4

Cryptography and Computation after Turing

Ueli Maurer

Abstract

This paper explores a topic in the intersection of two fields to which Alan Turing has made fundamental contributions: the theory of computing and cryptography.

A main goal in cryptography is to prove the security of cryptographic schemes. This means one wants to prove that the computational problem of breaking the scheme is infeasible, i.e., its solution requires an amount of computation beyond reach of current and even foreseeable future technology. As cryptography is a mathematical science, one needs a (mathematical) definition of computation and of the complexity of computation. In modern cryptography, and more generally in theoretical computer science, the complexity of a problem is defined via the number of steps it takes for the best program on a universal Turing machine to solve the problem.

Unfortunately, for this general model of computation, no proofs of useful lower bounds on the complexity of a computational problem are known. However, if one considers a more restricted model of computation, which captures reasonable restrictions on the power of a algorithm, then very strong lower bounds can be proved. For example, one can prove an exponential lower bound on the complexity of computing discrete logarithms in a finite cyclic group, a key problem in cryptography, if one considers only so-called generic algorithms that can not exploit the specific properties of the representation (as bit-strings) of the group elements.

The author is supported in part by the Swiss National Science Foundation.

4.1 Introduction

The task set to the authors of articles in this volume was to write about a topic of (general) scientific interest and related to Alan Turing's work. We present a topic in the intersection of computing theory and cryptography, two fields to which Turing has contributed significantly. The concrete technical goal of this paper is to reason about provable security in cryptography. The article is partly based on Maurer (2005).

Computation and *information* are the two most fundamental concepts in computer science, much like mass, energy, time, and space are fundamental concepts in physics. Understanding these concepts continues to be a primary goal of research in theoretical computer science. As witnessed by Turing's work, many underlying questions are of comparable intellectual depth as the fundamental questions in physics and mathematics, and are still far from being well-understood.

Unfortunately, this viewpoint on computer science is often overlooked in view of the enormous practical significance of information technology for the economy and the society at large. Prospective university students should know better that computer science is not only an engineering discipline of paramount importance, but at the same time a fundamental science, and the high school curricula should include more computer science topics, not only computer literacy courses.

Two of the greatest minds of the 20th century have contributed in a fundamental manner to the understanding of the concepts of computation and information. In the 1930's, Alan Turing Turing (1936) provided a mathematical definition of computation by proposing the Turing machine as a general model of computation. This model is still universally used in computer science. In the 1940's, Claude Shannon Shannon (1948) founded information theory and defined information for the first time in a meaningful and quantitative manner. This theory allowed to formalize the coding and transmission of information in a radically new way and was essential for the development of modern communication technologies.

Remarkably, both Turing and Shannon also made fundamental contributions to cryptography. Actually, their interest in cryptography can be seen as a possible source of inspiration for the mentioned foundational work on computing theory and information theory, respectively. In fact, as reported to the author by Andrew Hodges (see Hodges, 1992, p. 120 and p. 138), in late 1936, just after publication of Turing (1936), Turing wrote in a letter to his mother:

> I have just discovered a possible application of the kind of thing I am working on at present. It answers the question "What is the most general kind of code or cipher possible", and at the same time (rather naturally) enables me to construct a lot of particular and interesting codes. One of them is pretty well impossible to decode without the key, and very quick to encode. I expect I could sell them to H.M. Government for quite a substantial sum, but am rather doubtful about the morality of such things.

This demonstrates that Turing had an interest in cryptography before being appointed to work on breaking German ciphers at Bletchley Park. Unfortunately, this work never became publicly available and seems to have been lost. But what is clear is that he had in mind to develop a theory of provable cryptographic security, a topic this article explores. One can only speculate what Turing might have been able to achieve in the field of theoretical cryptography had he spent more time on the subject.

Another important connection between Turing's work on cryptography and on computing is the fact that his work on breaking German codes required the construction of one of the first practical computers. Turing's insights later helped construct the first electronic tube-based computers.

4.2 Cryptography

4.2.1 Introduction

Cryptography can be understood as the mathematical science of information security exploiting an information difference (e.g. a secret key known to one party but not to another). It is beyond the scope of this article to give a detailed account of achievements in cryptography, and we refer, for example, to Maurer (2000) for such a discussion.

Cryptography, and even more so cryptanalysis, has played an important role in history, for instance in both world wars. We refer to Kahn (1967); Singh (1999) for very good accounts of the history of cryptography. Before the second world war, cryptography can be seen as an art more than a science, mainly used for military applications, and concerned almost exclusively with encryption. The encryption schemes were quite ad-hoc with essentially no theory supporting their security. In sharp contrast, modern cryptography is a science with a large variety of applications other than encryption, often implemented by sophisticated cryptographic protocols designed by mathematicians and computer scientists. Without cryptography, security on the Internet or any other modern information system would be impossible.

There are perhaps two single most important papers which triggered the transition of cryptography from an art to a science: "Communication theory of secrecy systems" (Shannon, 1949), a companion paper of Shannon (1948); and, even more influential "New directions in cryptography" (Diffie–Hellman, 1976), in which they revealed their invention of public-key cryptography.

In an article in connection to Alan Turing's work, a historical note about the invention of public-key cryptography is unavoidable. In the late 1990s, the British government announced that public-key cryptography was originally invented at the

Government Communications Headquarters (GCHQ) in Cheltenham in the early 1970s (see Singh, 1999) by James Ellis and Clifford Cocks, who proposed essentially the Diffie–Hellman protocol (Diffie–Hellman, 1976) as well as the RSA public-key cryptosystem (Rivest–Shamir–Adleman, 1978) invented a year later. Since scientists working for government agencies can generally not publish their work, their contributions and inventions become publicly known much later only (if ever), often after their death. This remark also applies to Turing's work on cryptanalysis and the construction of practical (code-breaking) computers, which became publicly known only in the 1970s. Turing's life might have taken a very different turn had his great contributions been publicly known and acknowledged before his prosecution and tragic death.

4.2.2 The Need for Secret Keys

Encryption, like other cryptographic schemes, requires a secret key shared by sender and receiver (often referred to as Alice and Bob), but unknown to an eavesdropper. In a military context, such a key can be established by sending a trusted courier who transports the key from the headquarters to a communications facility. In a commercial context, sending a courier is completely impractical. For example, for a client computer to communicate securely with a server, one needs a mechanism that provides an encryption key instantaneously.

However, the problem is that Alice and Bob are connected only by an insecure channel, for example the Internet, accessible to an eavesdropper. Therefore, a fundamental problem in cryptography is the generation of such a shared secret key, about which the eavesdropper has essentially no information, by communication only over an authenticated[1] but otherwise insecure channel. This is known as the key agreement problem.

The key can then be used to encrypt and authenticate subsequently transmitted messages. That one can generate a secret key by only public communication appears highly paradoxical at first glance, but the abovementioned work of Diffie and Hellman provides a surprising solution to this paradox.

4.2.3 Proving Security

In cryptography, one of the primary goals is to prove the security of cryptographic schemes. Security means that it is impossible for a special hypothetical party, the adversary (or eavesdropper), to solve a certain problem, e.g. to determine the message or the key. The impossibility can be of two different types, and thus one distinguishes two types of security in cryptography.

