factors for primality and check that their product equals N. **NP** can be thought of as the class of "well-posed" search problems: it is not reasonable to search for something unless you can recognize when you have found it. Given this natural definition, it is not surprising that the class **NP** has taken on a fundamental position in computer science.

It is evident that $\mathbf{P} \subseteq \mathbf{NP}$, but whether or not $\mathbf{P} = \mathbf{NP}$ is considered to be one of the most important open problems in mathematics and com-

[3] Of course, this reduction is redundant given that PRIMALITY is in **P**, but suppose for a moment that we did not know this.

[4] **NP** stands for *nondeterministic polynomial time.* Like **P**, **NP** is typically defined as a class of decision problems, but again that constraint is not essential for our informal discussion.

puter science.[5] It is widely believed that $\mathbf{P} \neq \mathbf{NP}$, indeed, we have seen that FACTORING is one candidate for a problem in $\mathbf{NP} \setminus \mathbf{P}$. In addition to FACTORING, **NP** contains many other computational problems of great importance, from many disciplines, for which no polynomial-time algorithms are known.

The significance of **NP** as a complexity class is due in part to the **NP**-*complete* problems. A computational problem Π is said to be **NP**-complete if $\Pi \in \mathbf{NP}$ and every problem in **NP** reduces to Π. Thus the **NP**-complete problems are the "hardest" problems in **NP**, and are the most likely to be intractable. (Indeed, if even a single problem in **NP** is not in **P**, then all the **NP**-complete problems are not in **P**.) Remarkably, thousands of natural computational problems have been shown to be **NP**-complete. (See [1].) Thus, it is an appealing possibility to build cryptosystems out of **NP**-complete problems, but unfortunately, **NP**-completeness does not seem sufficient for cryptographic purposes (as discussed later).

Randomized Algorithms. Throughout cryptography, it is assumed that parties have the ability to make random choices; indeed this is how one models the notion of a secret key. Thus, it is natural to allow not just algorithms whose computation proceeds deterministically (as in the definition of **P**), but also consider *randomized algorithms* — ones that may make random choices in their computation. (Thus, such algorithms are designed to be implemented with a physical source of randomness. See random bit generation (hardware).)

Such a randomized (or *probabilistic*) algorithm A is said to solve a given computational problem if on every input x, the algorithm outputs the correct answer with high probability (over its random choices). The error probability of such a randomized algorithm can be made arbitrarily small by running the algorithm many times. For examples of randomized algorithms, see the probabilistic primality tests in the entry on prime number. The class of computational problems having polynomial-time randomized algorithms is denoted **BPP**.[6] A widely believed strengthening of the $\mathbf{P} \neq \mathbf{NP}$ conjecture is that $\mathbf{NP} \not\subseteq \mathbf{BPP}$.

[5] The significance of **P** vs. **NP** in mathematics comes from the fact that it is equivalent to asking whether we can find short mathematical proofs efficiently.

[6] **BPP** stands for "bounded-error probabilistic polynomial time."

P vs. NP and Cryptography. The assumption $\mathbf{P} \neq \mathbf{NP}$ (and even $\mathbf{NP} \not\subseteq \mathbf{BPP}$) is *necessary* for most of modern cryptography. For example, take any efficient encryption scheme and consider the following computational problem: given a ciphertext C, find the corresponding message M along with the key K and any randomization R used in the encryption process. This is an **NP** problem: the solution (M, K, R) can be verified by re-encrypting the message M using the key K and the randomization R and checking whether the result equals C. Thus, if $\mathbf{P} = \mathbf{NP}$, this problem can be solved in polynomial time, i.e. there is an efficient algorithm for breaking the encryption scheme.[7]

However, the assumption $\mathbf{P} \neq \mathbf{NP}$ (or even $\mathbf{NP} \not\subseteq \mathbf{BPP}$) does not appear *sufficient* for cryptography. The main reason for this is that $\mathbf{P} \neq \mathbf{NP}$ refers to *worst-case complexity.* That is, the fact that a computational problem Π is not in **P** only means that for every polynomial-time algorithm A, there *exist* inputs on which A fails to solve Π. However, these "hard inputs" could conceivably be very rare and very hard to find. Intuitively, to make use of intractability (for the security of cryptosystems), we need to be able to efficiently generate hard instances of an intractable computational problem.

