A number of common subjects in security and cryptography literature are briefly explained: primitives, protocols, encryption, authorization, authentication, probabilistic algorithms, oblivious transfer, adversary models, secure multiparty computation and zero-knowledge proofs.

Chapter 2

Preliminaries

When one talks about security in the context of computers, there are roughly two types of security to distinguish. *Computer security* is concerned with the protection of the computer itself, typically against viruses, malware, Trojan horses and hacker attacks. *Information security* is concerned with the protection of valuable information that may reside on or pass through a computer, typically against unauthorized manipulation and theft. In practice these two types of security are closely related: when a computer is compromised, the information that resides on the computer is no longer secure against theft and unauthorized manipulation. The two types of security can be independent targets of crime: many viruses only compromise computers to send out spam, and not to steal information; information can be stolen without compromising a computer.

In this thesis, we focus on the *information security* type of security.

The discipline of *cryptography* is concerned with hiding the meaning of a message (rather than hiding its existence). There are roughly two subdisciplines of cryptography. One focuses on *cryptographic primitives*, which are algorithms which transform information in cryptographically relevant ways. Examples of cryptographic primitives are encryption and cryptographic hashes. The other subdiscipline focuses on *cryptographic protocols*, also called *security protocols*, which are communication methods that use cryptographic primitives. Typical goals of a cryptographic protocols include *authentication* (proving that you are who you claim to be) and secure communication (the messages sent cannot be interpreted or modified by an adversary).

In this thesis, we focus on *cryptographic protocols*. As cryptographic primitives are the building blocks of cryptographic protocols, they figure frequently. We *design* new protocols, and *use* primitives.

The remainder of the current chapter introduces and explains a vast number of concepts and subjects common to cryptography which are used throughout this thesis. One cryptographic primitive is so essential for this thesis, that a separate chapter is devoted to it: Chapter 3 is about cryptographic hash functions. A particular tool for analysis of cryptographic protocols is so essential for this thesis, that a number of chapters is devoted to it: Chapters 4–6 are about *authentication logics*.

2.1 Encryption

An *encryption algorithm* is an algorithm that transforms an intelligible message (also called plaintext) into an unintelligible message (also called ciphertext). To an encryption algorithm belongs also a *decryption algorithm* which transforms the ciphertext back into the plaintext. To prevent that anybody can decrypt every ciphertext, encryption and decryption algorithms use *keys*. A key is a piece of information that is essential for either successful encryption or successful decryption of a message. Thus, to protect a piece of ciphertext from being decrypted, one only has to keep the *decryption key* secret.

Keeping a key secret only makes sense when it is difficult to guess the key correctly. For example, if a particular algorithm would only support two different keys, an adversary could simply try both keys, and find out which one reveals an intelligible message. Therefore, encryption and decryption algorithms use keys that stem from a very large domain. The domain has to be this large, that it is infeasible to try out all keys, or even any reasonable fraction of all keys. Trying out all possible keys until one works is called a *brute-force attack*. Trying out keys which stem from a precompiled list is called a *dictionary attack*.[1]

Ciphertext messages are unintelligible, but that does not mean that one cannot infer anything from a ciphertext. Most importantly, the length of a ciphertext is often closely related to the length of the corresponding plaintext. This is called *message length revealing* (as opposed to *message length concealing*).[2]

For some encryption and decryption algorithms the *encryption key* (the key used to create ciphertext) and the *decryption key* (the key used to reveal the plaintext) are equal. This is called *symmetric encryption*. For other encryption and decryption algorithms the keys are not the same. This is called *asymmetric encryption*. In asymmetric encryption, the encryption key is also called the *public key*, and the decryption key is also called the *private key*.