[1] The authenticity of this communication is often guaranteed by the use of so-called certificates.

A cryptographic system that no amount of computation can break is called *information-theoretically secure*. The best-known example is the so-called one-time pad which encrypts a binary plaintext sequence by adding (bitwise modulo 2) a uniformly random binary key sequence that is independent of the plaintext. The resulting ciphertext can easily be shown to be statistically independent of the plaintext, hence provides absolutely no information about it, even for a party with unbounded computing power. However, due to the required key length and the fact that the key cannot be reused, the one-time pad is highly impractical and is used only in special applications such as the encryption of the Washington–Moscow telephone hotline at Reagan and Gorbachev's time.

Systems used in practice could theoretically be broken by a sufficient amount of computation, for instance by an exhaustive key search. The security of these systems relies on the computational infeasibility of breaking it, and such a system is referred to as *computationally secure*.

Proving the security of such a cryptographic system means to prove a lower bound on the complexity of a certain computational problem, namely the problem of breaking the scheme. Such a proof must show not only that a key search is infeasible, but that any other conceivable way of breaking the scheme is infeasible. In a sense, such a proof would imply that no future genius can ever propose an efficient breaking algorithm.

Unfortunately, for general models of computation, such as a universal Turing machine, no useful lower bound proofs are known, and it is therefore interesting to investigate reasonably restricted models of computation if one can prove relevant lower bounds for them.

This paper investigates reasonable computational models in which one *can* prove computational security.

4.3 Computation

Computer science is concerned with the following fundamental questions. What is computation? Which functions (or problems) are computable (in principle)? For computable functions, what is the complexity of such a computation? As mentioned, in cryptography one is interested in proving lower bounds on the complexity.

Computation is a physical process. A computation is usually performed on a physical computational device, often called a computer. There are many different instantiations of computational devices, including the (by now) conventional digital computers, a human head, analog computational devices, biological computers, and quantum computers.

Computer science wants to make mathematical (as opposed to physical) state-

ments about computation, for example that a certain problem is computationally hard or that a certain function is not computable at all. Therefore one needs to define a mathematical model of computation. Turing was one of the first to recognize the need for a mathematical model of computation and proposed what has become known as the *Turing machine*, the most prominent model of computation considered in theoretical computer science.

Other models of computation, for example Church's lambda calculus (Church, 1932), have also been proposed. When judging the usefulness of a computational model, the first question to ask is whether it is general in the sense that anything computable in principle, by any computational device, is computable in the model under consideration. Since computation is ultimately physical, such an argument of complete generality can never be made (unless one can claim to completely understand physics and, hence, Nature). However, most proposed models can be shown to be equivalent in the sense that anything computable in one model is also computable in another model. The so-called Church–Turing thesis postulates that this notion of computation (e.g. Turing machines) captures what is computable in principle, with any physical device. In fact, Turing gave ingenious arguments for the claim that his model captures anything that person doing a computation with pencil and paper could do.

However, the choice of model does matter significantly when one wants to analyze the *complexity* of a computation, i.e., the minimal number of steps it takes to solve a certain computational problem. For example, quantum computers, which are not more powerful than classical computers in terms of what can be computed, are (believed to be) vastly more efficient for certain computational problems like factoring large integers (Shor, 1994).

4.4 The Diffie–Hellman Key-Agreement Protocol

4.4.1 Preliminaries

The abovementioned famous key-agreement protocol, as originally proposed in Diffie–Hellman (1976), makes use of exponentiation with respect to a base g, modulo a large prime p (for instance, a prime with 2048 bits, which corresponds to about 617 decimal digits), i.e., of the mapping

$$x \mapsto g^x \pmod{p},$$

where $a \pmod{b}$ for numbers a and b is the remainder when a is divided by b (for example, $67 \pmod{7}$ is 4). The prime p and the base g are public parameters, possibly generated once and for all, for all users of the system. In more mathematical terminology, one computes in the multiplicative group of the ring $\mathbf{Z}/p\mathbf{Z}$. We de-

note this group by $\mathbf{Z}_p^*$. The toy example $p = 19$ and $g = 2$ is shown in Figure 4.1. Note that $\mathbf{Z}_{19}^*$ is a cyclic group with 18 elements, the numbers from 1 to 18.

While $y = g^x \pmod p$ can be computed efficiently, even if p, g, and x are numbers of several hundred or thousands of digits (see below), computing x when given

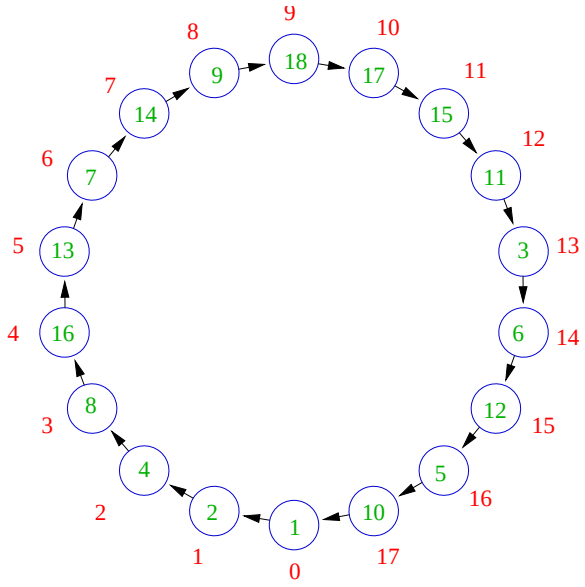

Figure 4.1 The group $\mathbf{Z}_{19}^*$, generated by the generator $g = 2$. The numbers on the outside of the circle are the exponents and the numbers within the small circles are the corresponding elements of $\mathbf{Z}_{19}^*$. For example, we have $2^{11} \equiv_{19} 15$, i.e., the remainder when 2^{11} is divided by 19 is 15, namely $2^{11} = 2048 = 107 \cdot 19 + 15$. Also, for example the discrete logarithm of 6 to the base 2 is 14, as can be seen by inspection.

p, g, and $y = g^x$ is generally believed to be computationally highly infeasible. This problem is known as (a version of) the *discrete logarithm problem*, which will be discussed later.

4.4.2 Efficient Exponentiation

We briefly describe an efficient exponentiation algorithm, the so-called *square-and-multiply* algorithm. To compute g^x in some mathematical structure (e.g. $\mathbf{Z}_p^*$), one writes the exponent x as a binary number. For example, $x = 23$ is written as $x = 10111_2$. An accumulator variable a is initialized to the value g. One then processes x bit-by-bit, as follows. In each step, say the ith step, one updates a by the rule

$$a := \begin{cases} a^2 & \text{if } x_i = 0 \\ a^2 g & \text{if } x_i = 1, \end{cases}$$

where x_i is the ith bit of x (starting from the left but ignoring the most significant bit, which is always 1). For example, for $x = 23 = 10111_2$, the algorithm performs four steps, where $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, and $x_4 = 1$. After the first step, the accumulator a contains the value g^2. After the second step, a contains the value $(g^2)^2 \cdot g = g^5$. After the third step, a contains the value $(g^5)^2 \cdot g = g^{11}$. Finally, after the fourth step, a contains the value $(g^{11})^2 \cdot g = g^{23}$. The running time of this algorithm is proportional to the bit-length of x, which is efficient even for very large values of x.

