Cryptography

December 25, 2001

1 Introduction

"Cryptography" means literally "secret writing." And indeed the common perception is that the field of cryptography is concerned with private communication in the presence of eavesdroppers. But cryptography has come to encompass a whole host of problem domains. This article seeks to survey several key areas in cryptography giving a sense of the flavor of what is involved when designing, attacking, and using the algorithms at the center of the science.

1.1 History

Historically, cryptography was primarily a military concern; obviously it is profoundly important to disallow an enemy from obtaining information regarding the organization of troops, plans of attack, and so forth. Therefore military groups sought to obscure the content of their communications in such a way that the receiver of these communications could understand their contents, but enemy eavesdroppers would be unable to extract any useful information from them. The techniques used were–by today's standards–quite simplistic, consisting often of a simple shift of the alphabet. For example, the message "ATTACK AT DAWN" might become the same sentence with each character shifted one place forward in the alphabet (and Z would be replaced with A): "BUUBDL BU EBXO". Of course the receiver simply shifts each character one place backward to recover the original message.

Although the jumbled message looks meaningless at first glance, it is obviously not a difficult task for an eavesdropper to figure out the amount of shift being used and to extract the meaning of the original message. Such simple methods are completely ineffectual today, especially given the emergence of the modern computer.

Up until the mid-1960's, all popular and unclassified methods of private communication were relatively easy to attack. Although reasonable levels of sophistication were reached by the Enigma designers during World War II, even this method was eventually broken. In the modern era, the best methods have no known exploitable weaknesses, when used properly.

1.2 Areas of Cryptography

In the modern era, "cryptography" means, broadly, the science of deliberate difficulty, often in the service of communication in the presence of adversaries. This would include the scenario above as well as a whole host of others. "Cryptanalysis" is the art of breaking, attacking, or analyzing cryptographic methods. Together these two fields are called "cryptology" though often the term "cryptography" is used to mean this as well.

The science of "deliberate difficulty" sounds at first a bit paradoxical. After all, isn't it a primary goal of science to *circumvent* difficulty? A unique aspect of cryptography is that difficulty is often *sought*. We model the world as having friends and enemies, and we desire to communicate with our friends using problems which are intractable for our enemies.

1.2.1 Privacy

Privacy is perhaps the most well-known problem of cryptography. Informally, two parties, Alice and Bob, wish to communicate over an insecure channel. There is an eavesdropper Eve who wishes to obtain information about the content of this communication. The privacy problem is to provide Alice and Bob with

a method of communication such that (1) Alice and Bob can recover the contents of their messages with little computational overhead, and yet (2) Eve has no reasonable way of obtaining much information about the contents of these messages.

1.2.2 Authentication

Authentication is an equally-important cryptographic problem. Here Alice wishes to send a message to Bob in such a way that Bob can be virtually certain that Alice was the originator of the message. In other words, we wish to prevent an adversary from impersonating Alice and forging messages to Bob. An obvious example for this application would occur in, say, the banking industry: Alice wishes to tell the Bank to transfer 1,000 dollars from her account to Carl's account; if an impersonator could convince the Bank that he were Alice, he might send more or less money to Carl's account (against Alice's wishes) or he may divert Alice's money to another account under his own control.

1.2.3 Commitment Schemes

A **commitment** scheme is a method where Alice wishes to decide on a value (from some set of available options) and send this decision to Bob in such a way that (1) Bob cannot determine what value Alice chose and (2) Alice cannot change the value once it is in Bob's possession. Later, at some agreed upon time, Alice then sends Bob an additional piece of information to reveal to Bob the value she chose earlier. Commitment schemes are trivial in the presence of a trusted third party, Phil, since Alice could just tell Phil what the value is, and Phil would not divulge it to Bob until the proper moment. But in the absence of any trusted third party, more involved methods are required. Commitment schemes are used in a whole host of cryptographic applications, such as electronic voting.