Somebody, say, a person named Whitfield, can create a public key/private key pair, and publish his public key. Anybody who wants to send a ciphertext message to Whitfield that only Whitfield can decrypt, can simply encrypt his plaintext with Whitfield's public key. Using some tricks which go beyond the scope of this explanation, Whitfield can also send a message to anyone using

[1] The terms brute-force attack and dictionary attack do apply to the process of guessing any secret information, not just decryption keys.

[2] For a comprehensive discussion of what a ciphertext may reveal, consult [AR02].

his private key, in such a way that anybody who knows the public key can verify that Whitfield sent the message originally. This is called a *cryptographic signature*.

A good introduction to cryptography is [Sin99]. For more technical and in-depth explanations, consult [Sch96] (slightly outdated) or [FS03].

2.2 Authorization and Authentication

Cryptographic protocols are often used where *access control* is of concern, in order to limit who can view or modify particular information. The process of establishing whether someone should be given access to information is called *authorization*.

The difficulty of authorization is often not on the implementation side (does the implemented policy reflect the actual policy?[3]) but on the policy side (does the actual policy reflect the intention of the policy?[4]). For example, to protect medical information, a policy could consist of only granting physicians access to medical files. A system that does just this perfectly can still fail in its objective to protect the medical files against misuse, as the policy is arguably too tolerant: it grants every physician access to every medical file, not just to the files of his clients.

A simple and often-used method for authorization is an *access list*: a list of persons who are granted access (for example, a list of a specific group f physicians). To actually get access, someone has to identify himself (which is called *authentication*) and it has to be verified whether the person is on the list. To some it seems that authentication is necessary for authorization, and that therefore these terms can be used interchangeably, but that is certainly not the case. Authorization and authentication can occur independently of one another. This can best be explained with two examples. To get physical access to the interior of a car, one has to put the correct key into the lock and turn it (authorization). When a police officer obliges you to identify yourself, you have to present your passport or drivers' license, and doing so does not result in any access whatsoever[5] (authentication).

To complicate matters even further, *authentication* has a number of other related meanings, depending on the context. To authenticate a *message* is to verify that a message has not been tampered with. In Chapter 8, we coin the term *knowledge authentication* for proving that someone has particular knowledge without disclosing it. This may seem distant from authentication in the sense of proving that you are who you claim you are, but it is not. In cryptographic protocols, the most common way to prove you are who you claim you are is to prove that you have a particular private key without disclosing it. As such, *knowledge authentication* is a generalization of 'just' *authentication*.

---

[3] Answering this question is called *verification* of a system.

[4] Answering this question is called *validation* of a system or policy.

[5] Nevertheless, failing to identify (authenticate) oneself may result in access to the interior of the police office.

2.3 Complexity

A central theme in computer science is *complexity*, and this theme figures in virtually every discipline of computer science. The fundamental field of *complexity theory* is so extremely important and intricate, that one single section in an introductory chapter can only explain a few very basic concepts. The purpose of this section is to give the reader who has no background in computer science an idea of what *complexity* is about, and to refresh the memory of the reader whose years in college may have drifted from the mind.[6]

Complexity theory is the study of which amount of resources an algorithm requires, as a function of the size of the input (the 'problem description'). The most important resources are *time* and *memory*. The time assessment of an algorithm is called its *computational complexity* and its memory complexity is called its *space complexity*.

Complexities can be determined for a particular algorithm, but also for the fundamental problem that an algorithm solves. The complexity *of a particular problem* is a property of the set of *all algorithms* that solve the particular problem. More precisely, the complexity of a particular problem is a measure of the amount of resources that the most efficient algorithm for that particular problem requires.

Using complexity theory, one can assess whether it is feasible to use a particular algorithm to solve a particular problem of a particular size. One can also assess whether feasible algorithms for a particular problem exist at all. Whether an algorithm is feasible obviously depends on the amount of available resources.