4.4.3 The Key Agreement Protocol

The Diffie–Hellman protocol is shown in Figure 4.2. Alice selects an exponent x_A at random, computes $y_A = g^{x_A}$ modulo p, and sends y_A over an authenticated but otherwise insecure channel to Bob. Bob proceeds analogously, selects an exponent x_B at random, computes $y_B = g^{x_B}$ modulo p, and sends y_B to Alice. Then Alice computes the value

$$k_{AB} = y_B^{x_A} = (g^{x_B})^{x_A} = g^{x_B x_A}$$

modulo p, and Bob computes, analogously,

$$k_{BA} = y_A^{x_B} = (g^{x_A})^{x_B} = g^{x_A x_B}$$

modulo p. The simple but crucial observation is that

$$k_{AB} = k_{BA}$$

due to the commutativity of multiplication (in the exponent). In other words, Alice and Bob arrive at the same shared secret value which they can use as a secret key, or from which they can derive a key of appropriate length, for example using a so-called cryptographic hash function.

Intuitively, the security of this protocol relies on the observation that in order to compute k_{AB} from y_A and y_B, it seems that an adversary would have to compute either x_A or x_B, which is the discrete logarithm problem believed to be infeasible.

The Diffie–Hellman protocol can be nicely explained by a mechanical analog, as shown in Figure 4.3. The exponentiation operation (e.g. the operation $x_A \mapsto g^{x_A}$) can be thought of as locking a padlock, an operation that is easy to perform but impossible (in a computational sense) to invert. Note that the padlock in this analog has no key; once locked, it can not be opened anymore. However, a party can remember the open state of the lock (i.e., x_A). Alice and Bob can exchange their locked padlocks (i.e., y_A and y_B), keeping a copy in the open state. Then they can both generate the same configuration, namely the two padlocks interlocked. For the adversary, this is impossible without breaking open one of the two padlocks.

Alice	**insecure channel**	**Bob**
select x_A at random from $\{0,\ldots,p-2\}$		select x_B at random from $\{0,\ldots,p-2\}$
$y_A := g^{x_A} \pmod p$		$y_B := g^{x_B} \pmod p$
	$y_A \rightarrow$	
	$\leftarrow y_B$	
$k_{AB} := y_B^{x_A} \pmod p$		$k_{BA} := y_A^{x_B} \pmod p$

Figure 4.2 The Diffie–Hellman key agreement protocol. The prime p and the generator g are publicly known parameters.

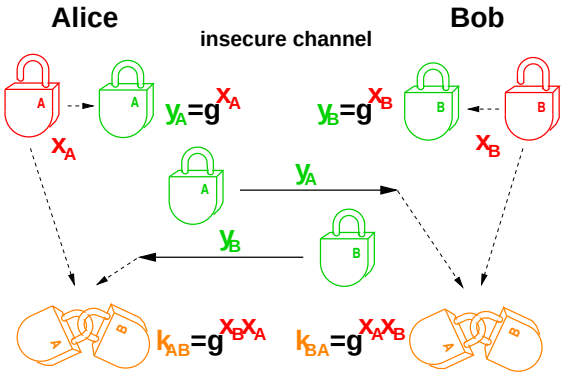

Figure 4.3 Mechanical analog of the Diffie–Hellman protocol.

The described Diffie–Hellman protocol can be generalized from computation modulo p (i.e., the group $\mathbf{Z}_p^*$) to any cyclic group $G = \langle g \rangle$, generated by a generator g, in which the discrete logarithm problem relative to the base g is computationally hard. The only modifications are that x_A and x_B must be selected from $\{0,\ldots,|G|-1\}$, and multiplication modulo p is replaced by the group operation of G. In practice, one often uses elliptic curves for which the discrete logarithm problem is believed to be even substantially harder than for the group $\mathbf{Z}_p^*$, for comparable group sizes.

4.5 Discrete Logarithms and Other Computational Problems on Groups

Let G be a cyclic group of order $|G| = n$ and let g be a generator of the group, i.e., $G = \langle g \rangle$. Then G is isomorphic to the (additively written) group $\langle \mathbf{Z}_n, + \rangle$, i.e., the

set $\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$ with addition modulo n. This is the standard representation of a cyclic group of order n. We can define the following three computational problems for G:

- The *Discrete Logarithm (DL) problem* is, for given (uniformly chosen) $a \in G$, to compute x such that $a = g^x$.
- The *Computational Diffie–Hellman (CDH) problem* is, for given (uniformly chosen) $a, b \in G$, to compute g^{xy}, where $a = g^x$ and $b = g^y$.
- The *Decisional Diffie–Hellman (DDH) problem* is, for given three elements $a, b, c \in G$, with a, b chosen uniformly at random (again as $a = g^x$ and $b = g^y$), to distinguish the setting where $c = g^{xy}$ from the setting where c is a third independent random group element.

The DL problem is that of making the abovementioned isomorphism between G and $\langle \mathbf{Z}_n, + \rangle$ explicit. Whether this is computationally easy or hard depends on the representation of the elements of G. It is easy to see that if one can compute discrete logarithms in G for some generator g, then one can compute them for any other generator g'. This is achieved by division by the DL of g' relative to generator g.

The DDH problem is at most as hard as the CDH problem, and the CDH problem is at most as hard as the DL problem. To see the latter, we only need to observe that the CDH problem can be solved by computing x from a, which means to compute the discrete logarithm. It is also known that if one could efficiently solve the CDH problem, then one could also efficiently solve the DL problem for almost all groups, i.e., the two problems are roughly equally hard (Maurer–Wolf, 1999) (see also Section 4.8.7).

We briefly discuss the cryptographic significance of these problems. It appears that breaking the Diffie–Hellman protocol means precisely to solve the CDH problem. However, it is possible that computing parts of the key $g^{x_A x_B}$ is easy, even though the CDH problem, i.e., computing the entire key, is hard. If an adversary could obtain part of the key, this could also be devastating in certain applications that make use of the secret key. In other words, even if the CDH problem is hard, a system making use of the Diffie-Hellman protocol could still be insecure. The (stronger) condition one needs for the Diffie-Hellman protocol to be a secure key agreement protocol in any context is that the DDH problem be hard, which means that a Diffie-Hellman key is indistinguishable (with feasible computation) from a random key. This implies in particular that no partial information about the key can leak.

4.6 Discrete Logarithm Algorithms

4.6.1 Introduction

In order to compute in a group G, one must represent the elements as bit-strings. For example, the elements of $\mathbf{Z}_p^*$ are typically assumed to be represented as integers (in binary representation). As mentioned above, the hardness of a problem generally strongly depends on the representation. In a concrete setting, for example when trying to break the Diffie–Hellman protocol, the adversary has to work in the given fixed representation which Alice and Bob are using. The hope of cryptographers is that for this representation, the problem is hard.