1.2.4 Secret Sharing

Secret Sharing works as follows: suppose a group of people wish to share a secret by partitioning the secret into pieces (called "shares") in such a way that all of them must put their pieces together to reconstruct the secret. A real-world example occurs when Alice wishes to store a precious gem at the Bank in a safety-deposit box: the Bank issues Alice two keys, one to be held by Alice and one to be left at the Bank. The gem is then deposited in the box and locked. The box cannot be opened without *both* keys.

If we instead had a piece of *information* we wished to share among two or more people, we could use cryptographic methods to accomplish this. An example might be the launch codes for missiles: we would not want to endow a single person with the power to launch missiles; instead we might wish to require at least three of some five persons to submit their shares in order to induce a launch.

1.2.5 Random Number Generators

A **random number generator** is an old concept in computer science: randomness is used in a myriad of domains from simulation, game playing, computer algorithms, and so forth. However, in some applications a very strong random number generator is needed in the sense that the outputs are not only random but *unpredictable*. That is, given a stream of outputs from a random number generator, it should be infeasible to predict what the next output value will be. An example application of such an object might be this: say Alice wants to prove to Bob that she waited until Sunday to send a particular email. Assume that the local newspaper (which is immune to outside influence) publishes the 100-digit output of some random number generator each day. Alice could include Sunday's number with her email to prove that she did not send the message before Sunday.

In reality, random number generators are theoretical objects, existing only abstractly. In practical circumstances we create an algorithm which attempts to approximate a random number generator; the outputs are then called **pseudorandom numbers** and the generator is called a **pseudorandom number generator**.

1.2.6 A Host of Other Problems

Many other problems are within the domain of cryptography: digital cash, entity authentication, flipping a coin on the telephone, zero-knowledge proof systems, and so on. The common theme in all cryptographic

solutions to these problems is the idea that difficulty is designed into a system in order to foil attempts at defeating the proposed goals of the system.

1.3 Modern Cryptography

Modern cryptography is a branch of cryptography which is intimately tied with computation complexity theory, information theory, probability theory, computational number theory, and several other branches of mathematics. The central theme of modern cryptography is that, if at all possible, one should *rigorously prove* the security of the cryptographic methods being used. This is a fairly recent idea.

Historically, cryptographic methods were constructed by trial-and-error. Typically a method would be proposed, and after a time it would be broken. Often, the proposer would then suggest a "patch" or a "fix" that specifically addressed the attack which broke his proposed method. Then, after a time, it would be broken once more. Sometimes this process would "converge" to a method which seemed to work, and other times the proposer would simply give up on the idea and move on.

The modern approach seeks to avoid this process and proposes that we prove our methods have a certain level of security. The approach works like this: first, we state a **model** in which we are working. Then we state our **assumptions**. Then we exhibit a proof that the cryptographic system we are proposing meets a given level of security, in our model and given our assumptions.

Although modern cryptography is now widely-practiced by researchers, it is still not demanded by the user community at large. This is despite the fact that methods which use the historical approach are still broken regularly. This situation is likely to change in the future.

1.4 Cryptography in Practice

Given the wide range of cryptographic problems above, it is obviously undesirable to start from scratch in designing solutions for each one. Putting forth a different solution for each of these problems and then hoping or expecting that each solution would be without any serious flaws is unrealistic. Instead cryptographers have developed a toolbox of **cryptographic primitives**. These primitives solve a set of basic problems; they are then used as subroutines by higher-level **protocols**. Examples of both primitives and protocols are given shortly.

Oftentimes we are unable to prove the security of the primitives, but we are able to prove the security of the protocols under the assumption that the primitives are secure. This means that the security of the collection of cryptographic methods we employ rests upon the security of a handful of primitives. Although this strategy means that a break of one of the primitives often means the failure of all protocols which employ that primitive, most cryptographers believe that this is the proper way to build cryptographic systems because it allows them to focus on a handful of relatively simply algorithms rather than a wide variety of complex ones.