A notorious class of infeasible problems is the class of NP-complete problems. Informally, an NP-complete problem is a problem in which a solution has to be found, and a solution can be *verified* to be correct in a number of steps which is polynomial in the size of the input. However, it may be that the only way to *find* a correct solution is to try *all possible* solutions (of polynomial length). An example of an NP-complete problem is the subset sum problem: for a given, finite list of integers, find out whether any subset of the integers sums up to zero. For any given subset, it is very easy to verify whether it sums up to zero. But there is no algorithm known that determines whether such a subset exists that is significantly faster than trying every possible subset, which is very slow. NP-complete problems are so difficult to solve, that by increasing the problem size slightly, the resource requirements increase dramatically.

NP-complete problems play a central role in cryptography: an encryption algorithm is only considered good if to construct the plaintext from the ciphertext, without access to the decryption key, is an NP-complete problem.

Complexity does not only apply to algorithms, but also to communication protocols. The *communication complexity of a particular protocol* is a measure of how many bits are exchanged in a protocol run. A protocol can be seen as a way to compute a particular function $f(X, Y)$ (the 'problem'), where X and Y are

[6] For a thorough treatment of complexity, consult [BDG88].

```
trivial-prime-test(n)
   for i = 2 to squareroot(n) do
      if (divisor(i, n)) then return composite;
   return prime;
```

FIGURE 2.1: A trivial primality testing algorithm. This algorithm gives a definite answer to the question whether n is prime. The algorithm could be optimized somewhat, but the running time remains $O(\sqrt{n})$.

```
miller-rabin-prime-test(n, k)
   for i = 1 to k do
      set w to some randomly chosen number 0 < w < n;
      if (not witness(w, n)) then return composite;
   return prime;
```

FIGURE 2.2: The Miller-Rabin primality testing algorithm. The accuracy of this algorithm depends on the security ('certainty') parameter k, a higher value of k will increase the accuracy. When this algorithm answers that a number is composite, then it is indeed composite. On every 4^k occasions that this algorithm answers a number is prime, the expected number of errors is at most one. The function `witness` is a subroutine performing some specific arithmetic test between w and n. The running time of this algorithm is $O(k \ \ln \ln \ln n)$.

known to separate parties. The *communication complexity of a particular problem* is a measure of how many bits need to be exchanged between the parties in the most efficient protocol that computes $f(X, Y)$ [Yao79, Kus97].

2.4 Probabilistic Algorithms

For many computational problems, it is very easy to specify how they should be solved. For example, a program that gives a definite answer to the question whether a specific number is prime, is only a few lines long (see Figure 2.1). For large numbers, this primality testing algorithm would take a prohibitively long time to compute. It is possible to improve dramatically on the computation time of this test, if we allow the test to be wrong in its answer in only a negligible fraction of the occasions it is invoked. Such an algorithm is a *probabilistic* algorithm, and it requires access to some source of randomness (like a virtual coin which it may flip).

Probabilistic algorithms have the very tricky property that they typically detect *the opposite* of the property one is interested in. When it fails to detect the opposite, it guesses the affirmative to be the case. A well known example of a probabilistic algorithm is the Miller-Rabin primality test [Mil75, Rab80] (shown in Figure 2.2). This algorithm tries to find proofs of compositeness[7] of a number n. If it fails to find such a proof for a sufficiently long time, it will

[7] Compositeness is the opposite of primality.

assume that n is a prime. This is not just some 'guess', it can be proven that the chance that the assumption is wrong can be made arbitrarily small, depending on the time the algorithm invests in finding proofs of the opposite.

Thus, a probabilistic algorithm is an algorithm that may give the wrong answer, but only in very few cases. Probabilistic algorithms are employed because they are often much much faster than non-probabilistic algorithms. Almost all algorithms used in practice in cryptography are probabilistic.

2.5 Oblivious Transfer

Oblivious transfer (OT), introduced by Rabin [Rab81] is a type of protocol in which the sender sends a bit with probability $1/2$, and remains oblivious whether it was received. Even, Goldreich and Lempel generalized this type of protocol to *1-out-of-2* oblivious transfer [EGL85]. In *One out of two* oblivious transfer, the sender sends two messages, and the receiver can choose to read one of the messages. The receiver can read that single part of the message, but not the other. *What* message has been chosen, remains secret to the sender.