One can distinguish between two types of DL algorithms. A *generic* algorithm works independently of the representation, i.e., it makes use only of the group operations. A simple example of a generic algorithm is the trivial algorithm that tries all possible values $x = 0, 1, 2, 3, \ldots$, until $a = g^x$.

In contrast, a *special-purpose* algorithm is designed for a specific type of representation. For example, the best algorithms for the group $\mathbf{Z}_p^*$ are special-purpose and much faster than any generic algorithm. For the group $\mathbf{Z}_p^*$, the best known algorithm is the so-called number-field sieve (Gordon, 1993), which has complexity

$$O(e^{c(\ln p)^{1/3}(\ln\ln p)^{2/3}})$$

for $c = 3^{2/3}$, which is much faster than the generic algorithms discussed below, but still infeasible for large enough primes. Such a special-purpose algorithm can make use of the fact that the group elements are numbers and hence embedded in the rich mathematical structure of the integers $\mathbf{Z}$. For example, one can try to factor a number into its prime factors and combine such factorizations.

There are groups, for instance most elliptic curves over finite fields, for which no faster than generic algorithms are known. In other words, it is not (yet) known how the representation of the elements of an elliptic curve can be exploited for computing discrete logarithms.

Proving a super-polynomial complexity lower bound for any special-purpose algorithm for computing the DL in a certain group would resolve the most famous open conjecture in theoretical computer science, as it would imply $\mathbf{P} \neq \mathbf{NP}$, and is therefore not expected to be achievable in the near future. However, as we will see, one can prove exponential lower bounds for any *generic* algorithm.

It should be mentioned that Peter Shor (1994) discovered a fast (polynomial-time) algorithm for computing discrete logarithms and for factoring integers on a quantum computer. A quantum computer is a (still) theoretical model of computation which exploits the laws of quantum physics and hence is potentially much more powerful than classical computers (including the Turing machine) whose implementation makes use only of classical physics. Whether quantum computers

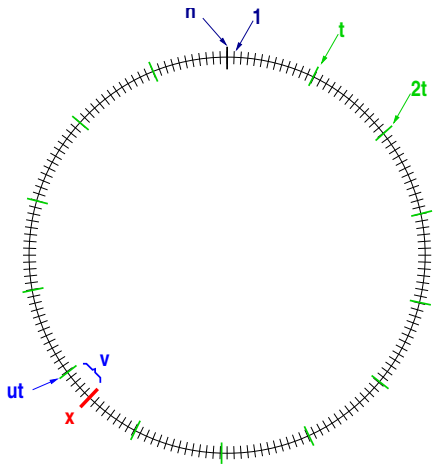

Figure 4.4 Illustrating the baby-step giant-step (BSGS) algorithm.

can ever be built is not known, but the research effort spent towards building one is enormous.

4.6.2 The Baby-step Giant-step Algorithm

The baby-step giant-step (BSGS) algorithm is the simplest non-trivial DL algorithm. It is generic, i.e., it works no matter how the group elements are represented. The group order $|G| = n$ need not be known; it suffices if a rough estimate of n is known.

The BSGS algorithm works as follows. Let $a = g^x$ be the given instance. Let t be a parameter of the algorithm, the typical choice being $t \approx \sqrt{n}$. The unknown value x can be represented uniquely as

$$x = ut - v$$

(see Figure 4.4) where $u < n/t \approx \sqrt{n}$ and $v < t$. The baby-step giant-step algorithm consists of the following steps:

(1) **Giant steps:** Compute the pairs (j, g^{jt}) for $0 \leq j < n/t$, sort these pairs according to the second value g^{jt}, and store them in a (sorted) table.
(2) **Baby steps:** Compute ag^i for $i = 0, 1, 2, \ldots$ until one of these values is contained in the (giant-step) table. This will happen when $i = v$, and the value retrieved from the table will be $j = u$. Compute $x = jt - i$.

The memory requirement for this algorithm is $O(n/t)$ which is $O(\sqrt{n})$ for $t = O(\sqrt{n})$. The time complexity is $O(\frac{n}{t} \log \frac{n}{t})$ for sorting the table and $O(t \log \frac{n}{t})$ for

accessing the table $O(t)$ times, hence $O(\max(t, \frac{n}{t})\log n)$, which is $O(\sqrt{n}\log n)$ for $t = O(\sqrt{n})$. We use the common notation $O(f(n)$ to say that a quantity grows asymptotically as $f(n)$ for increasing n.

The BSGS algorithm is an essentially optimal generic algorithm if n is a prime (see Section 4.8.3). If n has only small prime factors, then a significantly faster generic algorithm exists, which is discussed next.

4.6.3 The Pohlig–Hellman Algorithm

Let again $|G| = n$ and let q be a prime factor of n. One can write x as

$$x = uq + v$$

for some u and v which we wish to compute. Let $k := n/q$. Then $kx = kuq + kv$ and hence (since $kq = n$)

$$kx \equiv_n kv$$

which implies[2]

$$a^k = g^{kx} = g^{kv} = (g^k)^v.$$

In other words, v is the DL of a^k in the group $\langle g^k \rangle$, which has order q. Any generic algorithm, in particular the BSGS algorithm, can be used to compute this DL. The running time is $O(\sqrt{q}\log q)$.

It remains to compute u. Let

$$a' := ag^{-v} = g^{uq} = (g^q)^u.$$

Hence u is the DL of a' in the group $\langle g^q \rangle$ (which has order $k = n/q$) and can be computed using the same method as described above, but now for a group of order n/q.

Repeating this procedure for every prime factor of n (as many times for a prime q as it occurs in n), we obtain x, as desired. If the prime factorization of n is $n = \prod_{i=1}^{r} q_i^{\alpha_i}$, then the overall complexity of this algorithm is

$$O\left(\sum_{i=1}^{r} \alpha_i \sqrt{q_i}\log q_i\right) \;=\; O(\sqrt{q'}\log n),$$

where q' is the largest prime factor of n.

The Pohlig–Hellman algorithm is an essentially optimal generic algorithm (see Section 4.8.4). The existence of this algorithm is one of the reasons for choosing the group order of a DL-based cryptographic system to have a very large prime factor. For example, if one uses the group $\mathbf{Z}_p^*$, as in the original proposal by Diffie

[2] Here $a \equiv_n b$ means that a and b are congruent modulo n, i.e., n divides $a - b$.

and Hellman, then $n = p - 1$ must have a large prime factor q, for example $p - 1 = 2q$. In many cases one actually uses a group whose order is prime, also for other reasons.

4.7 Abstract Models of Computation

4.7.1 Motivation

As mentioned earlier, for general models of computation, no cryptographically useful lower-bound proofs are known for the complexity of any reasonable computational problem. It is therefore interesting to investigate reasonably restricted models of computation if one can prove relevant lower bounds for them.

In a restricted model one assumes that only certain types of operations are allowed. For example, in the so-called monotone circuit model one assumes that the logical (i.e., digital) circuit performing the computation consists only of AND-gates and OR-gates, excluding NOT-gates. However, a lower-bound proof for such a restricted model is uninteresting from a cryptographic viewpoint since it is obvious that an adversary can of course perform NOT-operations.

