CONTENT

1 MATHEMATICS

[3, c8–c9,App B][4, c1–c5]

Cryptography is inherently mathematical in nature, the reader is therefore going to be assumed to be familiar with a number of concepts. A good textbook to cover the basics needed, and more, is that of Galbraith [5].

Before proceeding we will set up some notation: The ring of integers is denoted by $\mathbb{Z}$, whilst the fields of rational, real and complex numbers are denoted by $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$. The ring of integers modulo N will be denoted by $\mathbb{Z}/N\mathbb{Z}$, when N is a prime p this is a finite field often denoted by $\mathbb{F}_p$. The set of invertible elements will be written $(\mathbb{Z}/N\mathbb{Z})^*$ or $\mathbb{F}_p^*$. An RSA modulus N will denote an integer N, which is the product of two (large) prime factors $N = p \cdot q$.

Finite abelian groups of prime order q are also a basic construct. These are either written multiplicatively, in which case an element is written as g^x for some $x \in \mathbb{Z}/q\mathbb{Z}$; when written additively an element can be written as $[x] \cdot P$. The element g (in the multiplicative case) and P (in the additive case) is called the generator.

The standard example of finite abelian groups of prime order used in cryptography are elliptic curves. An elliptic curve over a finite field $\mathbb{F}_p$ is the set of solutions (X, Y) to an equation of the form

$$E : Y^2 = X^3 + A \cdot X + B$$

where A and B are fixed constants. Such a set of solutions, plus a special point at infinity denoted by $\mathcal{O}$, form a finite abelian group denoted by $E(\mathbb{F}_p)$. The group law is a classic law dating back to Newton and Fermat called the chord-tangent process. When A and B are selected carefully one can ensure that the size of $E(\mathbb{F}_p)$ is a prime q. This will be important later in Section 2.3 to ensure the discrete logarithm problem in the elliptic curve is hard.

Some cryptographic schemes make use of lattices which are discrete subgroups of the subgroups of $\mathbb{R}^n$. A lattice can be defined by a generating matrix $B \in \mathbb{R}^{n \cdot m}$, where each column of B forms a basis element. The lattice is then the set of elements of the form $\mathbf{y} = B \cdot \mathbf{x}$ where $\mathbf{x}$ ranges over all elements in $\mathbb{Z}^m$. Since a lattice is discrete it has a well-defined length of the shortest non-zero vector. In Section 2.3 we note that finding this shortest non-zero vector is a hard computational problem.

Sampling a uniformly random element from a set A will be denoted by $x \leftarrow A$. If the set A consists of a single element a we will write this as the assignment $x \leftarrow a$; with the equality symbol $=$ being reserved for equalities as opposed to assignments. If A is a randomized algorithm, then we write $x \leftarrow A(y; r)$ for the assignment to x of the output of running A on input y with random coins r.

2 CRYPTOGRAPHIC SECURITY MODELS

[3, c1–c4][4, c11]

Modern cryptography has adopted a methodology of 'Provable Security' to define and understand the security of cryptographic constructions. The basic design procedure is to define the **syntax** for a cryptographic scheme. This gives the input and output behaviours of the algorithms making up the scheme and defines correctness. Then a **security model** is presented which defines what security goals are expected of the given scheme. Then, given a **specific instantiation** which meets the given syntax, a formal **security proof** for the instantiation is given relative to some known **hard problems**.

The security proof is not an absolute guarantee of security. It is a proof that the given instantiation, when implemented correctly, satisfies the given security model assuming some hard problems are indeed hard. Thus, if an attacker can perform operations which are outside the model, or manages to break the underlying hard problem, then the proof is worthless. However, a security proof, with respect to well studied models and hard problems, can give strong guarantees that the given construction has no fundamental weaknesses.

In the next subsections we shall go into these ideas in more detail, and then give some examples of security statements; further details of the syntax and security definitions can be found in [6, 7]. At a high level the reason for these definitions is that the intuitive notion of a cryptographic construction being secure is not sufficient enough. For example the natural definition for encryption security is that an attacker should be unable to recover the decryption key, or the attacker should be unable to recover a message encrypted under one ciphertext. Whilst these ideas are necessary for any secure scheme they are not sufficient. We need to protect against an attacker aims for find *some* information about an encrypted message, when the attacker is able to mount chosen plaintext and chosen ciphertext attacks on a legitimate user.

2.1 Syntax of Basic Schemes

The syntax of a cryptographic scheme is defined by the algorithms which make up the scheme, as well as a correctness definition. The correctness definition gives what behaviour one can expect when there is no adversarial behaviour. For example, a symmetric encryption scheme is defined by three algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$. The KeyGen algorithm is a probabilistic algorithm which outputs a symmetric key $\mathfrak{k} \leftarrow \mathsf{KeyGen}()$; Enc is a probabilistic algorithm which takes a message $m \in \mathcal{M}$, some randomness $r \in \mathcal{R}$ and a key and returns a $c \leftarrow \mathsf{Enc}(m, \mathfrak{k}; r) \in \mathcal{C}$; whilst Dec is (usually) a deterministic algorithm which takes a ciphertext and a key and returns the underlying plaintext. The correctness definition is:

$$\forall \mathfrak{k} \leftarrow \mathsf{KeyGen}(), r \leftarrow \mathcal{R}, m \leftarrow \mathcal{M}, \quad \mathsf{Dec}(\mathsf{Enc}(m, \mathfrak{k}; r), \mathfrak{k}) = m.$$

For public key encryption schemes the definitions are similar, but now $\mathsf{KeyGen}()$ outputs key pairs and the correctness definition becomes:

$$\forall (\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}(), r \leftarrow \mathcal{R}, m \leftarrow \mathcal{M}, \quad \mathsf{Dec}(\mathsf{Enc}(m, \mathfrak{pk}; r), \mathfrak{sk}) = m.$$

The equivalent constructions for authentication mechanisms are Message Authentication Codes (or MACs) in the symmetric key setting, and digital signatures schemes in the public key setting. A MAC scheme is given by a triple of algorithms $(\mathsf{KeyGen}, \mathsf{MAC}, \mathsf{Verify})$, where the MAC function outputs a tag given a message and a key (and possibly some random coins), and the Verify function checks the message, tag and key are consistent. A signature scheme is given by a similar triple $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, where now the tag produced is called a 'signature'. Thus the correctness definitions for these constructions are as follows

$$\mathfrak{k} \leftarrow \mathsf{KeyGen}(), r \leftarrow \mathcal{R}, m \leftarrow \mathcal{M}, \quad \mathsf{Verify}(m, \mathsf{MAC}(m, \mathfrak{k}; r), \mathfrak{k}) = \mathsf{true}.$$

and

$$(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}(), r \leftarrow \mathcal{R}, m \leftarrow \mathcal{M}, \quad \mathsf{Verify}(m, \mathsf{Sign}(m, \mathfrak{sk}; r), \mathfrak{pk}) = \mathsf{true}.$$

Note, that for deterministic MACs the verification algorithm is usually just to recompute the MAC tag $\mathsf{MAC}(m, \mathfrak{k})$, and then check it was what was received.

2.2 Basic Security Definitions

A security definition is usually given in the context of an attacker's **security goal**, followed by their **capabilities**. So, for example, a naive security goal for encryption could be to recover the underlying plaintext, so-called One-Way (or OW) security. This process of an attacker trying to obtain a specific goal is called a *security game*, with the attacker winning the *game*, if they can break this security goal with greater probability than random guessing. This advantage in probability over random guessing is called the adversary's *advantage*. The capabilities are expressed in terms of what oracles, or functions, we give the adversary access to. So, for example, in a naive security game for encryption we may give the adversary no oracles at all, producing a so-called Passive Attack (or PASS) capability.

The attacker is modelled as an arbitrary algorithm, or Turing machine, A, and if we give the adversary access to oracles then we write these as subscripts $A^{\mathcal{O}}$. In our naive security game (called OW-PASS) the adversary has no oracles and its goal is simply to recover the message underlying a given ciphertext. The precise definition is given in Figure 1, where $\mathsf{Adv}^{\mathsf{OW-PASS}}(A, t)$ denote the advantage over a random guess that a given adversary has after running for time t. We say that a given construction is secure in the given model (which our naive example would be named OW-PASS), if the above advantage is *negligible* for all probabilistic polynomial time adversaries A. Here, negligible and polynomial time are measured in terms of a *security parameter* (which one can think of as the key size). Note, for OW-PASS this assumes that the message space is not bigger than the space of all possible keys. Also note, that this is an asymptotic definition, which in the context of schemes with fixed key size, makes no sense. In such situations we require that (t/Adv) is greater than some given concrete bound such as 2^{128}, since it is believed that performing an algorithm requiring 2^{128} steps is infeasible even for a nation-state adversary.

In the context of encryption (both symmetric and public key) the above naive security goal is not seen as being suitable for real applications. Instead, the security goal of Indistinguishable encryptions (or IND) is usually used. This asks the adversary to first come up with two plaintexts, of equal length, and then the challenger (or environment) encrypts one of them and gives the resulting challenge to the adversary. The adversary's goal is then to determine which plaintext was encrypted. In the context of a passive attack this gives an advantage statement as given in the second part of Figure 1, where the two stages of the adversary are given by A_1 and A_2.

In terms of encryption, the above passive attack is almost always not sufficient in terms of capturing real-world adversarial capabilities, since real systems almost always give the attacker additional attack vectors. Thus two other (increasingly strong) attack capabilities are usually considered. These are a Chosen Plaintext Attack (or CPA capability), in which the adversary is given access to an encryption oracle to encrypt arbitrary messages of his choice, and a Chosen Ciphertext Attack (or CCA capability), in which the adversary has both an encryption and decryption oracle. In the case of a public key scheme the adversary always has access to an encryption oracle because it can encrypt plaintexts for itself using the public key, so in this case PASS and CPA are equivalent. In the case of a CCA capability we restrict the decryption oracle so that the adversary may not ask of it the challenge ciphertext c^*; otherwise it can trivially win the security game. Thus the advantage of an IND-CCA adversary against a public key encryption scheme would be defined as the third definition in Figure 1. Other security definitions are possible for encryption (such as Real-or-Random) but the above are the main ones.

OW-PASS Definition:

$$\mathsf{Adv}^{\mathsf{OW-PASS}}(A,t) = \Pr\Big[\ \mathfrak{k} \leftarrow \mathsf{KeyGen}(),\ m^* \leftarrow \mathcal{M},\ r \leftarrow \mathcal{R},\ c^* \leftarrow \mathsf{Enc}(m,\mathfrak{k};r),\ m \leftarrow A(c^*):\ m = m^*\Big] - \frac{1}{|\mathcal{M}|}.$$

One reads the probability statement as the being the probability that $m = m^*$, given that m and m^* are produced by first sampling $\mathfrak{k}$ from algorithm $\mathsf{KeyGen}()$, then sampling m^* and r from the spaces $\mathcal{M}$ and $\mathcal{R}$ at random, then determining c^* by calling $\mathsf{Enc}(m,\mathfrak{k};r)$ and finally passing c^* to the Adversary A, and getting m in return.