1.4.1 Performance

Although most cryptographic problems have good solutions available, it is still not generally true that cryptographic methods are widely employed. Certainly cryptography is used in virtually all highly-secret military communications in most countries, and it is certainly used between banks over their communications networks. Users commonly see the padlock icon when using a secure web-browser to make purchases via credit-card over the world-wide web. But other types of communications such as email, chat rooms, general web usage, etc., are largely without cryptographic protections. To some extent, this is because security was not built in to the system from the outset of the creation of the Internet (as it should have been). And to some extent it is because designers often consider the cost of cryptography, in terms of computing power, to be prohibitive. Therefore many researchers have focused on developing primitives and protocols which are both secure and *fast*. This thrust is likely to continue in the coming years.

2 Privacy

Privacy is perhaps the most well-known cryptographic goal. An informal description appeared in the introduction; we now explore some of the details.

2.1 The Model

Our model is as follows: Alice and Bob wish to communicate over a public channel. An eavesdropper, Eve, sees all communication between Alice and Bob, but we desire that she learns little to nothing about the content of the communication.

If we stop at this point in describing our model, the problem is unsolvable. There is nothing to stop Eve from learning everything Alice tells Bob over the public channel, and nothing to stop Eve from impersonating Bob to Alice in communications. We therefore must augment our model further. How we do this depends on whether we wish to use the **symmetric-key** or **asymmetric-key** model.

In the symmetric-key model, we enter the domain of **private-key cryptography**, and in the asymmetric-key model we use **public-key cryptography**. We explore the symmetric-key model first.

In the symmetric-key model, we assume that Alice and Bob both have a copy of some secret random bit string K which we call the **key**. It is further assumed that Eve has no information about K other than perhaps its length. We now ask, "is there any way that Alice and Bob can communicate over their public channel without imparting any useful information to Eve?"

In the asymmetric-key model, we do *not* assume Alice and Bob have a shared secret key; instead they must produce whatever secret information they require *in plain view* of Eve. It is, however, assumed that Eve will not inject messages on to the public channel in order to foil the exchanges between Alice and Bob. This requirement is perhaps unrealistic in some settings and therefore we address other models later on which do not have this requirement.

We now explore both private-key cryptography and public-key cryptography, providing solutions in their respective models.

2.2 Symmetric-Key Cryptography

The oldest solution to the privacy problem in the symmetric-key model is called the **one-time pad** or **Vernam Cipher**. We begin with a description of this solution.

2.2.1 The One-Time Pad

Suppose Alice wishes to send Bob one **bit** of information; that is, she either wishes to send $P = 0$, or $P = 1$ to Bob. Alice and Bob have pre-determined some key K which consists of a single bit, randomly chosen, which is also 0 or 1. (Perhaps the key was determined by physically flipping a coin and writing $K = 0$ if the coin came up heads, or $K = 1$ if the coin came up tails.) Alice now computes $P + K \bmod 2$, calling this result C. In other words, she adds P to K and writes $C = 1$ if the sum is 1 and $C = 0$ otherwise. Now she sends C to Bob over the public channel. If Eve views C what can she learn?

Eve learns two things: (1) Alice sent something to Bob; (2) the length of the message was at most 1 bit. Eve learns *nothing* beyond this; in particular, she learns nothing about the value of P the probability that $P = 0$ given the value of C is exactly the same as the probability that $P = 1$ given the value of C. (In either case, the probability is $1/2$.)

There is a problem with the one-time pad, however, which makes it impractical for most situations. If Alice now wishes to send another bit $P' = 0$ or $P' = 1$ to Bob, she cannot simply compute $P' + K \bmod 2$, calling this C', and then send C'. The problem is that Eve will then be able to determine something about the content of the communication she has viewed: she will realize that $C = C'$ implies $P = P'$. This is more than we want to allow Eve to know, and therefore means we must use a *new* key for each bit we wish to send.