Some restricted models are indeed meaningful in cryptography, for example the generic model of computation. The term generic means that one can not exploit non-trivial properties of the representation of the elements, except for two generic properties that any representation has:

- One can test equality of elements.
- One can impose a total order relation $\preceq$ on any representation, for example the usual lexicographic order relation on the set of bit-strings.[3]

We now propose a model of computation that allows to capture generic algorithms and more general restricted classes of algorithms. This model serves two purposes. It allows to phrase generic algorithms in a clean and minimal fashion, without having to talk about bit-strings representing group elements, and it allows to prove lower bounds on the complexity of any algorithm for solving a certain problem in this model.

4.7.2 The Computational Model

We consider an abstract model of computation (see Figure 4.5) characterized by a black-box **B** which can store values from a certain set S (e.g. a group) in internal registers $V_1, V_2, \ldots, V_m$. The storage capacity m can be finite or unbounded.

[3] This order relation is abstract in the sense that it is not related to any meaningful relation (like $\leq$ in $\mathbf{Z}_n$) on the set S. An algorithm using this relation must work no matter how $\preceq$ is defined, i.e., for the worst case in which it could be defined. It can for instance be used to establish a sorted table of elements of S, but it cannot be used to perform a binary search in S.

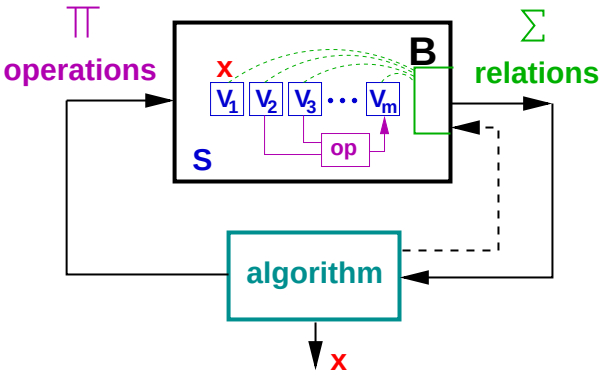

Figure 4.5 The abstract model of computation. An algorithm solving the extraction problem must, after querying the black-box **B** a certain number of times (say, k times), output the value x stored in the first register of the black-box.

The initial state encodes the problem instance and consists of the values of $V_1, \ldots, V_d$ (for some $d < m$; usually d is 1, 2, or 3), which are set according to some probability distribution (e.g. the uniform distribution).

The black-box **B** allows two types of operations, computation operations on internal state variables (shown on the left side of Figure 4.5) and queries about the internal state (shown on the right side of Figure 4.5). No other interaction with **B** is possible. We give a more formal description of these operations:

Computation operations For a set Π of operations on S of some arities (nullary, unary, binary, or higher arity), a computation operations consist of selecting an operation $f \in \Pi$ (say t-ary) as well as the indices $i_1, \ldots, i_{t+1} \leq m$ of $t+1$ state variables. **B** computes $f(V_{i_1}, \ldots, V_{i_t})$ and stores the result in $V_{i_{t+1}}$.[4]

Relation queries For a set Σ of relations (of some arities) on S, a query consist of selecting a relation $\rho \in \Sigma$ (say t-ary) as well as the indices $i_1, \ldots, i_t \leq m$ of t state variables. The query is answered by $\rho(V_{i_1}, \ldots, V_{i_t})$.[5]

For example, S could be the set $\mathbf{Z}_n = \{0, \ldots, n-1\}$, Π could consist of two operations, inserting constants and adding values modulo n, and Σ could consist of just the equality relation (which means that $\rho(V_{i_1}, V_{i_2}) = 1$ if and only if $V_{i_2} = V_{i_1}$) or, possibly, also the product relation (which means that $\rho(V_{i_1}, V_{i_2}, V_{i_3}) = 1$ if and only if $V_{i_3} = V_{i_1} \cdot V_{i_2}$).

[4] If m is unbounded, then one can assume without loss of generality that each new result is stored in the next free state variable; hence i_{t+1} need not be given as input.

[5] Here we consider a relation ρ, without loss of generality, to be given as a function $S^t \to \{0,1\}$.

This model captures two aspects of a restricted model of computation. The computation operations describe the types of computations the black-box can perform, and the state queries allow to model precisely how limited information about the representation of elements in S can be used.

A black-box $\mathbf{B}$ (i.e., a particular model of computation) is thus characterized by S, Π, Σ, m, and d. We are primarily interested in the generic algorithm setting where Σ consists of just the equality relation: $\Sigma = \{=\}$. We only consider the case where m is unbounded.

4.7.3 Three Types of Problems

We consider three types of computational problems for this black-box model of computation, where the problem instance is encoded into the initial state $(V_1, \ldots, V_d)$ of the device.

Extraction Extract the initial value x of V_1 (where $d = 1$). (See Figure 4.5.)

Computation Compute a function $f : S^d \to S$ of the initial state within $\mathbf{B}$, i.e., the algorithm must achieve $V_i = f(x_1, \ldots, x_d)$ for some (known) i, where $x_1, \ldots, x_d$ are the initial values of the state variables $V_1, \ldots, V_d$.

Distinction Distinguish two black-boxes $\mathbf{B}$ and $\mathbf{B}'$ of the same type with different distributions of the initial state $(V_1, \ldots, V_d)$.[6]

For the extraction problem, one may also consider algorithms that are allowed several attempts at guessing x. One can count such a guess as an operation; equivalently, we can assume without loss of generality that the algorithm, when making a guess, inputs the value (as a constant) to the black-box and wins if it is equal to x. We will take this viewpoint in the following, i.e., a guess is treated like a constant operation.

4.8 Proving Security: Complexity Lower Bounds

4.8.1 Introduction

One of the main goals of research in cryptography is to prove the security of cryptographic schemes, i.e., to prove that a scheme is hard to break. Unfortunately, not a single cryptographically relevant computational problem is known for which one can prove a significant lower bound on the complexity for a general model of computation. Such a proof would be a dramatic break-through in computer science, maybe comparable to the discovery of a new elementary particle in physics, and

[6] The performance of a distinguishing algorithm outputting a bit can be defined as the probability of the guess being correct, minus $\frac{1}{2}$. Note that success probability $\frac{1}{2}$ can trivially be achieved by a random guess; hence only exceeding $\frac{1}{2}$ can be interpreted as real performance.

would almost certainly yield the equivalent of the Nobel Prize in computer science, the Turing Award.[7]

However, we can prove lower bounds in the abstract model of computation discussed above. More precisely, we are interested in proving a relation between the number of operations an algorithm performs and its performance. For the extraction and the computation problems, the performance is defined as the algorithm's success probability. We do not consider the computing power that would be required in an implementation (of a black-box algorithm) for determining its next query; we only count the actual operations the algorithm performs.

When proving lower bounds, we will be on the safe side if we do not count the relation queries, i.e., if we assume that they are for free. In other words, we assume that any satisfied relation in the black-box (typically an equality of two register values, called a collision), is reported by the black-box, without requiring the algorithm to ask a query. Note that when designing actual algorithms in this model (as opposed to proving lower bounds), the relation queries are relevant and must be counted. (For example, a comparison in a conventional computer constitutes an operation and requires at least one clock cycle.)