One-way functions. The notion of a one-way function captures the kind of computational intractability needed in cryptography. Informally, a one-way function is a function f that is "easy to evaluate" but "hard to invert". That is, we require that the function f can be computed in polynomial time, but given $y = f(x)$, it is intractable to recover x. The difficulty of inversion is required to hold even when the input x is chosen *at random.* Thus, we can efficiently generate hard instances of the problem "find a preimage of y", by selecting x at random and setting $y = f(x)$. (Note that we actually generate a hard instance together with a solution; this is another way in which one-way functions are stronger than what follows from $\mathbf{P} \neq \mathbf{NP}$.) To formalize the definition, we need the concept of a *negligible function.* A function $\epsilon : \mathbb{N} \to [0, 1]$ is negligible if for every constant c, there is an n_0 such that $\epsilon(n) \leq 1/n^c$ for all $n \geq n_0$. That is, ϵ vanishes faster than any

[7] Technically, to conclude that the cryptosystem is broken requires that the message M is uniquely determined by ciphertext C. This will essentially be the case for most messages if the message length is greater than the key length. (If the message length is less than or equal to the key length, then there exist encryption schemes that achieve information-theoretic security (for a single encryption, e.g. the one-time pad), regardless of whether or not $\mathbf{P} = \mathbf{NP}$.

polynomial. Then we have:

Definition 1 (one-way function) *A one-to-one function f is* one-way *if it satisfies the following conditions.*

1. *(Easy to evaluate) f can be evaluated in polynomial time.*
2. *(Hard to invert) For every probabilistic polynomial-time algorithm A, there is a negligible function ϵ such that*

$$\Pr[A(f(X)) = X] \leq \epsilon(n),$$

where the probability is taken over selecting a input X of length n uniformly at random and the random choices of the algorithm A.

For simplicity, we have only given the definition for one-to-one one-way functions. Without the one-to-one constraint, the definition should refer to the problem of finding *some* preimage of $f(X)$, i.e. require the probability that $A(f(X)) \in f^{-1}(f(X))$ is negligible.[8]

The input length n can be thought of as corresponding to the *security parameter* (or *key length*) in a cryptographic protocol using f. If f is one-way, we are guaranteed that by making n sufficiently large, inverting f takes much more time than evaluating f. However to know how large to set n in an implementation requires a concrete security analogue of the above definition, where the maximum success probability ϵ is specified for A with a particular running time on a particular input length n, and a particular model of computation.

The "inversion problem" is an **NP** problem (to verify that X is a preimage of Y, simply evaluate $f(X)$ and compare with Y). Thus, if $\mathbf{NP} \subseteq \mathbf{BPP}$ then one-way functions do not exist. However, the converse is an open problem, and proving it would be a major breakthrough in complexity theory. Fortunately, even though the existence of one-way functions does not appear to follow from $\mathbf{NP} \not\subseteq \mathbf{BPP}$, there are a number of natural candidates for one-way functions.

[8] For technical reasons, we also need to require that f does not shrink its input too much, e.g. that the length of $|f(x)|$ and length of $|x|$ are polynomially related (in both directions.

Some Candidate One-Way Functions. These examples are described informally, and may not all match up perfectly with the simplified definition above. In particular, some are actually *collections of one-way functions* $\mathcal{F} = \{f_i : \mathcal{D}_i \to \mathcal{R}_i\}$, in the functions f_i are parameterized by an index i that is generated by some randomized algorithm.[9]

1. (Multiplication) $f(p,q) = p \cdot q$, where p and q are primes of equal length. Inverting f is the FACTORINGproblem (see integer factoring, which indeed seems intractable even on random inputs of the form $p \cdot q$.

2. (Subset Sum) $f(x_1, \ldots, x_n, S) = (x_1, \ldots, x_n, \sum_{i \in S} x_i)$. Here each x_i is an n-bit integer and $S \subseteq [n]$. Inverting f is the SUBSET SUMproblem (see knapsack cryptosystem). This problem is known to be **NP**-complete, but for the reasons discussed above, this does not provide convincing evidence that f is one way (nevertheless it seems to be so).

3. (The Discrete Log Collection) $f_{G,g}(x) = g^x$, where G is a cyclic group (e.g. $G = \mathbb{Z}_p^*$ for prime p), g is a generator of G, and $x \in \{1, \ldots, |G|-1\}$. Inverting $f_{G,g}$ is the DISCRETE LOGproblem (see discrete logarithm problem), which seems intractable. This (like the next two examples) is actually a collection of one-way functions, parametrized by the group G and generator g.

4. (The RSA Collection) $f_{n,e}(x) = x^e \bmod n$, where n is the product of two equal-length primes, e satisfies $\gcd(e, \phi(n)) = 1$, and $x \in \mathbb{Z}_n^*$, . Inverting $f_{n,e}$ is the RSA problem.

5. (Rabin's Collection (see Rabin cryptosystem, Rabin signature scheme) $f_n(x) = x^2 \bmod n$, where n is a composite and $x \in \mathbb{Z}_n^*$. Inverting f_n is known to be as hard as factoring n.

6. (Hash functions & block ciphers) Most cryptographic hash functions seem to be *finite* analogues of one-way functions with respect to *concrete* security. Similarly, one can obtain candidate one-way functions from block ciphers, say by defining $f(K)$ to be the block cipher applied to some fixed message using key K.

[9] Actually, one can convert a collection of one-way functions into a single one-way function, and conversely. See [3].