This problem means that if Alice wishes to send n bits to Bob, she must have already established n random key bits with Bob in advance. Moreover, once these key bits have been used, they cannot be re-used in any future communications. Despite this rather significant practical drawback, the one-time pad is still

	Input Block Values							
	000	001	010	011	100	101	110	111
Key = 00	001	010	100	101	110	000	011	111
Key = 01	111	101	010	011	110	100	001	000
Key = 10	010	101	001	011	111	000	100	110
Key = 11	110	111	101	000	100	010	001	011

Figure 1: *Example of a Block Cipher. To encipher plaintext* $P \in \{0,1\}^3$ *under key* $K \in \{0,1\}^2$*, look up the entry in the row corresponding to key* K *and the column corresponding to input block* P*. Notice that each row is a permutation of all possible input blocks: each 3-bit string occurs exactly once. This is one of the* $(2^3!)^{2^2} = 2642908293365760000$ *possible 3-bit block ciphers with a 2-bit key.*

useful for extremely sensitive communications. It was rumored to be the method of choice for protecting the communications between Washington and Moscow, for a time.

The one-time pad, as given, handles only a single bit. It can be extended to handle **bit-strings** of any length in a straightforward manner. But before we explain how this is done, we define a few terms.

The bit-string which Alice wishes to send to Bob is called the **plaintext**. The process by which Alice converts the plaintext into another form for public viewing is called **encryption**. The encryption of the plaintext produces the **ciphertext**. When Bob receives the ciphertext he recovers the plaintext using **decryption**.

Given an n-bit plaintext message P and an n-bit key K, Alice computes the function given above for each bit in turn; that is, she computes $C_i = P_i + K_i \bmod 2$ for $1 \leq i \leq n$, where S_i denotes the i-th bit of a bit-string S numbering the bits from left-to-right starting at 1. This operation is also known as the "bitwise exclusive OR" of P and K, denoted $P \oplus K$.

2.2.2 Toward a Practical Solution

Given the drawbacks listed above, we might prefer a more practical solution to our problem. The cryptographic primitive known as a **block cipher** is intended for precisely this scenario. A block cipher is a function $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ such that, for any fixed value $K \in \{0,1\}^k$, the function $E(K,\cdot)$ is a permutation on $\{0,1\}^n$. In other words, a block cipher is a function E with two inputs, a key and a block, and one output, a block. The idea is this: when a key is fixed, the input block is **enciphered** to the output block. Saying the function is a permutation on $\{0,1\}^n$ means that each input block corresponds to exactly one output block, under any given key. This requirement ensures we can **decipher** an output block, ie, recover the original input block.

Figure 1 shows an example block cipher with 3-bit blocks and a 2-bit key. To encipher $P = 001$ under key $K = 01$, we simply look it up in the table to obtain $C = 101$.

Of course this table is very small; the small block size allows an eavesdropper to quickly build a dictionary of input and output values, and the small key size allows an attacker to exhaustively try all keys. The shift cipher, mentioned in the introduction, can also be viewed as a block cipher where the key is the amount of shift employed; that cipher has a similar problem: the number of keys and the block size are simply too small. Exhaustive key search and using statistical information (such as the frequency counts on the characters of the plaintext in whatever natural language is being used) will quickly defeat encryption based on such a weak block cipher. As an aside, it is important to note that in all of our solutions we insist only on the secrecy of the *key* and not of the *method*. Experience has shown that trying to provide security by hiding the algorithm as well as the key does not work: the algorithm is eventually discovered.

Real block ciphers have much larger block sizes and key lengths. In the mid-1970's, the National Bureau of Standards of the United States (now the National Institute of Standards and Technology) standardized the Data Encryption Standard (**DES**) as their standard block cipher. DES has a 64-bit block size and a 56-bit key length. Of course DES is not described via a table as we did above since the table would be astronomically large and DES would be impossible to write down; instead DES is an algorithm which induces such a table via a small finite number of steps.