This rather weird type of protocol has a wide range of applications. For example, it can provide anonymity to buyers of digital goods. The seller (the sender in the protocol) cannot determine what the buyer has bought, but the seller can verify the item is paid for and only one single item is delivered [AIR01]. Oblivious transfer also has applications in auctions [Lip04]. In general, oblivious transfer is a building block for more complex protocols. Therefore, though oblivious transfer is technically a kind of cryptographic protocol, it is often considered a cryptographic primitive.

2.6 Adversary Models

The strength of a security protocol depends on the strength of the party that tries to break the protocol. Thus, when assessing the security of a protocol, one has to make assumptions about what the *adversary* is willing to do and about what he is capable of. Such an assumption is called an *adversary model* (or a *threat model*). There are three common adversary models:

honest The adversary is supposed to completely adhere to the protocol specification, and not to do anything more than the protocol specification.

honest-but-curious (HBC) The adversary is supposed to completely adhere to the protocol specification, but is allowed to perform extra calculations on the information it receives. This model is sometimes called the *semi-honest* adversary model. An attacker in this model is sometimes referred to as a *passive* attacker.

malicious (Dolev-Yao) The adversary is not required to adhere to the protocol specification, and is allowed to perform extra calculations on the information it receives. Moreover, the adversary is capable of intercepting

any message sent, and is capable of sending any message he is able to compose. An attacker in this model is sometimes referred to as an *active* attacker. [DY83]

In the honest-but-curious and the malicious adversary model, the adversary may perform calculations not specified in the protocol, but the amount of calculations is polynomially bounded. In particular, the adversary cannot perform brute-force attacks to find secret keys.

The honest adversary model is very weak, and therefore only erroneously used. Consider the following protocol for oblivious transfer:

> Alice sends Bob a message, but according to the protocol, Bob is allowed to only look at either the first half of the message, or the second half of the message.

This protocol is only secure if Alice knows that Bob is honest, that is in the *honest adversary model*. Trivially, this protocol is insecure in the honest-but-curious model.

When a protocol is insecure in the honest-but-curious adversary model, it is possible for some principal (i.e., a particupant in the protocol or an external observer) to obtain information that should be kept hidden from the principal. When a protocol is insecure in the malicious adversary model, it is possible for some principal to 'deceive' some other principal into obtaining false beliefs.

The malicious (Dolev-Yao) model is the strongest model, in the sense that if a protocol is secure in this model, then it is really very secure. As can be expected, it is rather difficult to prove a protocol to be secure in this model.

2.7 Secure Multiparty Computation

In his seminal paper "Protocols for Secure Computations" [Yao82], Yao has given a clear definition of what constitutes *secure multiparty computation* (SMC). Suppose there are two principals, Alice who possesses X and Bob who possesses Y. Alice and Bob are both interested in the value of some function $f(X, Y)$. A protocol for determining $f(X, Y)$ is an SMC if it satisfies the following conditions:

privacy The inputs X and Y are not mutually disclosed: Alice does not learn anything about Y except $f(X, Y)$, and Bob does not learn anything about X except $f(X, Y)$.

validity The protocol actually establishes $f(X, Y)$ and not something else. If one of the principals cheats, the other principal can detect this.

fairness Either both Alice and Bob learn $f(X, Y)$, or neither of them learns $f(X, Y)$. Essentially, the probability that one principal knows $f(X, Y)$ and withholds it from the other principal, is very small.

autarky Bob and Alice can determine $f(X,Y)$ without the assistance of a third party.[8]

Yao showed that if $f(X,Y)$ can be computed on a binary circuit that has m input wires and n gates, then there exists an SMC protocol with m oblivious transfers and communication complexity of $O(n)$, in a constant number of rounds [Yao86].[9] This may seem a promising result, but the feasibility of this solution is questionable at least. In 2004, eighteen years after publication of [Yao86], it was demonstrated that computing the rather trivial function $f(X,Y) = [X < Y]$ with $0 \leq X < 16$ and $0 \leq Y < 16$ would already take 1.25 seconds [MNPS04][10].