IND-PASS Symmetric Key Encryption Definition:

$$\mathsf{Adv}^{\mathsf{IND-PASS}}(A,t) = \Pr\Big[\ \mathfrak{k} \leftarrow \mathsf{KeyGen}(),\ b \leftarrow \{0,1\},\ m_0,\ m_1,\ \mathsf{state} \leftarrow A_1(),\ r \leftarrow \mathcal{R},\ c^* \leftarrow \mathsf{Enc}(m_b,\mathfrak{k};r),\ b' \leftarrow A_2(c^*,\mathsf{state}):\ b = b'\Big] - \frac{1}{2}.$$

IND-CCA Public Key Encryption Definition:

$$\mathsf{Adv}^{\mathsf{IND-CCA}}(A,t) = \Pr\Big[\ (\mathfrak{pk},\mathfrak{sk}) \leftarrow \mathsf{KeyGen}(),\ b \leftarrow \{0,1\},\ m_0,\ m_1,\mathsf{state} \leftarrow A_1^{\mathsf{Dec}(\cdot,\mathfrak{sk})}(\mathfrak{pk}),\ r \leftarrow \mathcal{R},\ c^* \leftarrow \mathsf{Enc}(m_b,\mathfrak{pk};r),\ b' \leftarrow A_2^{\mathsf{Dec}(\cdot,\mathfrak{sk})}(c^*,\mathsf{state}):\ b = b'\Big] - \frac{1}{2}.$$

UF-CMA Signature Security Definition:

$$\mathsf{Adv}^{\mathsf{UF-CMA}}(A,t) = \Pr\Big[\ (\mathfrak{pk},\mathfrak{sk}) \leftarrow \mathsf{KeyGen}(),\ (m,\sigma) \leftarrow A^{\mathsf{Sign}(\cdot,\mathfrak{sk})}(\mathfrak{pk}):\ \mathsf{Verify}(m,\sigma,\mathfrak{pk}) = \mathsf{true}\Big].$$

IND-CCA KEM Security Definition:

$$\mathsf{Adv}^{\mathsf{IND-CCA}}(A,t) = \Pr\Big[\ (\mathfrak{pk},\mathfrak{sk}) \leftarrow \mathsf{KEMKeyGen}(),\ b \leftarrow \{0,1\},\ \mathfrak{k}_0 \leftarrow \mathcal{K},\ r \leftarrow \mathcal{R},\ \mathfrak{k}_1, c^* \leftarrow \mathsf{KEMEnc}(\mathfrak{pk};r),\ b' \leftarrow A_2^{\mathsf{KEMDec}(\cdot,\mathfrak{sk})}(c^*,\mathfrak{k}_b):\ b = b'\Big] - \tfrac{1}{2}.$$

Figure 1: Technical Security Definitions

For MACs (resp. digital signature schemes) the standard security goal is to come up with a message/tag (resp. message/signature) pair which passes the verification algorithm, a so-called Universal Forgery (or UF) attack. We make no assumption about whether the message has any meaning, indeed, the attacker wins if he is able to create a signature on *any* bit-string. If the adversary is given no oracles then he is said to be mounting a passive attack, whilst if the adversary is given a tag generation (resp. signing oracle) he is said to be executing a Chosen Message Attack (CMA). In the latter case the final forgery must not be one of the outputs of the given oracle. In the case of MAC security, one may also give the adversary access to a tag verification oracle. However, for deterministic MACs this is implied by the CMA capability and is hence usually dropped, since verification only involves re-computing the MAC.

Again we define an advantage and require this to be negligible in the security parameter. For digital signatures the advantage for the UF-CMA game is given by the fourth equation in Figure 1.

2.3 Hard Problems

As explained above, security proofs are always relative to some hard problems. These hard problems are often called *cryptographic primitives*, since they are the smallest atomic object from which cryptographic schemes and protocols can be built. Such cryptographic primitives come in two flavours: Either they are keyed complexity theoretic definitions of functions, or they are mathematical hard problems.

In the former case one could consider a function $F_k(\cdot) : D \longrightarrow C$ selected from a function family $\{F_k\}$ and indexed by some index k (thought of as a key of varying length). One can then ask whether the function selected is indistinguishable (by a probabilistic polynomial time algorithm A which has oracle access to the function) from a uniform random function from D to C. If such an assumption holds, then we say the function family defines a (keyed) Pseudo-Random Function (PRF). In the case when the domain D is equal to the co-domain C we can ask whether the function is indistinguishable from a randomly chosen permutation, in which case we say the family defines a (keyed) Pseudo-Random Permutation (PRP).

In the case of a block cipher, such as AES (see later), where one has $C = D = \{0,1\}^{128}$, it is a basic assumption that the AES function family (indexed by the key k) is a Pseudo-Random Permutation.

In the case of mathematical hard problems we have a similar formulation, but the definitions are often more intuitive. For example, one can ask the question whether a given RSA modulus $N = p \cdot q$ can be factored into its prime components p and q, the so-called factoring problem. The RSA group $\mathbb{Z}/N\mathbb{Z}$ defines a finite abelian group of unknown order (the order is known to the person who created N), finding the order of this group is equivalent to factoring N. The RSA function $x \longrightarrow x^e \pmod{N}$ is believed to be hard to invert, leading to the so-called RSA-inversion problem of, given $y \in (\mathbb{Z}/N\mathbb{Z})^*$, finding x such that $x^e = y \pmod{N}$. It is known that the function can easily be inverted if the modulus N can be factored, but it is unknown if inverting the function implies N can be factored. Thus we have a situation where one problem (factoring) seems to be harder to solve than another problem (the RSA problem). However, in practice, we assume that both problems are hard, given appropriately chosen parameters. Details on the best method to factor large numbers, the so-called Number Field Sieve, can be found in [8].

In finite abelian groups of known order (usually assumed to be prime), one can define other problems. The problem of inverting the function $x \longrightarrow g^x$, is known as the Discrete Logarithm Problem (DLP). The problem of, given g^x and g^y, determining $g^{x \cdot y}$ is known as the Diffie–Hellman Problem (DHP). The problem of distinguishing between triples of the form (g^x, g^y, g^z) and $(g^x, g^y, g^{x \cdot y})$ for random x, y, z is known as the Decision Diffie–Hellman (DDH) problem. When written additively in an elliptic curve group, a DDH triple has the form $([x] \cdot P, [y] \cdot P, [z] \cdot P)$.

Generally speaking, the mathematical hard problems are used to establish the security of public key primitives. A major issue is that the above problems (Factoring, RSA-problem, DLP, DHP, DDH), on which we base all of our main existing public key algorithms, are easily solved by large-scale quantum computers. This has led designers to try to build cryptographic schemes on top of mathematical primitives which do not appear to be able to be broken by a quantum computer. Examples of such problems are the problem of determining the shortest vector in a high dimensional lattice, the so-called Shortest Vector Problem (SVP), and the problem of determining the closest lattice vector to a non-lattice vector, the so-called Closest Vector Problem (CVP). The best algorithms to solve these hard problems are lattice reduction algorithms, a nice survey of these algorithms and applications can be found in [9]. The SVP and CVP problems, and others, give rise to a whole new area called Post-Quantum Cryptography (PQC).

Example: Putting the above ideas together, one may encounter statements such as: *The public key encryption XYZ is IND-CCA secure assuming the RSA-problem is hard and AES is a PRP.* This statement tells us that *any* attack against the XYZ scheme must either be against some weakness in the implementation, or must come from some attack not captured in the IND-CCA model, or must come from solving the RSA-problem, or must come from showing that AES is not a PRP.

2.4 Setup Assumptions

Some cryptographic protocols require some setup assumptions. These are assumptions about the environment, or some data, which need to be satisfied before the protocol can be considered secure. These assumptions come in a variety of flavours. For example, one common setup assumption is that there exists a so-called Public-Key Infrastructure (PKI), meaning that we have a trusted binding between entities' public keys and their identities.

Another setup assumption is the existence of a string (called the Common Reference String or CRS) available to all parties, and which has been set up in a trusted manner, i.e. such that no party has control of this string.

Other setup assumptions could be physical, for example, that the algorithms have access to good sources of random numbers, or that their internal workings are not susceptible to an invasive attacker, i.e. they are immune to side-channel attacks.

2.5 Simulation and UC Security

The above definitions of security make extensive use of the notion of indistinguishability between two executions. Indeed, many of the proof techniques used in the security proofs construct *simulations* of cryptographic operations. A simulation is an execution which is indistinguishable from the real execution, but does not involve (typically) the use of any key material. Another method to produce security models is the so-called *simulation paradigm*, where we ask that an adversary cannot tell the simulation from a real execution (unless they can solve some hard problem). This paradigm is often used to establish security results for more complex cryptographic protocols.

A problem with both the game/advantage-based definitions defined earlier and the simulation definitions is that they only apply to stand-alone executions, i.e. executions of one instance of the protocol in one environment. To cope with arbitrarily complex executions and composition of cryptographic protocols an extension to the simulation paradigm exists called the Universal Composability (UC) framework.

3 INFORMATION-THEORETICALLY SECURE CONSTRUCTIONS

[3, c2][4, c19]

Whilst much of cryptography is focused on securing against adversaries that are modelled as probabilistic polynomial time Turing machines, some constructions are known to provide security against unbounded adversaries. These are called information-theoretically secure constructions. A nice introduction to the information theoretic side of cryptography can be found in [10].

3.1 One-Time Pad

The most famous primitive which provides information-theoretic security is the one-time pad. Here, a binary message $m \in \{0,1\}^t$ is encrypted by taking a key $k \in \{0,1\}^t$ uniformly at random, and then producing the ciphertext $c = m \oplus k$. In terms of our earlier security models, this is an IND-PASS scheme even in the presence of a computationally unbounded adversary. However, the fact that it does not provide IND-CPA security is obvious, as the encryption scheme is determinisitic. The scheme is unsuitable in almost all modern environments as one requires a key as long as the message and the key may only be used once; hence the name one-time pad.

3.2 Secret Sharing

Secret sharing schemes allow a secret to be shared among a set of parties so that only a given subset can reconstruct the secret by bringing their shares together. The person who constructs the sharing of the secret is called the dealer. The set of parties who can reconstruct the secret are called qualified sets, with the set of all qualified sets being called an *access structure*.

Any set which is not qualified is said to be an unqualified set, and the set of all unqualified sets is called an *adversary structure*. The access structure is usually assumed to be monotone, in that if the parties in A can reconstruct the secret, then so can any super-set of A.

Many secret sharing schemes provided information-theoretic security, in that any set of parties which is unqualified can obtain no information about the shared secret even if they have unbounded computing power.

A special form of access structure is a so-called *threshold* structure. Here we allow any subset of $t + 1$ parties to reconstruct the secret, whereas any subset of t parties is unable to learn anything. The value t is being called the threshold. One example construction of a threshold secret sharing scheme for a secret s in a field $\mathbb{F}_p$, with $n > p$ is via Shamir's secret sharing scheme.

In Shamir secret sharing, one selects a polynomial $f(X) \in \mathbb{F}_p[X]$ of degree t with constant coefficients s, the value one wishes to share. The share values are then given by $s_i = f(i) \pmod{p}$, for $i = 1, \ldots, n$, with party i being given s_i. Reconstruction of the value s from a subset of more than t values s_i can be done using Lagrange interpolation.

Due to an equivalence with Reed-Solomon error correcting codes, if $t < n/2$, then on receipt of n share values s_i, a reconstructing party can detect if any party has given it an invalid share. Additionally, if $t < n/3$ then the reconstructing party can correct for any invalid shares.

Replicated secret sharing is a second popular scheme which supports any monotone access structure. Given a boolean formula defining who should have access to the secret, one can define a secret sharing scheme from this formula by replacing all occurrences of AND with $+$ and all occurrences of OR with a new secret. For example, given the formulae

$$(P_1 \text{ AND } P_2) \text{ OR } (P_2 \text{ AND } P_3),$$

one can share a secret s by writing it as $s = s_1 + s_2 = s_2' + s_3$ and then, giving party P_1 the value s_1, party P_2 the pair of values s_2 and s_2', and party P_3 the value s_3. Replicated secret sharing is the scheme obtained in this way when putting the boolean formulae into Conjunctive Normal Form.