Despite much scrutiny, no known practical break of DES was ever successful beyond exhaustive key search. A "break" can mean several things; an informal and often-used definition is as follows: we break DES if we can recover the key K in some "reasonable" amount of time, given some "reasonable" number of plaintext/ciphertext pairs under the key K. Another definition of "break" requires that the output of DES under some key and various distinct inputs can be distinguished from distinct random outputs.

Exhaustive key search is the only known way to break DES. Since there are 2^{56} possible DES keys, about half of them, or $2^{55} \approx 3.6 \times 10^{16}$ need to be searched for such an attack. This was largely infeasible in the mid 1970's, but with the advances made in CPU speeds, is quite possible for a modest budget. (A machine which finds a DES key in an expected 3.5 hours can be built for about $100,000 US.)

In 2001, DES was officially supplanted by the Advanced Encryption Standard (**AES**), an algorithm originally called "Rijndael" created by two Belgian cryptographers. AES has a block length of 128 bits and key length of 128, 192, or 256 bits. To search attack AES with a key length of 128 bits would mean testing an expected 2^{127} keys; this is considered well beyond what is possible for even the most sophisticated computers today. (To give an idea of the magnitude of this number, there are an estimated 2^{77} atoms in the known universe, a number far smaller than the number of 128-bit keys.)

Although Rijndael is now the official standard for NIST, it has not received the scrutiny of DES since it is relatively young. Although widely thought to be secure, there will undoubtedly be many interesting aspects of its operation discovered in the coming years.

2.2.3 CBC Encryption

The block ciphers we described above are useful for transforming a plaintext block $P \in \{0,1\}^n$ to a ciphertext $C \in \{0,1\}^n$ under a key $K \in \{0,1\}^k$, but we are left with several problems. Firstly, this transformation does not meet the requirements for any of the standard notions of encryption; a block cipher is sufficient for *enciphering* but not *encrypting*. Secondly, message lengths might be something other than n bits, and we therefore need a solution which handles messages of any bit-length. There are several ways of remedying these problems; we now examine one such method.

Take some block cipher, like AES, which accepts 128-bit blocks and a 128-bit key. Suppose Alice and Bob share a 128-bit key K and Alice has a message M whose length is a multiple of 128 bits in length which she wishes to send to Bob. Alice first writes M as a sequence of j 128-bit blocks $M_1 M_2 \cdots M_j$. Next, Alice generates some uniform random 128-bit string called the Initialization Vector (**IV**). Now she sets $C_0 = IV$ and computes $C_i = AES(K, M_i \oplus C_{i-1})$ for each $1 \leq i \leq j$. Finally, she transmits the ciphertext $C = C_1 C_2 \cdots C_j$ and the IV to Bob. Bob uses the IV and the key K to recover M from C. This commonly-used scheme is called "CBC Mode Encryption with Random IV," where **CBC** stands for "Cipher Block Chaining." It can be proven that, in our model, this scheme is "secure" provided that AES is "secure." Note that our encryption scheme now handles only messages which are a multiple of the block size; this deficiency can be easily remedied by padding the message (adding some additional bits in a well-defined way).

2.2.4 Notions of Security for Symmetric-Key Cryptography

In the previous section we used the term "secure" without a clear idea of what it means. There are several standard attack models within the symmetric-key model given above:

- Ciphertext-Only Attack Model
- Known-Plaintext Attack Model
- Chosen-Plaintext Attack Model
- Chosen-Ciphertext Attack Model

In the Ciphertext-Only model, the attacker (also known as the **adversary**) sees some amount of ciphertext, but does not know the corresponding plaintext. This is the attack model typically used against a class of ciphers called "stream ciphers." In the Known-Plaintext model, the adversary is given some number of plaintexts along with their corresponding ciphertexts (all under the same key). In the Chosen-Plaintext

model the adversary is allowed to *select* plaintexts of her choosing and is then given the corresponding ciphertext. Finally, the Chosen-Ciphertext model allows the adversary to both encrypt *and* decrypt messages of her choice. Obviously, the list above gives models in order of increasing adversarial power.