Feige, Kilian and Naor have proven that SMC is possible for any function $f(X,Y)$, if the condition of autarky is relaxed somewhat: their solution involves a third party, which can only learn the outcome of $f(X,Y)$, but nothing else. This third party is then supposed to honestly inform Alice and Bob about the outcome [FKN94]. The communication complexity of their solution, however, is not convincingly attractive, just as [Yao86].

SMC is not to be confused with *secure circuit evaluation* (SCE) [AF90], in which one principal provides the input X, and the other principal provides the function $f(\cdot)$. The aim in SCE is to compute $f(X)$ without mutual disclosure of X and $f(\cdot)$. Importantly, SMC is not a special case of SCE because in SMC it is known to both participants which function is computed. Conversely, SCE can be perceived as a special case of SMC, where the inputs are X and $g(\cdot)$, and where the function $f(X, g(\cdot))$ applies function $g(\cdot)$ to X.

2.8 Zero-Knowledge Proofs

Goldwasser, Micali and Rackoff introduced the concepts *zero-knowledge* and *interactive proof systems* in 1985 [GMR85].

Suppose again that there are two principals, now called Peggy (the prover) and Victor (the verifier). Peggy wants to convince Victor that she has some very special knowledge. For example, she knows the secret ingredients to make Coca-Cola[11], she can solve any Rubik's cube[12] (shown in Figure 2.3), or alternatively: she knows her own private key. Peggy wants to convince Victor, but without disclosing the special knowledge itself.

We will first focus more generally on protocols which convince Victor of Peggy's special knowledge. Protocols that convince Victor without disclosing the secret are a subclass of these protocols.

[8] The condition of autarky has not been explicitly named as such in [Yao82]. It is included here for clarity.

[9] This result has been generally accepted, but a proof has never been published.

[10] This is on two PCs with 2.4 GHz processors, and a communication link with 617.8 MBPS bandwidth and a latency of 0.4 ms. If the communication link is changed to a wide-area setting, e.g. a bandwidth of 1.06 MBPS and a latency of 237.0 ms, the computation time increases to 4.01 seconds.

[11] Coca-Cola is a registered trademark of The Coca-Cola Company.

[12] Rubik's Cube is a registered trademark of Seven Towns Ltd.

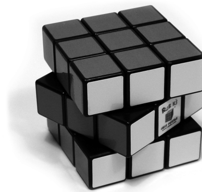

FIGURE 2.3: A Rubik's cube. Picture courtesy of Seven Towns Ltd.

Definition 2.1 (Interactive Proof Systems[13]). *An* interactive proof system *for a set S is a two-party game between a* verifier *executing a probabilistic polynomial-time (based on polynomial p) strategy (denoted V) and a* prover *executing a computationally unbounded strategy (denoted P), which satisfies the following conditions:*

completeness *For every $x \in S$ the verifier V always accepts after interacting with the prover P on common input x.*

soundness *For some polynomial p, it holds that for every $x \notin S$ and every potential strategy P^*, the verifier V rejects with probability at least $1/p(|x|)$, after interacting with P^* on common input x.*

The terms *soundness* and *completeness* have a meaning different from the meaning in logic (where it refers to the relation between logics and models).

The soundness condition may seem tricky or weak, but observe that when such a proof system is repeated $O(p(|x|)^2)$ times, the probability that V accepts for an $x \notin S$ reduces to $2^{-p(|x|)}$, which is 'close to zero'. The fact that the prover is not computationally bounded should not automatically be considered a structural problem either: the soundness criterion holds the prover to not cheating, and any practical implementation of an interactive proof will be forced to bounded computational resources by mere reality.