Of importance in applications of secret sharing, especially to Secure Multi-Party Computation (see Section 9.4) is whether the adversary structure is Q_2 or Q_3. An adversary structure is said to be Q_i if no set of i unqualified sets have union the full set of players. Shamir's secret sharing scheme is Q_2 if $t < n/2$ and Q_3 when $t < n/3$. The error detection (resp. correction) properties of Shamir's secret sharing scheme mentioned above follow through to any Q_2 (resp. Q_3) adversary structure.

4 SYMMETRIC PRIMITIVES

[3, c3–c6][4, c11–c14]

Symmetric primitives are a key component of many cryptographic constructions. There are three such basic primitives: block ciphers, stream ciphers, and hash functions. Theoretically, all are keyed functions, i.e. they take as input a secret key, whilst in practice one often considers hash functions which are unkeyed. At a basic level, all are functions $f : \mathcal{K} \times D \longrightarrow C$ where $\mathcal{K}$ is the key space, D is the domain (which is of a fixed finite size for block ciphers and stream ciphers).

As explained in the introduction we will not be discussing in this report cryptanalysis of symmetric primitives, we will only be examining secure constructions. However, the main two techniques for attacks in this space are so-called differential and linear cryptanalysis. The interested reader is referred to the excellent tutorial by Howard Heys [11] on these topics, or the book [12].

4.1 Block Ciphers

A block cipher is a function $f : \mathcal{K} \times \{0,1\}^b \longrightarrow \{0,1\}^b$, where b is the block size. Despite their names such functions should not be thought of as an encryption algorithm. It is, however, a building block in many encryption algorithms. The design of block ciphers is a deep area of subject in cryptography, analogous to the design of number theoretic one-way functions. Much like number-theoretic one-way functions, cryptographic constructions are proved secure relative to an associated hard problem which a given block cipher is assumed to satisfy.

For a fixed key, a block cipher is assumed to act as a permutation on the set $\{0,1\}^b$, i.e. for a fixed key k, the map $f_k : \{0,1\}^b \longrightarrow \{0,1\}^b$ is a bijection. It is also assumed that inverting this permutation is also easy (if the key is known). A block cipher is considered *secure* if no polynomial time adversary, given oracle access to a permutation on $\{0,1\}^b$, can tell the difference between being given a uniformly random permutation or the function f_k for some fixed hidden key k, i.e. the block cipher is a PRP. In some applications, we only require that the block cipher is a function, i.e. not a bijection. In which case we require the block cipher is a PRF.

One can never prove that a block cipher is a PRP, so the design criteria is usually a task of building a mathematical construction which resists all known attacks. The main such attacks which one resists are so-called *linear cryptanalysis*, where one approximates non-linear components within the block cipher by linear functions, and *differential cryptanalysis*, where one looks at how two outputs vary on related input messages, e.g. one applies f_k to various inputs m_0 and m_1 where $m_0 \oplus m_1 = \Delta$ a fixed value.

The design of a block cipher is made up of a number of simpler components. There are usually layers of simple fixed permutations, and layers of table lookups. These table lookups are called S-boxes, where the S stands for substitutions. There are two main techniques to design block ciphers. Both repeat a simple operation (called a round) a number of times. Each round consists of a combination of permutations and substitutions, and a key addition. The main key is first expanded into round-keys, with each round having a different round-key.

In the first methodology, called a *Feistel Network*, the S-Boxes allowed in each round can be non-injective, i.e. non-invertible. Despite this, the Feistel constructions still maintain the overall invertibility of the block cipher construction. The second method is a *Substitution-Permutation Network* design in which each round consists of a round-key addition, followed by a fixed permutation, followed by the application of bijective S-boxes. In general, the Feistel construction requires more rounds than the Substitution-Permutation network construction.

The DES (Data Encryption Standard) block cipher (with an original key of 56-bits and block size of $b = 64$) is a Feistel construction. The DES algorithm dates from the 1970s, and the key size is now considered far too small for any application. However, one can extend DES into a 112- or 168-bit key block cipher to construct an algorithm called 2DES or 3DES. The use of 2DES or 3DES is still considered secure, although in some applications, the block size of 64-bits is considered insecure for real-world use.

The AES (Advanced Encryption Standard) block cipher is the modern replacement for DES, and it is a block cipher with a 128-, 192- or 256-bit key, and with a block size of $b = 128$ bits. The AES algorithm has hardware support on many microprocessors, making operations using AES much faster than using other cryptographic primitives. Readers who wish to understand more about the design of the AES block cipher referred to [13].

4.2 Stream Ciphers

A stream cipher is one which produces an arbitrary length string of output bits, i.e. the co-domain of the function is essentially unbounded. Stream ciphers can be constructed from block ciphers, by using a block cipher in Counter Mode (see Section 5.1). However, the stream cipher is usually reserved for constructions which are special-purpose and for which the hardware complexity is much reduced.

Clearly, a stream cipher cannot be a permutation, but we require that no polynomial time adversary can distinguish oracle access to the stream cipher from oracle access to a uniformly random function with infinite co-domain. The design of stream ciphers is more ad-hoc than that of the design of block ciphers. In addition, there is less widespread adoption outside specific application areas. The interested reader is referred to the outcome of the eStream competition for details of specific ad-hoc stream cipher designs [14].

4.3 Hash Functions

Hash functions are much like block ciphers in that they should act as PRFs. However, the input domain can be unbounded. Since a PRF needs to be keyed to make any sense in theoretical tracts, a hash function is usually a keyed object. In practice, we often require an unkeyed object, in which case one considers the actual hash function used to have an implicit inbuilt fixed key, and have been chosen from a function family already.

When considering a fixed hash function, one is usually interested in the intractability of inverting the hash function (the one-way property), the intractability of finding two inputs with the same output (the collision resistance property), or the intractability of finding, given an input/output pair, a new input which gives the same output (the second-preimage resistance property).

4.3.1 Merkle-Damgård Construction

Early hash functions were based on the Merkle-Damgård construction. The family of such functions (MD4, MD5, SHA-1, SHA-2) have a number of issues, with only SHA-2 now being considered secure. The Merkle-Damgård construction takes a compression function $f(x, y)$ taking two inputs x, y of fixed length with the output length of x. This is used to derive a function which allows arbitrary length inputs by first dividing a message m into t blocks $m_1, \ldots, m_t$ each of length $|y|$, and then applying

$$h_i = f(h_{i-1}, m_i) \text{ for } i = 1, \ldots, t,$$

where the output is h_t and h_0 is some initial value, which can be thought of as a fixed key for the hash function.

The above methodology requires a method to pad the initial input block to encode the length, and it suffers from a number of practical issues. For example, there are obvious length extension attacks (namely a hash on a message m can be extended to a hash on $m\|m'$ without knowing the whole of m) which render the use of such hash functions problematic in some applications. For example, in HMAC (see Section 5.2), one requires two applications of the hash function to prevent such attacks.

4.3.2 Sponge Constructions

A more modern approach to hash function design is to create a so-called sponge construction. This is the design philosophy behind SHA-3 (a.k.a.). A sponge is a function which operates in two phases. In the first phase, one enters data into the sponge state and, in the second phase, one squeezes data out from the sponge.

The sponge state is a pair $(r, c) \in \{0, 1\}^{|r|+|c|}$, where the length of r denotes the input/output rate and c is a variable holding an internal state hidden from any attacker. The size of c is directly related to the security of the construction. At each round, a public permutation p is applied to the state (r, c).

In the input phase of the sponge, the input data m, after suitable padding, is divided into t blocks $m_1, \ldots, m_t$ of size $|r|$. Then the state is updated via the mapping

$$(r_i, c_i) = p(r_{i-1} \oplus m_i, c_{i-1}) \text{ for } i = 1, \ldots, t,$$

where r_0 and c_0 are initialized to be fixed all-zero bit strings. After data is entered into the sponge, one can obtain s blocks of $|r|$-bit outputs $o_1, \ldots, o_s$ by computing

$$(o_i, c'_i) = p(o_{i-1}, c'_{i-1}) \text{ for } i = 1, \ldots, s,$$

where $o_0 = r_t$ and $c'_0 = c_t$. Thus the whole function is given by

$$H(m_1, \ldots, m_t) = o_1, \ldots, o_s.$$

Further details on sponge constructions, and the further objects one can construct from them, and the SHA-3 design in particular can be found at the Keccak web page [15].

4.3.3 Random Oracle Model

Many cryptographic constructions are only secure if one assumes that the hash function used in the construction behaves 'like a random oracle'. Such constructions are believed to be secure in the real world, but theoretically, they are less pleasing. One can think of a proof of security in the random oracle model as a proof in which we allow the attacker to have their usual powers; however, when they (or any of the partners they are attacking) call the underlying hash function the call is made to an external party via an oracle call. This external party then simply plays back a random value, i.e. it does not use any algorithm to generate the random values. All that is required is that if the input is given to the oracle twice, then the same output is always returned.

This clearly does not capture attacks in which the adversary makes clever use of exactly how the hash function is defined etc, and how this definition interacts with other aspects of the scheme/protocol under analysis. However, this modelling methodology has proved remarkably good in enabling cryptographers to design schemes which are secure in the real world.

5 SYMMETRIC ENCRYPTION AND AUTHENTICATION

[3, c3–c4][4, c13–c14]

A block cipher, such as AES or DES, does not provide an effective form of data encryption or data/entity authentication on its own. To provide such symmetric cryptographic constructions, one needs a scheme, which takes the primitive and then utilizes this in a more complex construction to provide the required cryptographic service. In the context of symmetric encryption, these are provided by modes of operation. In the case of authentication, it is provided by a MAC construction. Additionally, block ciphers are often used to take some entropy and then expand, or collapse, this into a pseudo-random stream or key; a so-called XOF (or Extendable Output Function) or KDF (or Key Derivation Function). Further details on block cipher based constructions can be found at [16], whereas further details on Sponger/Keccak based constructions can be found at [15].

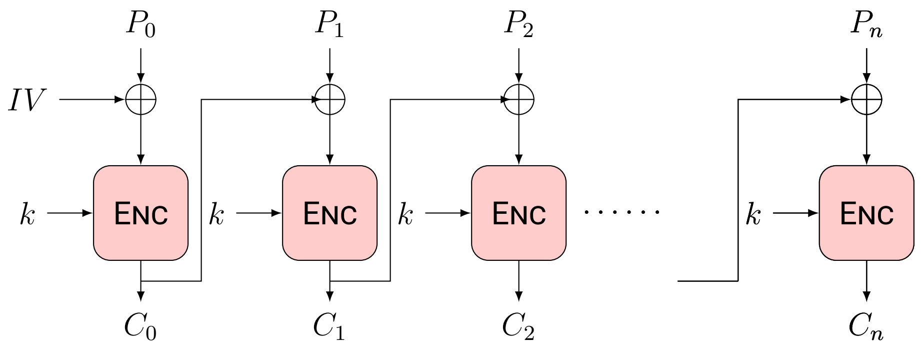

Figure 2: CBC Mode Encryption (All Figures are produced using *TikZ for Cryptographers* ***https://www.iacr.org/authors/tikz/***).