In this section we consider a few concrete instantiations of the abstract model of computation to illustrate how lower bounds can be proved. Our primary interest will be to prove lower bounds for computing discrete logarithms in a cyclic group (the DL problem) and for the CDH and the DDH problems.

We introduce some notation. Let **Const** denote the set of constant (nullary) operations, which correspond to inserting a constant into (a register of) the black-box. For a given set Π of operations, let $\overline{\Pi}$ be the set of functions on the initial state that can be computed using operations in Π, i.e., it is the closure of Π. For example, if Π consists only of the increment function $x \mapsto x+1$, then $\overline{\Pi} = \{x \mapsto x+c \mid c \geq 1\}$ since by applying the increment function c times one can compute the function $x \mapsto x+c$ for any c.

The simplest case of an extraction problem is when $\Pi = \mathbf{Const}$ and $\Sigma = \{=\}$, i.e., one can only input constants and check equality. This is analogous to a card game where one has to find a particular card among n cards and the only allowed operation is to lift a card, one at a time. It is obvious that the best strategy for the extraction problem is to randomly guess, i.e., to input random constants, and the success probability of any k-step algorithm is hence bounded by $k/|S|$. However, if one could also query other relations, for example a total order relation $\leq$ on S, then much faster algorithms can be possible, for example binary search.

[7] The fact that the main award in computer science is named after Turing reflects the central role Turing has played in the early days of this field.

4.8.2 Two Lemmas

We need a lemma about the number of roots a multivariate polynomial modulo a prime q can have, as a function of its degree. The degree of a multivariate polynomial $Q(x_1,\ldots,x_t)$ is the maximal degree of an additive term, where the degree of a term is the sum of the powers of the variables in the term. For example, the degree of the polynomial

$$Q(x_1,x_2,x_3,x_4) \;=\; x_1^6 + 5x_1^3x_2x_3^4 + 2x_1x_2^3x_3x_4^2$$

is 8, the degree of the second term. The following lemma is known as the Schwartz–Zippel lemma (Schwartz, 1980). For the single-variable case, it corresponds to the well-known fact that a (single-variable) polynomial of degree d over a field can have at most d roots.

Lemma 4.8.1 *For a prime q and any $t \geq 1$, the fraction of solutions $(x_1,\ldots,x_t) \in \mathbf{Z}_q^t$ of any multivariate polynomial equation*

$$Q(x_1,\ldots,x_t) \equiv_q 0$$

of degree d is at most d/q.

This lemma tells us, for example, that the above polynomial $Q(x_1,x_2,x_3,x_4)$, if considered (say) modulo the prime $q = 101$, has at most a fraction $8/101$ of tuples (x_1,x_2,x_3,x_4) for which $Q(x_1,x_2,x_3,x_4) = 0$.

We also need the a second lemma. Consider a general system which takes a sequence $X_1,X_2,\ldots$ of inputs from some input alphabet $\mathscr{X}$ and produces, for every input X_i, an output Y_i from some output alphabet $\mathscr{Y}$. The system may be probabilistic and it may have state. For such a system one can consider various tasks of the following form. By an appropriate choice of the inputs $X_1,\ldots,X_k$, achieve that $Y_1,\ldots,Y_k$ satisfies a certain property (e.g., is all zero). In general, the task is easier to solve (i.e., the success probability is higher) if an *adaptive* strategy is allowed, i.e., if X_i is to be chosen only after Y_{i-1} has been observed. In contrast, a *non-adaptive* strategy requires $X_1,\ldots,X_k$ to be chosen at once.

Lemma 4.8.2 *Consider the task of preventing that a particular output sequence $y_1,\ldots,y_k$ occurs. The success probability of the best non-adaptive strategy is equal to that of the best adaptive strategy.*

Proof Any adaptive strategy A with access to $Y_1,Y_2,\ldots$ can be converted into an equally good non-adaptive strategy A' by feeding to A, instead of the actual values $Y_1,Y_2,\ldots$ output by the system, the (fixed) values $y_1,y_2,\ldots$, respectively. As long as A is not successful (in provoking a deviation of $Y_1,Y_2,\ldots$ from $y_1,y_2,\ldots$), these constant inputs $y_1,y_2,\ldots$ are actually correct and A and A' behave identically. As soon as A is successful (in achieving an output $Y_i \neq y_i$ for some i), so is A'. □

4.8.3 Group Actions and the Optimality of the Baby-Step Giant-Step Algorithm

Let S be a finite group of size $|S| = n$ with group operation denoted as $\star$. A group action on S is an operation of the form

$$x \mapsto x \star a$$

for some (constant) parameter a. For example, if S is the set of integer numbers, and $\star$ is addition (i.e., $+$), then this operation corresponds simply to incrementing x by a.

Theorem 4.8.3 *Let $\star$ be a group operation on S, let $\Pi = \mathbf{Const} \cup \{x \mapsto x \star a \mid a \in S\}$ consist of all constant functions and group actions, and let $\Sigma = \{=\}$. Then the success probability of every k-step algorithm for extraction is at most $\frac{1}{4}(k+1)^2/n$.*

Proof We use three simple general arguments which will also be reused implicitly later.

- First, we assume (conservatively) that, as soon as some collision occurs (more generally, some relation in Σ is satisfied for some state variables) in the black-box $\mathbf{B}$, the algorithm is successful. One can therefore restrict the analysis to algorithms for provoking some non-trivial collision in the black-box.
- Second, we observe, by considering the black-box as the system in Lemma 4.8.2, that if the only goal is to provoke a collision (i.e., an output sequence of $\mathbf{B}$ different from always outputting 'no collision'), then adaptive strategies are not more powerful than non-adaptive ones. Hence we can restrict our analysis to non-adaptive algorithms.
- Third, for lower-bound proofs we can assume that an algorithm can not only perform operations in Π but can, in every step, compute a function in $\overline{\Pi}$ (of the initial state $(V_1, \ldots, V_d)$). This can only improve the algorithm's power; a lower-bound proof for this model also holds for the weaker model. Without loss of generality we can assume that only distinct functions are chosen by the algorithm (since a trivial collision would not "count" and hence be useless).

In the setting under consideration, the composition of two operations in Π is again in Π, i.e., $\overline{\Pi} = \Pi$. For example, if the operation $x \mapsto x \star a$ is composed with the operation $x \mapsto x \star b$, then this corresponds to the operation $x \mapsto x \star c$ for $c = a \star b$.

For all $x \in S$ and distinct a and b we have $x \star a \neq x \star b$. Thus collisions can occur only between a (value computed by) function of the form $x \mapsto x \star a$ and a constant function c, namely if $x \star a = c$, which is equivalent to $x = c \star a^{-1}$. Let u [and v] be the number of constant operations [group actions] the algorithm performs. Then the probability of a collision is upper bounded by $u(v+1)/|S|$. The optimal choice is $u = \lceil k/2 \rceil$ (and $v = k - u$), for which $uv \leq (k+1)^2$. □

Example Consider the group $\langle \mathbf{Z}_{100}, + \rangle$. The black-box contains a random value between 0 and 99. If our strategy to provoke a collision computes the values $x+22$ and $x+53$ (in addition to x already stored in the black-box) and inserts the constants 34 and 87, then a collision occurs if (and only if) $x = 12$, $x = 34$, $x = 65$, $x = 81$, or $x = 87$. For example, if $x = 81$, then a collision occurs between the values $x+53$ and 34. Note that for this strategy we have $k = 4$ and indeed the success probability (for a random x) is $5/100$, which is less than $\frac{1}{4}5^2/100 = 1/16$.