If Peggy would like to prove that she knows the solution for any Rubik's cube, she could challenge Victor for a scrambled Rubik's cube. Victor may choose or construct some scrambled Rubik's cube x, and hand it over to Peggy. Peggy could in turn, under the watchful eye of Victor, solve the puzzle. This rather trivial interactive proof is both complete and sound. It is complete because if Peggy knows the solution for x, she can solve it. Though Victor may not be proficient in solving Rubik's cubes, he can easily verify Peggy's solution. This makes the strategy sound. This trivial protocol which proves Peggy's knowledge of how to solve a Rubik's cube, discloses the solution to Victor.

There is also a solution which does not disclose the solution for x to Victor: when Peggy wants to prove knowledge of the solution of Rubik's cube x, Peggy presents another Rubik's cube y. Victor may then either (1) ask Peggy to solve y, or (2) ask Peggy to show how x can be transformed into y. Victor

[13] This definition, which is cited from [Gol02], is a slight variation of the original definition, which can be found in [GMR85].

may not ask both, but he may choose one of both as he wishes. If Peggy is able to solve every Rubik's cube, she will be always be able to perform both the solution of y, and the transformation of y into x.[14] These two together constitute the solution of x. But that solution will never be disclosed to Victor, as he will only see (1) or (2), but never both. The first time Peggy and Victor run this protocol, Victor might believe that Peggy has had the sheer luck that she knows the half of the solution that Victor asked for. But if they repeat the protocol k times, the chance that Peggy does not know the solution reduces to 2^{-k}, which converges to 0 as k increases.

This second protocol is a *zero-knowledge proof*: it convinces Victor of Peggy's knowledge, without disclosing the knowledge itself. Roughly, a zero-knowledge proof is an interactive proof in which the verifier learns nothing more than the assertion proven. The more formal definition of 'nothing more' states that anything that Victor learns by means of the protocol, can be computed just as efficiently from the assertion proven by the protocol alone. For an elaborate discussion of the definition of 'nothing more', consult [Gol02]. For rather simple explanations and examples of zero-knowledge proofs, consult [QQQ+90, NNR99].

It has been shown that every NP-complete set has a zero-knowledge proof [GMW91], provided that one-way functions exist.[15] This is a very valuable result, because it can be used in a cryptographic protocol to convince other parties that one adheres to the protocol specification without disclosing ones private inputs. Instead of disclosing the private inputs, principals must provide a zero-knowledge proof of the correctness of their secret-based actions. This means that any protocol which is secure in the honest-but-curious adversary model can be transformed into a protocol which is secure in the malicious adversary model [Gol02]. Unfortunately, the computational complexity and communication complexity of such a transformed protocol which is secure in the malicious adversary model may well be very unattractive.

There are some correspondences between secure multiparty computation (see Section 2.7) and zero-knowledge proofs. Zero-knowledge proofs satisfy properties roughly equivalent to the the *privacy* and *validity* properties of a secure multiparty computation. *Privacy* because no information on the private inputs is disclosed, and *validity* because an interactive proof is both *sound* and *complete*.[16] Because the roles of the principals in interactive proof systems are not symmetric, *fairness* is not a relevant property. *Autarky* applies to interactive proof systems in the sense that no third party is involved.

---

[14] Remember that Peggy may choose y. If Peggy is knows the solution to x, she can construct a y in such a way that she knows both how to solve y and how to transform y into x. If Peggy does not know the solution to x, she can guarantee only one of both, giving her a 50% chance of failing to provide a convincing proof on y.

[15] Cryptographic hash functions are a particular type of one-way functions, and are explained in the next chapter.

[16] Again: logicians, take note that sound and complete here are used with the meaning of Definition 2.1 shown on page 25.