5.1 Modes of Operation

Historically, there have been four traditional modes of operation to turn a block cipher into an encryption algorithm. These were ECB, CBC, OFB and CFB modes. In recent years, the CTR mode has also been added to this list. Among these, only CBC mode (given in Figure 2) and CTR mode (given in Figure 3) are used widely within current systems. In these Figures, the block cipher is represented by the function Enc

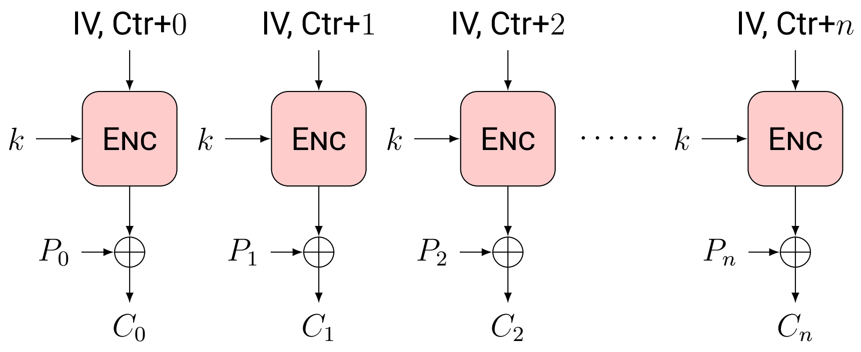

Figure 3: CTR Mode Encryption

On their own, however, CBC and CTR modes only provide IND-CPA security. This is far weaker than the 'gold standard' of security, namely IND-CCA (discussed earlier). Thus, modern systems use modes which provide this level of security, also enabling additional data (such as session identifiers) to be tagged into the encryption algorithm. Such algorithms are called AEAD methods (or Authenticated Encryption with Associated Data). In such algorithms, the encryption primitive takes as input a message to be encrypted, plus some associated data. To decrypt, the ciphertext is given, along with the associated data. Decryption will only work if both the key is correct and the associated data is what was input during the encryption process.

The simplest method to obtain an AEAD algorithm is to take an IND-CPA mode of operation such as CBC or CTR, and then to apply a MAC to the ciphertext and the data to be authenticated, giving us the so-called Encrypt-then-MAC paradigm. Thus, to encrypt m with authenticated data a, one applies the transform

$$c_1 \leftarrow \mathsf{Enc}(m, \mathfrak{k}_1; r), \qquad c_2 \leftarrow \mathsf{MAC}(c_1 \| a, \mathfrak{k}_2; r),$$

with the ciphertext being (c_1, c_2). In such a construction, it is important that the MAC is applied to the ciphertext as opposed to the message.

A major issue with the Encrypt-then-MAC construction is that one needs to pass the data to the underlying block cipher twice, with two different keys. Thus, new constructions of AEAD schemes have been given which are more efficient. The most widely deployed of these is GCM (or Galois Counter Mode), see Figure 4, which is widely deployed due to the support for this in modern processors.

One time AEAD constructions, otherwise known as DEMs, can be obtained by simply making the randomized AEAD deterministic by fixing the IV to zero.

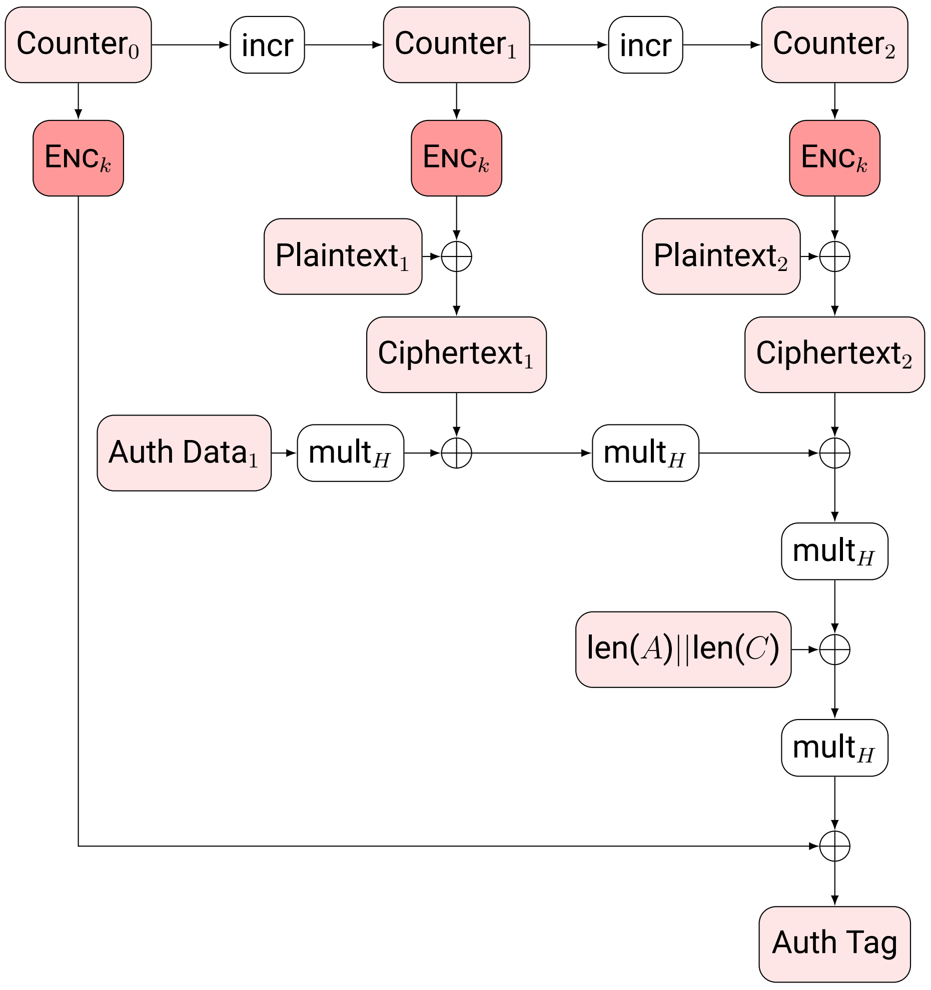

Figure 4: GCM Mode Encryption

5.2 Message Authentication Codes

Message authentication codes can be produced in roughly the same manner as modes of operation. In particular, the standard MAC function is to utilize CBC mode with a zero-IV, and then to output the final ciphertext block as the MAC tag, thus producing a deterministic MAC function. On its own, even with suitable padding of the message, this is only secure when used with fixed length messages. Thus, often a form of post-processing of the MAC output tag is performed. For example, the final CBC ciphertext block is then passed through another application of the underlying block cipher, but using a different key.

The GCM AEAD method of the previous section can be thought of as an Encrypt-then-MAC construction, with the IND-CPA encryption being CTR mode, and the MAC function being a function called GMAC. Although this is rarely used on its own as a MAC function.

Hash functions can also be used to construct MAC functions. The most famous of these is HMAC which is a construction designed for use with Merkle–Damgård-based hash functions. Since Merkle–Damgård-based hash functions suffer from length extension attacks, the HMAC function requires two applications of the underlying hash function. The construction produces a deterministic MAC function given by

$$\text{HMAC}(m, \mathfrak{k}) = H\left((\mathfrak{k} \oplus \text{opad}) \| H((\mathfrak{k} \oplus \text{ipad}) \| m)\right),$$

where opad is the string 0x5c5c...5c5c and ipad is the string 0x3636...3636.

As HMAC is designed specifically for use with Merkle–Damgård-based hash functions, it makes no-sense to use this construction when using a sponge based hash function such as SHA-3. The standardized MAC function derived from SHA-3 is called KMAC (or Keccak MAC). In this function, the sponge construction is used to input a suitably padded message, then the required MAC output is taken as the squeezed output of the sponge; whereas as many bits as squeezed are as needed for the MAC output.

5.3 Key Derivation and Extendable Output Functions

The security definition of a deterministic MAC is essentially equivalent to the definition that the output of the MAC function is indistinguishable from a random string, if one does not know the underlying secret key. As such, MAC functions can be used for other cryptographic operations. For example, in many situations, one must derive a long (or short) string of random bits, given some random input bits. Such functions are called KDFs or XOFs (for Key Derivation Function and Extendable Output Function). Usually, one uses the term KDF when the output is of a fixed length, and XOF when the output could be of an arbitrary length. But the constructions are, usually, essentially the same in both cases.

Such functions can take an arbitrary length input string, and produce another arbitrary length output string which is pseudo-random. There are three main constructions for such functions; one based on block ciphers, one on the Merkle–Damgård hash functions, and one based on sponge-based hash functions.

The constructions based on a block cipher are, at their heart, using CBC-MAC, with a zero key to compress the input string into a cryptographic key and then use the CTR mode of operation under this key to produce the output string. Hence, the construction is essentially given by

$$\mathfrak{k} \leftarrow \text{CBC-MAC}(m, \mathbf{0}), \qquad o_1 \leftarrow \mathsf{Enc}(\mathbf{1}, \mathfrak{k}), \qquad o_2 \leftarrow \mathsf{Enc}(\mathbf{2}, \mathfrak{k}), \quad \ldots$$

where Enc is the underlying block cipher.

The constructions based on the Merkle–Damgård hash function use a similar structure, but using one hash function application per output block, in a method similar to the following

$$o_1 \leftarrow H(m\|1), \qquad o_2 \leftarrow H(m\|2), \quad \ldots$$

Due to the way Merkle–Damgård hash functions are constructed, the above construction (for large enough m) can be done more efficiently than simply applying H as many times as the number of output blocks will dictate.

As one can imagine, the functions based on Keccak are simpler—one simply inputs the suitably padded message into the sponge and then squeezes as many output bits out as required.

Special KDFs can also be defined which take as input a low entropy input, such as a password or PIN, and produce a key for use in a symmetric algorithm. These *password based key derivation functions* are designed to be computationally expensive, so as to mitigate problems associated to brute force attacking of the underlying low entropy input.

5.4 Merkle-Trees and Blockchains

An application of cryptographic hash functions which has recently come to prominance is that of using Merkle-Trees and by extension blockchains. A Merkle-Tree, or hash-tree, is a tree in which each leaf node contains data, and each internal node is the hash of its child nodes. The root node is then publicly published. Merkle-Trees enable efficient demonstration that a leaf node is contained in the tree, in that one simply presents the path of hashes from the leaf up to the root node. Thus verification is logarithmic in the number of leaf nodes. Merkle-Trees can verify any form of stored data and have been used in various protocols such as version control systems, such as Git, and backup systems.

A blockchain is a similar structure, but now the data items are aligned in a chain, and each node hashes both the data item and a link to the previous item in the chain. Blockchains are used in cryptocurrencies such as *Bitcoin*, but they have wider application. The key property a blockchain provides is that (assuming the current head of the chain is authenticated and trusted) the data provides an open distributed ledger in which previous data items are immutable, and the ordering of data items is preserved.

6 PUBLIC KEY ENCRYPTION

[3, c11][4, c15–c17]

As explained above, public key encryption involves two keys, a public one $\mathfrak{pk}$ and a private one $\mathfrak{sk}$. The encryption algorithm uses the public key, whilst the decryption algorithm uses the secret key. Much of public key cryptography is based on number theoretic constructions, thus [5] provides a good coverage of much in this section. The standard security requirement for public key encryption is that the scheme should be IND-CCA. Note that since the encryption key is public we have that IND-PASS is the same as IND-CPA for a public key encryption scheme.

6.1 KEM-DEM Philosophy

In general, public key encryption schemes are orders of magnitude less efficient than symmetric key encryption schemes. Thus, the usual method in utilizing a public key scheme, when large messages need to be encrypted, is via a hybrid method. This hybrid methodology is called the KEM-DEM philosophy A KEM, which stands for Key Encapsulation Mechanism, a public key method to transmit a short key, selected at random from a set $\mathcal{K}$, to a designated recipient. Whereas, a DEM, or Data Encryption Mechanism, is essentially the same as an IND-CCA symmetric encryption scheme, which has key space $\mathcal{K}$. Since a DEM is only ever used once with the same key, we can actually use a weaker notion of IND-CCA encryption for the DEM, in which the adversary is not given access to an encryption oracle; which means the DEM can be *deterministic*.