The above bound implies that in order to achieve a constant success probability (e.g. $\frac{1}{2}$), the number k of operations must be on the order of $\sqrt{n}$.

The theorem illustrates that the baby-step giant-step (BSGS) algorithm is essentially optimal. It achieves the lower bound, even though it only requires the operations in $\Pi = \mathbf{Const} \cup \{x \mapsto x+1\}$, i.e., only increments by 1 (and not by general constants). The BSGS algorithm, if interpreted in this model, inserts equidistant constants with gap $t \approx \sqrt{n}$ and increments the secret value x until a collision with one of these values occurs.[8]

4.8.4 Discrete Logarithms and the Optimality of the Pohlig–Hellman Algorithm

We now consider the additive group $\langle \mathbf{Z}_n, + \rangle$. The extraction problem for this group corresponds to the discrete logarithm (DL) problem for a cyclic group of order n. In other words, every extraction algorithm for $\langle \mathbf{Z}_n, + \rangle$ is a generic DL algorithm for any group of order n, and vice versa. In the sequel, let q denote the largest prime factor of n. The following theorem is an abstract formulation of a result due to Nechaev (1994) and Shoup (1997).

Theorem 4.8.4 *For $S = \mathbf{Z}_n$, $\Pi = \mathbf{Const} \cup \{+\}$ and $\Sigma = \{=\}$, the success probability of every k-step extraction algorithm is at most $\frac{1}{2}(k+1)^2/q$.*

Proof We use the same line of reasoning as in the proof of Theorem 4.8.3. Every value computed in the black-box is of the form $ax+b$ for some (known) a and b. In other words,

$$\overline{\Pi} = \{ax+b \mid a,b \in \mathbf{Z}_n\},$$

i.e., only linear functions of x can be computed. As argued above, we need to consider only non-adaptive algorithms for provoking a collision. Since a collision modulo n implies that there is also a collision modulo q, a lower bound on the prob-

[8] Note that the number of equality checks is actually $O(n)$, which is too high. In order to reduce the number of equality checks, the BSGS algorithm makes use of the abstract order relation $\preceq$ to generate a sorted table of stored values.

ability of provoking a collision modulo q is also a lower bound on the probability of provoking a collision modulo n.

Consider hence a fixed algorithm (for provoking a collision modulo q) computing in each step (say the ith) a new value $a_i x + b_i$. A collision (modulo q) occurs if

$$a_i x + b_i \equiv_q a_j x + b_j$$

for some distinct i and j, i.e., if $(a_i - a_j)x + (b_i - b_j) \equiv_q 0$. This congruence has at most one solution for x modulo q (according to Lemma 4.8.1).[9] Therefore the total number of remainders of x modulo q for which any collision modulo q (which is necessary for a collision modulo n) can occur is bounded by $\binom{k}{2}$. Thus the fraction of x (modulo q, and hence also modulo n) for which a collision modulo q can occur is at most $\binom{k+1}{2}/q < \frac{1}{2}(k+1)^2/q$. (Recall that x is already in the black-box before the algorithm performs any operation.) □

For achieving a constant success probability, the number k of operation must hence be $O(\sqrt{q})$. The Pohlig–Hellman algorithm requires $k = O(\sqrt{q}\log n)$ operations and matches this bound up to a factor $\log n$, which is due to the fact that in the lower-bound proof we were too generous with the algorithm by not counting the equality queries. It should be mentioned that Pollard (1978) proposed a much more practical algorithm with comparable complexity but which requires essentially no memory. However, the running time analysis of this algorithm is based on heuristic arguments.

The reader may want to prove, as an exercise, a lower bound for DL computation, even if one assumes the additional availability of a DDH oracle. This can be modeled by including in the set Σ of relations the product relation $\{(a,b,c)|ab \equiv_n c\}$.

4.8.5 Product Computation in $\mathbf{Z}_n$ and the CDH Problem

We now consider the computation problem for the product function $(x,y) \mapsto xy$ in $\mathbf{Z}_n$. This corresponds to the generic computational Diffie–Hellman (CDH) problem in a cyclic group of order n. In other words, every algorithm in the black-box model for computing $(x,y) \mapsto xy$, when $S = \mathbf{Z}_n$, $\Pi = \mathbf{Const} \cup \{+\}$ and $\Sigma = \{=\}$, is a generic CDH algorithm for any cyclic group of order n, and vice versa. The following theorem shows that for generic algorithms, the DL and the CDH problems are comparably hard.[10]

[9] Namely, if $a_i \equiv_q a_j$ and hence $b_i \equiv_q b_j$, there is no solution, and if $a_i \not\equiv_q a_j$, then the single solution modulo q is $(b_j - b_i)/(a_i - a_j)$, computed modulo q.

[10] For general algorithms, the statement that they are comparably hard requires a proof that the DL problem can be reduced to the CDH problem, see Section 4.8.7.

Theorem 4.8.5 *For $S = \mathbf{Z}_n$, $\Pi = \mathbf{Const} \cup \{+\}$ and $\Sigma = \{=\}$ the success probability of every k-step algorithm for computing the product function is at most $(\frac{1}{2}k^2 + 4k + 5)/q$.*

Proof Again, to be on the safe side, we can assume that as soon as a collision modulo q occurs among the computed values, the algorithm is successful. We have

$$\overline{\Pi} = \{ax + by + c \mid a, b, c \in \mathbf{Z}_n\}.$$

In addition to the computed values (and x and y), the black-box is assumed to contain already the value xy in a register that can not be used as the input to operations (but is considered when checking for collisions). There are two types of collisions:

$$a_i x + b_i y + c_i \equiv_q a_j x + b_j y + c_j$$

for some $i \neq j$, and

$$a_i x + b_i y + c_i \equiv_q xy$$

for some i. The fraction of x for which a collision of the first type can occur is bounded by $\binom{k+2}{2}/q$ (the two values x and y are already contained in the black-box), and the fraction of x for which a collision of the second type can occur is bounded by $2(k+2)/q$ (according to Lemma 4.8.1), as the degree of the equation $a_i x + b_i y + c_i - xy \equiv_q 0$ is 2. Hence the total fraction of x (modulo n) for which one of the collision events (modulo q) occurs is bounded by $\left(\binom{k+2}{2} + 2(k+2)\right)/q = (\frac{1}{2}k^2 + 4k + 5)/q$. □

4.8.6 The DDH Problem

For the DDH problem one can prove a lower bound of $O(\sqrt{p'})$ for any algorithm with a significant distinguishing advantage, where p' is the *smallest* prime factor of n. This can again be shown to be tight, i.e., there does exist a generic algorithm for the DDH problem with complexity essentially $\sqrt{p'}$. For example, the DDH problem for the group $\mathbf{Z}_p^*$ for any large prime p is trivial since the order of $\mathbf{Z}_p^*$ is $p-1$ and contains the small prime factor $p' = 2$. The theorem implies that for (large) groups of prime order, the DDH problem is very hard for generic algorithms.