In a long sequence of works by many researchers, it has been shown that one-way functions are indeed the "right assumption" for complexity-based cryptography. On one hand, almost all tasks in cryptography imply the existence of one-way functions. Conversely (and more remarkably), many useful cryptographic tasks can be accomplished given any one-way function.

Theorem 1 (see [3, 4]) *The existence of one-way functions is necessary and sufficient for each of the following:*

- *The existence of commitment schemes.*
- *The existence of pseudo-random number generators.*
- *The existence of pseudorandom functions.*
- *The existence of symmetric cryptosystems.*
- *The existence of digital signature schemes.*

These results are proven via the notion of reducibility mentioned above, albeit in much more sophisticated forms. For example, to show that the existence of one-way functions implies the existence of pseudorandom generators, one describes a general construction of a pseudorandom generator G from any one-way function f. To prove the correctness of this construction, one shows how to "reduce" the task of inverting the one-way function f to that of "distinguishing" the output of the pseudorandom generator G from a truly random sequence. That is, any polynomial-time algorithm that distinguishes the pseudorandom generator can be converted into a polynomial-time algorithm that inverts the one-way function. But if f is one-way, it cannot be inverted, so we conclude that the pseudorandom generator is secure. These reductions are much more delicate than those arising in, say, the **NP**-completeness, because they involve non-traditional computational tasks (e.g., inversion, distinguishing) that must be analyzed in the average case (i.e. with respect to nonnegligible success probability).

The general constructions asserted in Theorem 1 are very involved and not efficient enough to be used in practice (though still polynomial time), so it should be interpreted only as a "plausibility result". However, from special cases of one-way functions, such as one-way permutations (see one-way function) or some of the specific candidate one-way functions mentioned earlier, much more efficient constructions are known.

Trapdoor Functions. For some tasks in cryptography, most notably public-key encryption (see public-key cryptography), one-way functions do not seem to suffice, and additional properties are used. One such property is the *trapdoor* property, which requires that the function *can* be easily inverted given certain "trapdoor information". We do not give the full definition here, but just list the main properties. (See also trapdoor one-way function.)

Definition 2 (trapdoor functions, informal) *A collection of one-to-one functions $\mathcal{F} = \{f_i : \mathcal{D}_i \to \mathcal{R}_i\}$ is a collection of* trapdoor functions *if*

1. *(Efficient generation) There is a probabilistic polynomial-time algorithm that, on input a security parameter n, generates a pair (i, t_i), where i is the index to a (random) function in the family and t_i is the associated "trapdoor information".*

2. *(Easy to evaluate) Given i and $x \in \mathcal{D}_i$, one can compute $f_i(x)$ in polynomial time.*

3. *(Hard to invert) There is no probabilistic polynomial-time algorithm that on input $(i, f_i(x))$ outputs x with nonnegligible probability. (Here, the probability is taken over i, $x \in \mathcal{D}_i$, and the coin tosses of the inverter.)*

4. *(Easy to invert with trapdoor) Given t_i and $f_i(x)$, one can compute x in polynomial time.*

Thus, trapdoor functions are *collections* of one-way functions with an additional trapdoor property (Item 4). The RSA and Rabin collections described earlier have the trapdoor property. Specifically, they can be inverted in polynomial time given the factorization of the modulus n.

One of the main applications of trapdoor functions is for the construction of public-key encryption schemes.

Theorem 2 (see [4]) *If trapdoor functions exist, then public-key encryption schemes exist.*

There are a number of other useful strengthenings of the notion of a one-way function, discussed elsewhere in this volume: claw-free permutations, collision-resistant hash functions (see collision resistance), and universal one-way hash functions

Other Interactions with Cryptography. The interaction between computational complexity and cryptography has has been very fertile. Above we have described the role that computational complexity plays in cryptography. Conversely, several important concepts that originated in cryptography research have had a tremendous impact on computational complexity. Two notable examples are the notions of pseudo-random number generators and interactive proof systems. For more on these topics and the resulting developments in computational complexity, see [2].

Further Reading. Above we have touched upon only a small portion of computational complexity, and even on the topics covered, many important issues were ignored (not to mention historical references). Thus, we refer the reader to the text [3, 4] for more on computational complexity as it relates to cryptography (including formal definitions and proofs for the theorems mentioned above), and the texts [6, 5] for other aspects of complexity theory.

Acknowledgments. I thank Mihir Bellare, Ran Canetti, Oded Goldreich, the editor Burt Kaliski, and an anonymous reviewer for helpful comments on this entry.

References

[1] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[2] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs, and Pseudorandomness.* Number 17 in Algorithms and Combinatorics. Springer-Verlag, 1999.

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

[4] Oded Goldreich. *Foundations of Cryptography II: Basic Applications.* Cambridge University Press, 2004.

[5] Christos H. Papadimitriou. *Computational Complexity.* Addison–Wesley, 1994.

[6] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 1997.

[7] Henk van Tilborg, ed. *Encyclopedia of Cryptography and Security.* Kluwer. To appear.