For a KEM, we call the encryption and decryption mechanisms encapsulation and decapsulation, respectively. It is usual for the syntax of the encapsulation algorithm to not take any input, bar the randomness, and then to return both the ciphertext and the key which it encapsulates. Thus, the syntax, and correctness, of a KEM becomes

$$(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KEMKeyGen}(), r \leftarrow \mathcal{R}, (\mathfrak{k}, c) \leftarrow \mathsf{KEMEnc}(\mathfrak{pk}; r), \quad \mathsf{KEMDec}(c, \mathfrak{sk}) = \mathfrak{k}.$$

The security definition for a KEM is described in the last equation of Figure 1. To construct the hybrid public key encryption scheme we define $\mathsf{KeyGen}()$ to be equal to $\mathsf{KEMKeyGen}$, then $\mathsf{Enc}(m, \mathfrak{pk}; r)$ outputs (c_0, c_1) where

$$\mathfrak{k}, c_0 \leftarrow \mathsf{KEMEnc}(\mathfrak{pk}; r), \quad c_1 \leftarrow \mathsf{DEM}(m, \mathfrak{k}),$$

with $\mathsf{Dec}((c_0, c_1), \mathfrak{sk})$ being given by

$$\mathfrak{k} \leftarrow \mathsf{KEMDec}(c_0, \mathfrak{sk}), \quad m \leftarrow \mathsf{DEM}^{-1}(c_1, \mathfrak{k}).$$

6.2 Constructions based on RSA

The simplest public key encryption scheme is the RSA scheme, which is based on the difficulty of factoring integers. In the key generation algorithm, two large primes p and q are selected and multiplied together to form $N = p \cdot q$. Then a (usually small) integer e is selected which is co-prime to $\phi(N) = (p-1)\cdot(q-1)$. Finally, the integer d is found, using the extended Euclidean algorithm, such that $d = 1/e \pmod{\phi(N)}$. The public key is set to be $\mathfrak{pk} = (N, e)$, whereas the secret key is set to be $\mathfrak{sk} = (N, d)$. Note that given the $\mathfrak{pk}$ only, finding the secret key $\mathfrak{sk}$ is provably equivalent to factoring N.

The public/private keys are used via the RSA function $x \longrightarrow x^e \pmod N$, which has the inverse map $x \longrightarrow x^d \pmod N$. Thus, the RSA function is a trapdoor permutation on the group $(\mathbb{Z}/N\mathbb{Z})^*$. It is not believed that inverting the RSA map is equivalent to factoring, so inversion of this map is identified as a separate problem (the RSA problem). At the time of writing, one should select p and q to be primes of at least 1536 bits in length to obtain suitable security.

There are many historic ways of using the RSA function as a public key encryption scheme, many of which are now considered obsolete and/or insecure. The two recommended methodologies in using RSA for encryption are RSA-OAEP and RSA-KEM; which are both IND-CCA secure in the random oracle model.

OAEP, or Optimized Asymmetric Encryption Padding, is a method to use the RSA function directly as an IND-CCA public key encryption algorithm. OAEP is parametrized by two integers $k_0, k_1 \geq 128$ such that $n = \log_2 N - k_0 - k_1 \geq 0$. We then encrypt messages of at most n bits in length as follows, using hash functions (which are assumed to be random oracles for the security proof)

$$G : \{0,1\}^{k_0} \longrightarrow \{0,1\}^{n+k_1}$$
$$H : \{0,1\}^{n+k_1} \longrightarrow \{0,1\}^{k_0}.$$

We then encrypt using the function

$$c \leftarrow \left(\left\{\left(m \parallel 0^{k_1}\right) \oplus G(R)\right\} \parallel \left\{R \oplus H\left(\left(m \| 0^{k_1}\right) \oplus G(R)\right)\right\}\right)^e \pmod N$$

where

- $m \parallel 0^{k_1}$ means m followed by k_1 zero bits,
- R is a random bit string of length k_0,
- $\parallel$ denotes concatenation.

RSA-KEM, on the other hand, is a KEM which is much simpler to execute. To produce the encapsulated key and the ciphertext, one takes the random input r (which one thinks of as a uniformly random element in $(\mathbb{Z}/N\mathbb{Z})^*$). Then the KEM is defined by

$$c \leftarrow r^e \pmod{N}, \;\; \mathfrak{k} \leftarrow H(r),$$

where $H : (\mathbb{Z}/N\mathbb{Z}) \longrightarrow \mathcal{K}$ is a hash function, which we model as a random oracle.

6.3 Constructions based on Elliptic Curves

Elliptic Curve Cryptography, or ECC, uses the fact that elliptic curves form a finite abelian group. In terms of encryption schemes, the standard method is to use ECIES (Elliptic Curve Integrated Encryption Scheme) to define a public key, KEM which is IND-CCA in the random oracle model, assuming the DDH problem in the subgroup of the elliptic curve being used. In practice, this means that one selects a curve $E(\mathbb{F}_p)$ for which there is a point $P \in E(\mathbb{F}_p)$ whose order is a prime $q > 2^{256}$.

For ECIES, the KeyGen algorithm is defined as follows. A secret key $\mathfrak{sk} \leftarrow \mathbb{F}_q^*$ is selected uniformly at random, and then the public key is set to be $Q \leftarrow [\mathfrak{sk}]P$. Key encapsulation is very similar to RSA-KEM in that it is defined by

$$r \leftarrow \mathbb{F}_q^*, \;\; C \leftarrow [r] \cdot P, \;\; \mathfrak{k} \leftarrow H([r] \cdot Q),$$

where $H : E(\mathbb{F}_p) \longrightarrow \mathcal{K}$ is a hash function (modelled as a random oracle). To decapsulate the key is recovered via

$$\mathfrak{k} \leftarrow H([\mathfrak{sk}]C).$$

Compared to RSA-based primitives, ECC-based primitives are relatively fast and use less bandwidth. This is because, at the time of writing, one can select elliptic curve parameters with $p \approx q \approx 2^{256}$ to obtain security equivalent to a work-factor of 2^{128} operations. Hence, in current systems elliptic curve-based systems are preferred over RSA-based ones.

6.4 Lattice-based Constructions

A major problem with both RSA and ECC primitives is that they are not secure against quantum computers; namely, Shor's algorithm will break both the RSA and ECC hard problems in polynomial time. Hence, the search is on for public key schemes which would resist the advent of a quantum computer. The National Institute of Standards and Technology (NIST) is currently engaged in a process to determine potential schemes which are *post-quantum* secure, see [17] for more details on this.

The most prominent of these so-called post-quantum schemes are those based on hard problems on lattices. In particular, the NTRU schemes and a variety of schemes based on the Learning With Errors (LWE) problem, and its generalisation to polynomial rings, known as the Ring-LWE problem. There are other proposals based on hard probles in coding theory, on the difficulty of computing isogenies between elliptic curves and other constructs. NIST is currently conducting a program to select potential post-quantum replacements.

7 PUBLIC KEY SIGNATURES

[3, c12][4, c16]

Public key encryption assumes that the receivers public key is known to be associated with the physical entity that the sender wishes to communicate with. This binding of public key with an entity is done via means of a so called *digital certificate*. A digital certificate is a signed statement that a given entity is associated with a given public key. This certificate is issued by a certificate authority, and utilizes the second main public key construct; namely a digital signature.

Just as with public key encryption algorithms, modern digital signature algorithms are based either on the RSA problem or a variant of the DLP problem; hence the reader is also directed again to [5] for more advanced details. For post-quantum security, there are a number of proposals based on lattice constructions; but none have yet been widely accepted or deployed at the time of writing this document. Again for PQC signatures we refer to the current NIST process [17].

The prototypical digital signature scheme given in text-books is loosely called RSA-FDH, where FDH stands for Full Domain Hash. The algorithm takes a message m and signs it by outputting

$$s = H(m)^d \pmod{N}.$$

Verification is then performed by testing whether the following equation holds

$$s^e = H(m) \pmod{N}.$$

Here, the hash function is assumed to have domain $\{0,1\}^*$ and co-domain $(\mathbb{Z}/N\mathbb{Z})^*$. This scheme comes with a security proof in the random oracle model, but is almost impossible to implement as no standardized hash function has co-domain the whole of $(\mathbb{Z}/N\mathbb{Z})^*$, since N is much bigger than the output of hash functions such as SHA-2.

All standardized hash functions output a value in $\{0,1\}^t$ for some t, thus what is usually done is to take a hash value and then prepend it with some known pre-determined values, and then 'sign' the result. This forms the basic idea behind the Public Key Cryptography Standards (PKCS) v1.5 signature standard. This signs a message by computing

$$s = (0x01 \| 0xFF \ldots 0xFF \| 0x00 \| H(m))^d \pmod{N},$$

where enough padding of $0xFF$ bytes is done to ensure the whole padded string is just less than N in size. Despite the close relationship to RSA-FDH, the above signature scheme has no proof of security, and hence a more modern scheme is usually to be preferred.

7.1 RSA-PSS

The modern way to use the RSA primitive in a digital signature scheme is via the padding method called PSS (Probabilistic Signature Scheme). This is defined much like RSA-OAEP via the use of two hash functions, one which expands data and one which compresses data:

$$G : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k-k_1-1},$$
$$H : \{0,1\}^* \longrightarrow \{0,1\}^{k_1},$$

where $k = \log_2 N$. From G we define two auxiliary functions

$$G_1 : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k_0}$$

which returns the first k_0 bits of $G(w)$ for $w \in \{0,1\}^{k_1}$,

$$G_2 : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k-k_0-k_1-1}$$

which returns the last $k - k_0 - k_1 - 1$ bits of $G(w)$ for $w \in \{0,1\}^{k_1}$, i.e. $G(w) = G_1(w)\|G_2(w)$.

To sign a message m the private key holder performs the following steps:

- $r \leftarrow \{0,1\}^{k_0}$.
- $w \leftarrow H(m\|r)$.
- $y \leftarrow 0\|w\|(G_1(w) \oplus r)\|G_2(w)$.
- $s \leftarrow y^d \pmod N$.

To verify a signature (s, m) the public key holder performs the following

- $y \leftarrow s^e \pmod N$.
- Split y into the components $b\|w\|\alpha\|\gamma$ where b is one bit long, w is k_1 bits long, α is k_0 bits long and γ is $k - k_0 - k_1 - 1$ bits long.
- $r \leftarrow \alpha \oplus G_1(w)$.
- The signature is verified as correct if and only if b is the zero bit, $G_2(w) = \gamma$ and $H(m\|r) = w$.

Despite being more complicated than PKCS-1.5, the RSA-PSS has a number of advantages. It is a randomized signature scheme, i.e. each application of the signing algorithm on the same message will produce a distinct signature, and it has a proof of security in the random oracle model relative to the difficulty of solving the RSA problem.

7.2 DSA, EC-DSA and Schnorr Signatures

The standard methodology for performing signatures in the discrete logarithm setting is to adapt interactive verification protocols, using proofs of knowledge of a discrete logarithm (see Sections 8.1 and 9.3), and then to convert them into a non-interactive signature scheme using a hash function.

The two most well known of these are the DSA (Digital Signature Algorithm) method and a method due to Schnorr. The former has widespread deployment but establishing security via security proofs uses less well-accepted assumptions, whereas the latter is less deployed but has well-established security proofs. The former also has, as we shall see, a more complex signing process. Both cases use a hash function H with co-domain $(\mathbb{Z}/q\mathbb{Z})^*$, unlike RSA-FDH this is easy to construct as q is relatively small.