Theorem 4.8.6 *For $S = \mathbf{Z}_n$, $\Pi = \mathbf{Const} \cup \{+\}$ and $\Sigma = \{=\}$, the advantage of every k-step algorithm for distinguishing a random triple (x, y, z) from a triple (x, y, xy) is at most $(k+3)^2/p'$, where p' is the smallest prime factor of n.*

Proof Again we can assume that, as soon as a collision occurs among the values computed in the black-box, the algorithm is declared successful. (One could imagine a genie sitting in the black-box who announces the correct answer, namely

which configuration the black-box is in, as soon as any non-trivial collision occurs.) Hence it suffices to compute the probabilities, for the two settings, that a collision can be provoked by an algorithm, and take the larger value as an upper bound for the distinguishing advantage of the two settings. We only analyze the case where the initial state is (x,y,xy), as the collision probability is larger in this case. The set $\overline{\Pi}$ of computable functions is

$$\overline{\Pi} = \{ax+by+cxy+d \mid a,b,c,d \in \mathbf{Z}_n\},$$

i.e., the ith computed value is of the form

$$a_i x + b_i y + c_i xy + d_i$$

for some a_i, b_i, c_i, d_i. For any choice of $(a_i,b_i,c_i,d_i) \neq (a_j,b_j,c_j,d_j)$ we must bound the probability that

$$a_i x + b_i y + c_i xy + d_i \equiv_n a_j x + b_j y + c_j xy + d_j.$$

This is a polynomial equation of degree 2. Since $(a_i,b_i,c_i,d_i) \neq (a_j,b_j,c_j,d_j)$ implies only that $(a_i,b_i,c_i,d_i) \not\equiv_r (a_j,b_j,c_j,d_j)$ for some prime divisor r of n, it could be the smallest one, i.e., $r = p'$. Hence the fraction of pairs (x,y) for which it is satisfied is at most $2/p'$. Therefore, the fraction of pairs (x,y) for which one of the $\binom{k+3}{2}$ relations is satisfied modulo p' is at most $\binom{k+3}{2}(2/p') < (k+3)^2/p'$. □

4.8.7 Generic Reduction of the DL Problem to the CDH Problem

We have seen that the CDH and the DL problems are roughly equally hard in the generic model. An important question is whether this is also true for general models of computation. In other words, we want to prove that, in a general model of computation, breaking CDH is as hard as solving the DL problem. Even though this question is not about our black-box model, it can be answered using this mode. One can show a generic reduction of the DL problem to the CDH problem, i.e., a generic algorithm which efficiently solves the DL problem, provided it has access to a (hypothetical) oracle that answers CDH challenges.

This is modeled by including multiplication modulo n in the set Π of allowed operations for the generic extraction problem, i.e., by considering the extraction problem for the ring $\mathbf{Z}_n$. The Diffie-Hellman oracle assumed to be available for the reduction implements multiplication modulo n. There exist such an efficient generic algorithm (Maurer, 1994) (see also Maurer–Wolf, 1999) for essentially all n and, under a plausible number-theoretic conjecture, for all n. In contrast to the algorithms presented in this article, this algorithm is quite involved and makes use of the theory of elliptic curves.

4.9 Conclusions

One may speculate what contributions to cryptography Turing might have made had his life not ended much too early and had he continued his research in cryptography. His contributions could have been on several fronts, including cryptanalysis. But what could perhaps have been the most important contribution is that Turing's strive for rigor might have helped advance cryptography from a fascinating discipline making use of various branches of mathematics to a science that is itself an axiomatically defined branch of mathematics. An attempt in this direction is described in Maurer–Renner (2011). One may doubt whether Turing could have removed the apparently unsurmountable barrier preventing us from proving cryptographically significant lower bounds on the complexity of computational problems (which would probably imply a proof of $\mathbf{P} \neq \mathbf{NP}$), but it's not inconceivable.

Acknowledgments

I am grateful to Andrew Hodges for sharing with me some of his insights about Turing's life, in particular about Turing's interest in cryptography, and to Divesh Aggarwal and Björn Tackmann for helpful feedback on this paper.

References

A. Church. A set of postulates for the foundation of logic, *Annals of Mathematics*, Series 2, **33**, 346–366 (1932).

W. Diffie and M. E. Hellman. New directions in cryptography, *IEEE Transactions on Information Theory*, **22** (6), 644–654 (1976).

D.M. Gordon. Discrete logarithms in $GF(p)$ using the number field sieve, *SIAM J. Discrete Mathematics*, **6** (1), 124–138 (1993).

A. Hodges. *Alan Turing: The Enigma*, London: Vintage Books (1992).

D. Kahn. *The code breakers, the story of secret writing*, New York: MacMillan (1967).

U. Maurer. Towards the equivalence of breaking the Diffie–Hellman protocol and computing discrete logarithms, *Advances in Cryptology – CRYPTO '94*, Lecture Notes in Computer Science, **839**, 271–281, Springer-Verlag (1994).

U. Maurer. Cryptography 2000 ± 10. In Lecture Notes in Computer Science **2000**, 63–85, R. Wilhelm (ed.), Springer-Verlag (2000).

U. Maurer. Abstract models of computation in cryptography. In *Cryptography and Coding 2005*, Lecture Notes in Computer Science, **3796**, 1–12, Springer-Verlag (2005).

U. Maurer and R. Renner. Abstract cryptography. In *The Second Symposium in Innovations in Computer Science, ICS 2011*, Tsinghua University Press, 1–21, (2011).

U. Maurer and S. Wolf. On the complexity of breaking the Diffie-Hellman protocol, *SIAM Journal on Computing*, **28**, 1689–1721 (1999).

V.I. Nechaev. Complexity of a deterministic algorithm for the discrete logarithm, *Mathematical Notes*, **55** (2), 91–101 (1994).

S.C. Pohlig and M.E. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance, *IEEE Transactions on Information Theory*, **24** (1), 106–110 (1978).

J.M. Pollard. Monte Carlo methods for index computation mod p, *Mathematics of Computation*, **32**, 918–924 (1978).

R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM*, **21** (2), 120–126 (1978).

J.T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities, *Journal of the ACM*, **27** (3), 701–717 (1980).

C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, **27**, 379–423; 623–656 (1948).

C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, **28**, 656–715 (1949).

P.W. Shor. Algorithms for quantum computation: discrete log and factoring. In *Proc. 35th IEEE Symposium on the Foundations of Computer Science (FOCS)*, 124–134, IEEE Press (1994).

V. Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology – EUROCRYPT '97*, Lecture Notes in Computer Science, **1233**, 256–266, Springer-Verlag (1997).

S. Singh, *The Code Book*, Fourth Estate, London (1999).

A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Ser. 2, **42**, 230–265 (1937).