We will describe both algorithms in the context of elliptic curves. Both make use of a public key of the form $Q = [x] \cdot P$, where x is the secret key. To sign a message m, in both algorithms, one first selects a random value $k \in (\mathbb{Z}/q\mathbb{Z})^*$, and computes $r \leftarrow \mathsf{x} - \mathsf{coord}([k] \cdot P)$. One then computes a hash of the message. In the DSA algorithm, this is done with $e \leftarrow H(m)$, whereas for the Schnorr algorithm, one computes it via $e \leftarrow H(m\|r)$. Then the signature equation is applied which, in the case of EC-DSA, is

$$s \leftarrow (e + x \cdot r)/k \pmod q$$

and, in the case of Schnorr, is

$$s \leftarrow (k + e \cdot x) \pmod q.$$

Finally, the output signature is given by (r, s) for EC-DSA and (e, s) for Schnorr.

Verification is done by checking the equation

$$r = \mathsf{x} - \mathsf{coord}([e/s] \cdot P + [r/s] \cdot Q)$$

in the case of EC-DSA, and by checking

$$e = H\left(m \| \mathsf{x} - \mathsf{coord}([s] \cdot P - e \cdot Q)\right)$$

in the case of Schnorr. The key difference in the two algorithms is not the signing and verification equations (although these do affect performance), but the fact that, with the Schnorr scheme, the r value is also entered into the hash function to produce e. This small distinction results in the different provable security properties of the two algorithms.

A key aspect of both EC-DSA and Schnorr signatures is that they are very brittle to exposure of the per-message random nonce k. If only a small number of bits of k leak to the attacker with every signing operation, then the attacker can easily recover the secret key.

8 STANDARD PROTOCOLS

[4, c18]

Cryptographic protocols are interactive operations conducted between two or more parties in order to realize some cryptographic goal. Almost all cryptographic protocols make use of the primitives we have already discussed (encryption, message authentication, secret sharing). In this section, we discuss the two most basic forms of protocol, namely authentication and key agreement.

8.1 Authentication Protocols

In an authentication protocol, one entity (the Prover) convinces the other entity (the Verifier) that they are who they claim to be, and that they are 'online'; where 'online' means that the verifying party is assured that the proving party is actually responding and it is not a replay. There are three basic types of protocol: Encryption based, Message Authentication based and Zero-Knowledge based.

8.1.1 Encryption-Based Protocols

These can operate in the symmetric or public key setting. In the symmetric key setting, both the prover and the verifier hold the same secret key, whilst in the public key setting, the prover holds the private key and the verifier holds the public key. In both settings, the verifier first encrypts a random nonce to the prover, the prover then decrypts this and returns it to the verifier, the verifier checks that the random nonce and the returned value are equivalent.

$$\begin{array}{rcl}
\underline{\text{Verifier}} & & \underline{\text{Prover}} \\
N \leftarrow \mathcal{M} & & \\
c \leftarrow \mathsf{Enc}(N, \mathfrak{pk}; r) & \xrightarrow{c} & \\
 & \xleftarrow{m} & m \leftarrow \mathsf{Dec}(c, \mathfrak{sk}) \\
N \overset{?}{=} m & &
\end{array}$$

The encryption scheme needs to be IND-CCA secure for the above protocol to be secure against active attacks. The nonce N is used to prevent replay attacks.

8.1.2 Message Authentication-Based Protocols

These also operate in the public key or the symmetric setting. In these protocols, the verifier sends a nonce in the clear to the prover, the prover then produces a digital signature (or a MAC in the symmetric key setting) on this nonce and passes it back to the verifier. The verifier then verifies the digital signature (or verifies the MAC). In the following diagram we give the public key/digital signature based variant.

$$\begin{array}{rcl}
\underline{\text{Verifier}} & & \underline{\text{Prover}} \\
N \leftarrow \mathcal{M} & \xrightarrow{N} & \\
 & \xrightarrow{\sigma} & \sigma \leftarrow \mathsf{Sign}(N, \mathfrak{sk}) \\
\mathsf{Verify}(N, \sigma, \mathfrak{pk}) \overset{?}{=} \mathsf{true} & &
\end{array}$$

8.1.3 Zero-Knowledge-Based

Zero-knowledge-based authentication protocols are the simplest examples of zero-knowledge protocols (see Section 9.3) available. The basic protocol is a so-called Σ- (or Sigma-) protocol consisting of three message flows; a commitment, a challenge and a response. The simplest example is the Schnorr identification protocol, based on the hardness of computing discrete logarithms. In this protocol, the Prover is assumed to have a long-term secret x and an associated public key $Q = [x] \cdot P$. One should note the similarity of this protocol to the Schnorr signature scheme above.

$$\begin{array}{rcl}
\underline{\text{Verifier}} & & \underline{\text{Prover}} \\
 & & k \leftarrow \mathbb{Z}/q\mathbb{Z} \\
 & \xleftarrow{r} & R \leftarrow [k] \cdot P \\
e \leftarrow \mathbb{Z}/q\mathbb{Z} & \xrightarrow{e} & \\
 & \xleftarrow{s} & s \leftarrow (k + e \cdot x) \pmod q \\
R \overset{?}{=} [s] \cdot P - e \cdot Q & &
\end{array}$$

Indeed, the conversion of the Schnorr authentication protocol into the Schnorr signature scheme is an example of the Fiat–Shamir transform, which transforms any Σ-protocol into a signature scheme. If the underlying Σ-protocol is secure, in the sense of a zero-knowledge proofs of knowledge (see Section 9.3), then the resulting signature scheme is UF-CMA.

8.2 Key Agreement Protocols

A key agreement protocol allows two parties to agree on a secret key for use in subsequent protocols. The security requirements of key agreement protocols are very subtle, leading to various subtle security properties that many deployed protocols may or may not have. We recap on basic properties of key agreement protocols here, but a more complete discussion can be found in [18]. The basic security requirements are

- The underlying key should be indistinguishable from random to the adversary, or that at least it should be able to be used in the subsequent protocol without the adversary breaking the subsequent protocol.
- Each party is assured that only the other party has access to the secret key. This is so-called mutual authentication. In many application scenarios (e.g. in the standard application of Transport Layer Security (TLS) to web browsing protocol), one only requires this property of one-party, in which case we are said to only have one-way authentication.

Kerberos is an example of a (usually) symmetric key-based key agreement system. This is a protocol that requires trusted parties to relay and generate secret keys from one party to another. It is most suited to closed corporate networks. On the public internet, protocols like Kerberos are less useful. Thus, here one uses public key-based protocols such as TLS and IPSec. More advanced properties required of modern public key-based protocols are as follows.

- **Key Confirmation:** The parties know that the other party has received the same secret key. Sometimes this can be eliminated as the correct execution of the subsequent protocol using the secret key provides this confirmation. This later process is called *implicit key confirmation*.
- **Forward Secrecy:** The compromise of a participant's long-term secret in the future does not compromise the security of the secret key derived now, i.e. current conversations are still secure in the future.
- **Unknown Key Share Security:** This prevents one party (Alice) sharing a key with Bob, whereas Bob thinks he shares a key with Charlie, despite sharing it with Alice.

Variations on the theme of key agreement protocols include group key agreement, which enables a group of users to agree on a key, or password based key agreement, in which two parties only agree on a (high entropy) key if they also agree on a shared password.

8.2.1 Key Transport

The most basic form of key agreement protocol is a form of key transport in which the parties use public key encryption to exchange a random key. In the case of a one-way authenticated protocol, this was the traditional method of TLS operation (up until TLS version 1.2) between a server and a client

$$\begin{array}{ccc} \underline{\text{Client}} & & \underline{\text{Server}} \\ & \xleftarrow{\mathfrak{pk}} & \\ k \leftarrow \mathcal{K} & & \\ c \leftarrow \mathsf{Enc}(k, \mathfrak{pk}; r) & \xrightarrow{c} & \\ & & k \leftarrow \mathsf{Dec}(c, \mathfrak{sk}) \end{array}$$

This protocol produced the pre-master secret in older versions of TLS (pre-TLS 1.2). To derive the final secret in TLS, further nonces were exchanged between the parties (to ensure that both parties were alive and the key was fresh). Then, a master secret was derived from the pre-master secret and the nonces. Finally, key confirmation was provided by the entire protocol transcript being hashed and encrypted under the master secret (the so-called *FINISHED* message). In TLS, the resulting key is not indistinguishable from random as the encrypted FINISHED message provides the adversary with a trivial check to determine whether a key is real or not. However, the protocol can be shown to be secure for the purposes of using the master secret to produce a secure bi-directional channel between the server and the client.

A more basic issue with the above protocol is that it is not forward-secure. Any adversary who records a session now, and in the future manages to obtain the server's long-term secret $\mathfrak{sk}$, can obtain the pre-master secret, and hence decrypt the entire session.

8.2.2 Diffie–Hellman Key Agreement

To avoid the issues with forward secrecy of RSA-based key transport, modern protocols make use of Diffie–Hellman key exchange. This allows two parties to agree on a uniformly random key, which is indistinguishable from random assuming the Decision Diffie–Hellman problem is hard

$$
\begin{array}{ccc}
\underline{\text{Alice}} & & \underline{\text{Bob}} \\
a \leftarrow \mathbb{Z}/q\mathbb{Z} & & b \leftarrow \mathbb{Z}/q\mathbb{Z} \\
Q_A \leftarrow [a] \cdot P & \xrightarrow{Q_A} & \\
 & \xleftarrow{Q_B} & Q_B \leftarrow [b] \cdot P \\
K \leftarrow [a] \cdot Q_B & & K \leftarrow [b] \cdot Q_A
\end{array}
$$

This protocol provides forward secrecy, but provides no form of authentication. Due to this, the protocol suffers from a man-in-the-middle attack. To obtain mutual authentication, the message flow of Q_A is signed by Alice's public key and the message flow of Q_B is signed by Bob's public key. This prevents the man-in-the-middle attack. However, since the signatures are not bound into the message, the signed-Diffie–Hellman protocol suffers from an unknown-key-share attack; an adversary (Charlie) can strip Alice's signature from Q_A and replace it with their signature. The adversary does not learn the secret, but does convince Bob he is talking to another entity.

The one-way authenticated version of Diffie–Hellman key agreement is the preferred method of key agreement in modern TLS deployments, and is the only method of key agreement supported by TLS 1.3. In TLS, the FINISHED message, which hashes the entire transcript, prevents the above unknown-key-share attack. However, it also prevents the protocol from producing keys which are indistinguishable from random, as mentioned above.

8.2.3 Station-to-Station Protocol

The Station-to-Station (STS) protocol can be used to prevent unknown-key-share attacks on signed Diffie–Hellman and maintain key indistinguishability. In this protocol, the Diffie–Hellman derived key is used to encrypt the signatures, thus ensuring the signatures cannot be stripped off the messages. In addition, the signatures are applied to the transcript so as to convince both receiving parties that the other party is ‘alive’.

$$\begin{array}{ccc}
\underline{\text{Alice}} & & \underline{\text{Bob}} \\
a \leftarrow \mathbb{Z}/q\mathbb{Z} & & b \leftarrow \mathbb{Z}/q\mathbb{Z} \\
Q_A \leftarrow [a] \cdot P & \xrightarrow{Q_A} & \\
 & & Q_B \leftarrow [b] \cdot P \\
 & & K \leftarrow [b] \cdot Q_A \\
 & & \sigma_B \leftarrow \mathsf{Sign}(\{Q_B, Q_A\}, \mathfrak{sk}_B) \\
 & \xleftarrow{Q_B, c_B} & c_B \leftarrow \mathsf{Enc}_K(\sigma_B) \\
K \leftarrow [a] \cdot Q_B & & \\
\sigma_B \leftarrow \mathsf{Dec}_K(c_B) & & \\
\mathsf{Verify}(\{Q_B, Q_A\}, \sigma_B, \mathfrak{pk}_B) \overset{?}{=} \mathsf{true} & & \\
\sigma_A \leftarrow \mathsf{Sign}(\{Q_A, Q_B\}, \mathfrak{sk}_A) & & \\
c_A \leftarrow \mathsf{Enc}_K(\sigma_A) & \xrightarrow{c_A} & \\
 & & \sigma_A \leftarrow \mathsf{Dec}_K(c_A) \\
 & & \mathsf{Verify}(\{Q_A, Q_B\}, \sigma_A, \mathfrak{pk}_A) \overset{?}{=} \mathsf{true}
\end{array}$$

9 ADVANCED PROTOCOLS

[4, c20–c22]

Modern cryptography has examined a number of more complex protocols to achieve more complex ends. For example, secure e-voting schemes, secure auctions, data storage and retrieval, etc. Most of these advanced protocols are either based on the simpler components described in this section and/or on the encryption and signature schemes with special properties discussed in the next section. Here we restrict ourselves to discussing three widely needed protocols: Oblivious Transfer, Zero-Knowledge and Multi-Party Computation.

9.1 Oblivious Transfer

While Oblivious Transfer (OT) is at the heart of many advanced protocols, it is a surprisingly simple primitive which enables one to accomplish various more complex tasks. In the following, we describe the basic 1-out-of-2 Oblivious Transfer, but extensions to n-out-of-m are immediate. In all cases, the protocol is one executed between a Sender and a Receiver.

In a 1-out-of-2 Oblivious Transfer, the Sender has two messages m_0 and m_1, whilst the Receiver has an input bit b. The output of the protocol should be the message m_b for the Receiver, and the Sender obtains no output. In particular, the Receiver learns nothing about the message m_{1-b}, whilst the Sender learns nothing about the bit b.

This passively secure protocol can be implemented as follows. We assume the Sender’s messages are two elements M_0 and M_1 in an elliptic curve group $E(\mathbb{F}_p)$ of prime order q.

$$\begin{array}{rcl}
\underline{\text{Sender}} & & \underline{\text{Receiver}} \\
C \leftarrow E(\mathbb{F}_p) & \xrightarrow{C} & \\
 & & x \leftarrow (\mathbb{Z}/q\mathbb{Z}) \\
 & & Q_b \leftarrow [x] \cdot P \\
 & & Q_{1-b} \leftarrow C - Q_b \\
 & \xleftarrow{Q_0} & \\
Q_1 \leftarrow C - Q_0 & & \\
k \leftarrow (\mathbb{Z}/q\mathbb{Z}) & & \\
C_1 \leftarrow [k] \cdot P & & \\
E_0 \leftarrow M_0 + [k] \cdot Q_0 & & \\
E_1 \leftarrow M_1 + [k] \cdot Q_1 & & \\
 & \xrightarrow{C_1, E_0, E_1} & \\
 & & M_b \leftarrow E_b - [x] \cdot C_1
\end{array}$$

The extension to an actively secure protocol is only a little more complex, but beyond the scope of this article.

9.2 Private Information Retrieval and ORAM

A Private Information Retrieval (PIR) protocol is one which enables a computer to retrieve data from a server held database, without revealing the exact item which is retrieved. If the server has n data items then this is related to a 1-out-of-n OT protocol. However, in PIR we do not insist that the user does not learn anything else about the servers data, we only care about privacy of the user query. In addition protocols for PIR are meant to be run many times, and we are interested in hiding the total set of access patterns, i.e. even whether a data item is retrieved multiple times. The goal of PIR protocols is to obtain greater efficiency than the trivial solution of the server sending the user the entire database.

An Oblivious Random Access Memory (ORAM) protocol is similar but now we not only allow the user to obliviously read from the server's database, we also allow the user to write to the database. So as to protect the write queries the server held database must now be held in an encrypted form (so what is written cannot be determined by the server). In addition the access patterns, i.e. where data is written to and read from, needs to be hidden from the server.

9.3 Zero-Knowledge

A Zero-Knowledge protocol is a protocol executed between a Prover and a Verifier in which the Prover demonstrates that a statement is true, without revealing why the statement is true. The concept is used in many places in cryptography, to construct signature schemes, to attest ones identity, and to construct more advanced protocols. An introduction to the more theoretical aspects of zero-knowledge can be found in [6]. More formally, consider an NP language $\mathcal{L}$ (i.e. a set of statements x which can be verified to be true in polynomial time given a witness or proof w). An interactive proof system for $\mathcal{L}$ is a sequence of protocol executions by an (infinitely powerful) Prover P and a (probabilistic polynomial time) Verifier V, which on joint input x proceeds as follows:

$$\begin{array}{rcl}
\underline{\text{Verifier}} & & \underline{\text{Prover}} \\
 & \xleftarrow{p_1} & (p_1, s'_1) \leftarrow P_1(x) \\
(v_1, s_1) \leftarrow V_1(x, p_1) & \xrightarrow{v_1} & \\
 & \xleftarrow{p_2} & (p_2, s'_2) \leftarrow P_2(s'_1, v_1) \\
(v_2, s_2) \leftarrow V_2(s_1, p_2) & \xrightarrow{v_2} & \\
 & \xleftarrow{p_3} & (p_3, s'_3) \leftarrow P_3(s'_2, v_2) \\
\vdots & & \vdots \\
 & \xrightarrow{p_r} & (p_r, s'_r) \leftarrow P_r(s_{r_1}, v_{r_1})
\end{array}$$

By the end of the protocol, the Verifier will output either true or false. An interactive proof system is one which is both *complete* and *sound*

- **Completeness:** If the statement x is true, i.e. $x \in \mathcal{L}$, then if the Prover is honest then the Verifier will output true.
- **Soundness:** If the statement is false, i.e. $x \notin \mathcal{L}$, then no cheating Prover can convince the Verifier with probability greater than p.

Note that even if p is large (say $p = 0.5$) then repeating the proof multiple times can reduce the soundness probability to anything desired. Of course, protocols with small p to start with are going to be more efficient.

For any NP statement, there is a trivial proof system. Namely, the Prover simply sends over the witness w which the Verifier then verifies. However, this reveals the witness. In a zero-knowledge proof, we obtain the same goal, but the Verifier learns nothing bar the fact that $x \in \mathcal{L}$. To formally define zero-knowledge, we insist that there is a (probabilistic polynomial time) *simulator* S which can produce protocol transcripts identical to the transcripts produced between a Verifier and an honest Prover; except the simulator has no access to the Prover. This implies that the Verifier cannot use the transcript to perform any other task, since what it learned from the transcript it could have produced without the Prover by simply running the simulator.

A zero-knowledge proof is said to be *perfect zero-knowledge* if the distribution of transcripts produced by the simulator is identical to those produced between a valid prover and verifier. If the two distributions only cannot be distinguished by an efficient algorithm we say we have *computational zero-knowledge*.

A zero-knowledge proof is said to be a *proof of knowledge* if a Verifier given rewinding access to the prover (i.e. the Verifier can keep resetting the Prover to a previous protocol state and continue executing) can extract the underlying witness w. This implies that the Prover must 'know' w since we can extract w from it.

A *non-interactive zero-knowledge proof* is one in which there is no message flowing from the Verifier to the Prover, and only one message flowing from the Prover to the Verifier. Such non-interactive proofs require additional setup assumptions, such as a Common Reference String (CRS), or they require one to assume the Random Oracle Model. Traditionally these are applied to specific number theoretic statements, such to show knowledge of a discrete logarithm (see the next section on Σ-protocols), however recently so called Succinct Non-Interactive Arguments of Knowledge (SNARKs) have been developed which enable such non-interactive arguments for more complex statements. Such SNARKs are finding applications in some blockchain systems.

9.3.1 Σ-Protocols

The earlier Σ-protocol for identification is a zero-knowledge proof of knowledge.

$$\begin{array}{ccc}
\underline{\text{Verifier}} & & \underline{\text{Prover}} \\
 & & k \leftarrow \mathbb{Z}/q\mathbb{Z} \\
 & \xleftarrow{R} & R \leftarrow [k] \cdot P \\
e \leftarrow \mathbb{Z}/q\mathbb{Z} & \xrightarrow{e} & \\
 & \xleftarrow{s} & s \leftarrow (k + e \cdot x) \pmod q \\
R \overset{?}{=} [s] \cdot P - e \cdot Q & &
\end{array}$$

The protocol is obviously complete since $Q = [x] \cdot P$, and the soundness error is $1/q$. That it is zero-knowledge follows from the following simulation, which first samples $e, s \leftarrow \mathbb{Z}/q\mathbb{Z}$ and then computes $R = [s]P - e \cdot Q$; the resulting simulated transcript being (R, e, s). Namely, the simulator computes things in the wrong order.

The protocol is also a proof of knowledge since if we execute two protocol runs with the same R value but different e-values (e_1 and e_2) then we obtain two s-values (s_1 and s_2). This is done by rewinding the prover to just after it has sent its first message. If the two obtained transcripts (R, e_1, s_1) and (R, e_2, s_2) are both valid then we have

$$R = [s_1] \cdot P - e_1 \cdot Q = [s_2] \cdot P - e_2 \cdot Q$$

and so

$$[s_1 - s_2] \cdot P = [e_1 - e_2] \cdot Q$$

and hence

$$Q = \left[\frac{s_1 - s_2}{e_1 - e_2} \right] \cdot P$$

and hence we 'extract' the secret x from $x = (s_1 - s_2)/(e_1 - e_2) \pmod q$.

9.4 Secure Multi-Party Computation

Multi-Party Computation (MPC) is a technique to enable a set of parties to compute on data, without learning anything about the data. Consider n parties $P_1, \ldots, P_n$ each with input $x_1, \ldots, x_n$. MPC allows these parties to compute *any* function $f(x_1, \ldots, x_n)$ of these inputs without revealing any information about the x_i to each other, bar what can be deduced from the output of the function f. A general introduction to the theory of such protocols can be found in [7].

In an MPC protocol, we assume that a subset of the parties A is corrupt. In *statically* secure protocols, this set is defined at the start of the protocol, but remains unknown to the honest parties. In an *adaptively* secure protocol, the set can be chosen by the adversary as the protocol progresses. An MPC protocol is said to be *passively* secure if the parties in A follow the protocol, but try to learn data about the honest parties' inputs from their joint view. In an *actively* secure protocol, the parties in A can arbitrarily deviate from the protocol.

An MPC protocol should be *correct*, i.e. it outputs the correct answer if all parties follow the protocol. It should also be *secure*, i.e. the dishonest parties should learn nothing about the inputs of the honest parties. In the case of active adversaries, a protocol is said to be *robust* if the honest parties will obtain the correct output, even when the dishonest parties deviate from the protocol. A protocol which is not robust, but which aborts with overwhelming probability when a dishonest party deviates, is said to be an actively secure MPC protocol *with abort*.

MPC protocols are catagorized by whether they utilize information-theoretic primitives (namely secret sharing), or they utilize computationally secure primitives (such as symmetric-key and public-key encryption). They are also further characterized by the properties of the set A. Of particular interest is when the size t of A is bounded by a function of n (so-called threshold schemes). The cases of particular interest are $t < n$, $t < n/2$, and $t < n/3$; the threshold cases of $t < n/2$ and $t < n/3$ can be generalized to Q_2 and Q_3 access structures, as discussed in Section 3.2.

In the information-theoretic setting, one can achieve passively secure MPC in the case of $t < n/2$ (or Q_2 access structures). Actively secure robust MPC is possible in the information-theoretic setting when we have $t < n/3$ (or Q_3 access structures). All of these protocols are achieved using secret sharing schemes. A detailed study of secret sharing based MPC protocols is given in [19].

In the computational setting, one can achieve actively secure robust computation when $t < n/2$, using Oblivious Transfer as the basic computational foundation. The interesting case of two party computation is done using the Yao protocol. This protocol has one party (the Circuit Creator, also called the Garbler) 'encrypting' a boolean function gate by gate using a cipher such as AES, the circuit is then sent to the other party (called the Circuit Evaluator). The Evaluator then obtains the 'keys' for their input values from the Creator using Oblivious Transfer, and can then evaluate the circuit. A detailed study of two party Yao based protocols is given in [20].

Modern MPC protocols have looked at active security with abort in the case of $t < n$. The modern protocols are divided into a function-dependent offline phase, which requires public key functionality but which is function independent, then a function-dependent online phase which mainly uses information-theoretic primitives. Since information theoretic primitives are usually very fast, this means the time-critical online phase can be executed as fast as possible.

10 PUBLIC KEY ENCRYPTION/SIGNATURES WITH SPECIAL PROPERTIES

[3, c13]

A major part of modern cryptography over the last twenty years has been the construction of encryption and signature algorithms with special properties or advanced functionalities. A number of the following have been deployed in specialized systems (for example, U-PROVE, IDEMIX, attestation protocols and some cryptocurrencies). We recap the main variants below, giving for each one the basic idea behind their construction.

10.1 Group Signatures

A group signature scheme defined a group public key $\mathfrak{pk}$, associated to a number of secret keys $\mathfrak{sk}_1, \ldots, \mathfrak{sk}_n$. The public key is usually determined by an entity called a *Group Manager*, during an interaction with the group members. Given a group signature s, one cannot tell which secret key signed it, although one is guaranteed that one did. Thus group signatures provide the anonymity of a Signer. Most group signature algorithms have a special entity called an *Opener* who has some secret information which enables them to revoke the anonymity of a Signer. This last property ensures one can identify group members who act dishonestly in some way.

A group signature scheme can either support *static* or *dynamic* groups. In a static group signature scheme, the group members are fixed at the start of the protocol, when the public key is fixed. In a dynamic group signature scheme the group manager can add members into the group as the protocol proceeds, and (often) revoke members as well.

An example of this type of signature scheme which is currently deployed is the Direct Anonymous Attestation (DAA) protocol; which is essentially a group signature scheme in which the *Opener* is replaced with a form of user controlled linkability; i.e. a signer can decide whether two signatures output by the specific signer can be linked or not.

10.2 Ring Signatures

A ring signature scheme is much like a group signature scheme, but in a ring signature there is no group manager. Each user in a ring signature scheme has a public/private key pair $(\mathfrak{pk}_i, \mathfrak{sk}_i)$. At the point of signing, the Signer selects a subset of the public keys (containing this own), which is called a ring of public keys, and then produces a signature. The Receiver knows the signature was produced by someone in the ring, but not which member of the ring.

10.3 Blind Signatures

A blind signature scheme is a two party protocol in which a one party (the User) wants to obtain the signature on a message by a second party (the Signer). However, the Signer is not allowed to know which message is being signed. For example, the Signer may be simply notarising that something happened, but does not need to know precisely what. Security requires that the Signer should not learn anything about any message passed to it for signing, and the user should not obtain the signature on any message other than those they submitted for signing.

10.4 Identity-Based Encryption

In normal public key encryption, a user obtains a public key $\mathfrak{pk}$, along with a certificate C. The certificate is produced by a trusted third party, and binds the public key to the identity. Usually, a certificate is a digitally signed statement containing the public key and the associated user identity. So, when sending a message to Alice the Sender is sure that Alice is the legitimate holder of public key $\mathfrak{pk}$.

Identity Based Encryption (IBE) is an encryption scheme which dispenses with the need for certificate authorities, and certificates. To encrypt to a user, say Alice, we simply use her identity *Alice* as the public key, plus a global 'system' public key. However, to enable Alice to decrypt, we must have a trusted third party, called a Key Generation Centre, which can provide Alice with her secret key. This third party uses its knowledge of the 'system' secret key to be able to derive Alice's secret key. Whilst dispensing with certificates, an IBE system inherently has a notion of key escrow; the Key Generation Centre can decrypt all messages.

10.5 Linearly Homomorphic Encryption

In a linearly homomorphic encryption scheme one can perform a number of linear operations on ciphertexts, which result in a ciphertext encrypting a message having had the same operations performed on the plaintext. Thus, given two encryptions $c_1 \leftarrow \mathsf{Enc}(m_1, \mathfrak{pk}; r_1)$ and $c_2 \leftarrow \mathsf{Enc}(m_2, \mathfrak{pk}; r_2)$ one can form a 'sum' operation $c \leftarrow c_1 \oplus c_2$ such that c decrypts to $m_1 + m_2$. The standard example of such encryption schemes is the Paillier encryption scheme, which encrypts elements $m \in (\mathbb{Z}/N\mathbb{Z})$, for an RSA-modulus N by computing $c \leftarrow (1+N)^m \cdot r^N \pmod{N^2}$ where r is selected in $\mathbb{Z}/N\mathbb{Z}$.

Such encryption algorithms can never be IND-CCA secure, as the homomorphic property produces a trivial malleability which can be exploited by a CCA attacker. However, they can have applications in many interesting areas. For example, one can use a linearly homomorphic encryption scheme to add up votes in a digitally balloted election for two candidates, where each vote is an encryption of either the message zero or one.

10.6 Fully Homomorphic Encryption

Fully Homomorphic Encryption (or FHE) is an extension to linearly homomorphic encryption, in that one can not only homomorphically evaluate linear functions, but also non-linear ones. In particular, the ability to homomorphically evaluate both addition *and* multiplication on encrypted data enables one to (theoretically) evaluate any function. Applications of FHE which have been envisioned are things such as performing complex search queries on encrypted medical data etc. Thus, FHE is very interesting in a cloud environment.

All existing FHE schemes are highly inefficient. Thus only very simple functions can be evaluated in suitable time limits. A scheme which can perform homomorphic operations from a restricted class of functions (for example, to homomorphically evaluate all multi-variate polynomials of total degree five) is called a Somewhat Homomorphic Encryption (or SHE) scheme. Obviously, if the set of functions are all multi-variate polynomials of degree one, then the SHE scheme is a linear homomorphic encryption scheme.

11 IMPLEMENTATION ASPECTS

There are two aspects one needs to bear in mind with respect to cryptographic implementation. Firstly security and secondly performance.

In terms of security the main concern is one of side-channel attacks. These can be mounted against both hardware implementations, for example cryptographic circuits implemented on smart-cards, or against software implementations running on commodity processors. Any measurable difference which occurs when running an algorithm on one set of inputs versus another can lead to an attack. Such measurements may involve timing differences, power comsumption differences, differences in electromagnetic radiation, or even differences in the sound produced by the fan on the processor. It is even possible to mount remote side-channel attacks where one measures differences in response times from a remote server. A good survey of such attacks, focused on power analysis applied to symmetric algorithms such as AES, can be found in [21].

To protect against such side-channel attacks at the hardware level various techniques have been proposed including utilizing techniques based on secret-sharing (called *masking* in the side-channel community). In the area of software one needs to ensure code is constant-time at the least (i.e. every execution path takes the same amount of time), indeed having multiple execution paths can itself lead to attacks via power-analysis.

To enable increased performance it is becoming increasingly common for processor manufacturers to supply special instructions to enable improvements to cryptographic algorithms. This is similar to the multi-media extensions which have been common place for other applications for some decades. An example of this is special instructions on x86 chips to perform operations related to AES, to perform GCM-mode and to perform some ECC operations.

Public key, i.e. number theoretic constructions, are particularly expensive in terms of computational resources. Thus it is common for these specific algorithms to be implemented in low level machine code, which is tuned to a specific architecture. However, this needs to be done with care so as to take into account the earlier mentioned side-channel attacks.

Finally an implementation can also be prone to fault attacks. These are attacks in which an attacker injects faults (either physical faults on hardware, or datagram faults into a protocol).

Defences against such attacks need to be considered including standard fault tolerent computing approaches in hardware, and full input validation in all protocols. Further details on fault attacks can be found in [22].

CROSS-REFERENCE OF TOPICS VS REFERENCE MATERIAL

The two main textbooks below we cross-reference against the main sections here. Further topic specific reading is given by references to the main bibliography.

	[3]	[4]	Other
1 Mathematics	c8–c9, App B	c1–c5	[5]
2 Cryptographic Security Models	c1–c4	c11	[6, 7, 8, 9]
3 Information-theoretically Secure Constructions	c2	c19	[10]
4 Symmetric Primitives	c3–c6	c11–c14	[11, 12, 13, 14, 15]
5 Symmetric Encryption and Authentication	c3–c4	c13–c14	[15, 16]
6 Public Key Encryption	c11	c15–c17	[5, 17]
7 Public Key Signatures	c12	c16	[5, 17]
8 Standard Protocols	–	c18	[18]
9 Advanced Protocols	–	c20–c22	[6, 7, 19, 20]
10 Public Key Encryption/Signatures With Special Properties	c13	–	
11 Implementation Aspects	–	–	[21, 22]

FURTHER READING

The following two text books are recommended to obtain further information on the topics in this knowledge area. Further topic specific reading is given in the bibliography.

Introduction to Modern Cryptography (J. Katz and Y. Lindell) [3]

A standard modern textbook covering aspects of the design of cryptographic schemes from a provable security perspective.

Cryptography Made Simple (N.P. Smart) [4]

A textbook with less mathematical rigour than the previously mentioned one, but which also covers a wider range of areas (including zero-knowledge and MPC), and touches on aspects related to implementation.

978-3-642-29656-7

ACRONYMS

AEAD	Authenticated Encryption with Associated Data.
AES	Advanced Encryption Standard.
CBC	Cipher Block Chaining.
CCA	Chosen Ciphertext Attack.
CFB	Cipher Feedback.
CMA	Chosen Message Attack.
CPA	Chosen Plaintext Attack.
CRS	Common Reference String.
CTR	Counter Mode.
CVP	Closest Vector Problem.
DAA	Direct Anonymous Attestation.
DDH	Decision Diffie–Hellman.
DEM	Data Encryption Mechanism.
DES	Data Encryption Standard.
DHP	Diffie–Hellman Problem.
DLP	Discrete Logarithm Problem.
DSA	Digital Signature Algorithm.
ECB	Electronic Code Book.
ECC	Elliptic Curve Cryptography.
ECIES	Elliptic Curve Integrated Encryption Scheme.
FDH	Full Domain Hash.
FHE	Fully Homomorphic Encryption.
GCM	Galois Counter Mode.
GMAC	Galois Message Authentication Code.
HMAC	Hash MAC.
IBE	Identity Based Encryption.
IND	Indistinguishable.

IV	Initialisation Vector.
KDF	Key Derivation Function.
KEM	Key Encapsulation Mechanism.
KMAC	Keccak MAC.
LWE	Learning With Errors.
MAC	Message Authentication Code.
MPC	Multi-Party Computation.
NIST	National Institute of Standards and Technology.
OAEP	Optimized Asymmetric Encryption Padding.
OFB	Output Feedback.
ORAM	Oblivious Random Access Memory.
OT	Oblivious Transfer.
OW	One-Way.
PASS	Passive Attack.
PIN	Personal Identification Number.
PIR	Private Information Retrieval.
PKCS	Public Key Cryptography Standards.
PKI	Public-Key Infrastructure.
PQC	Post-Quantum Cryptography.
PRF	Pseudo-Random Function.
PRP	Pseudo-Random Permutation.
PSS	Probabilistic Signature Scheme.
RSA	Rivest-Shamir-Adleman.
SHE	Somewhat Homomorphic Encryption.
SNARK	Succinct Non-Interactive Arguments of Knowledge.
STS	Station-to-Station.
SVP	Shortest Vector Problem.
TLS	Transport Layer Security.
UC	Universal Composability.

UF Universal Forgery.

XOF Extendable Output Function.