The adversary is given the corresponding powers according to the model above. It must then be asked, "what is her goal?" That is, what must she accomplish in order to have "broken" the encryption scheme? This varies according to the attack model, and even within each model there are several possible goals; consider the following example.

Consider CBC Mode Encryption with Random IV in the Chosen-Plaintext Attack Model. Consider the following "game" which the adversary is asked to play: we supply the adversary two black boxes which are outwardly indistinguishable. Each box accepts messages M of any length provided it is a multiple of 128. Each box has an identical copy of a randomly-generated k-bit key K. When Box #1 is given a message M, it generates a random IV and outputs the CBC Encryption of M under this IV using block cipher E under key K. When Box #2 is given a message M, it discards M and replaces it with a *random* string M' of the same length as M. It then generates a random IV and outputs the CBC Encryption of M' under this IV using block cipher E under key K.

The adversary is given both boxes, but not told which is Box #1 or Box #2. She is allocated some amount of computational resources. Then the adversary encrypts messages of her choice (thus using the Chosen-Plaintext Attack Model as claimed), up to some maximum number of messages allowed, and her goal is to announce which is Box #1 and which is Box #2. If she does this with probability significantly better than 1/2, she has succeeded in "breaking" the encryption scheme. In good encryption schemes, such as the CBC Mode described here, the probability of her breaking the scheme is very closely related to her ability to break the underlying block cipher.

The above definition of secure encryption immediately implies that any encryption system must use randomness in order be secure. For example, any system that encrypts a message M twice under the same K and always outputs the same ciphertext C both times, cannot be secure since the adversary trivially wins the above game.

Though we do not describe the adversarial goals for the Chosen-Ciphertext Attack Model, suffice to say that our scheme is *not* secure in this setting.

2.2.5 Other Modes of Operation for Symmetric-Key Encryption

CBC Mode is by no means the only method for secure symmetric-key encryption. Counter Mode Encryption with Random IV (**CTR** Mode), selects a random IV, and then invokes the underlying block cipher E with inputs $E(K, IV)$, $E(K, IV + 1)$, $E(K, IV + 2)$, $\cdots$. The outputs are used as a one-time pad to encrypt a message M. One nice advantage CTR Mode enjoys over CBC Mode is that it is highly-parallelizable: if parallelism is available, it can be well-exploited since we can compute the block-cipher invocations at the same time. CBC Mode, on the other, requires left-to-right processing for encryption and therefore cannot be parallelized to any significant degree; CBC Mode decryption is, however, fully parallelizable.

Other standard modes for encryption are the Electronic CodeBook mode (ECB), the Output FeedBack mode (OFB), and the Cipher FeedBack mode (CFB). The security of all of these modes, except for ECB, can be rigorously demonstrated.

2.3 Cryptographic Hash Functions

Before studying Public-Key Cryptography in the next section, it is interesting to introduce the notion of a **cryptographic hash function**.

A hash function is a function $h : \{0,1\}^* \to \{0,1\}^c$ where $\{0,1\}^*$ means "strings of any bit-length" and c is some fixed constant, defined as part of the scheme. Note that there is *no* mention of a key here: these hash functions are keyless. A cryptographic hash function is a hash function with two properties:

- **Collision Resistance:** It is "infeasible" to find distinct bit-strings M_1 and M_2 such that $h(M_1) = h(M_2)$. (Two such strings are said to "collide" under h.)
- **Onewayness:** Given an output value y, it is "infeasible" to find an input value m such that $h(m) = y$.

The above discussion does not constitute a definition, and indeed it does not seem possible to give a proper definition of these objects in any meaningful way. Cryptographers have gotten around this difficulty by defining a model where all parties have access to a "random oracle" (a publically-available random function) and then replaced this oracle with a cryptographic hash function. Despite this difficulty, the above does capture the intuition we seek, and cryptographic hash functions have myriad uses as a primitive in various cryptographic protocols.

Although cryptographic hash functions are most-often considered a primitive, they can be constructed from block ciphers. There are several constructions known to be secure; we examine one of them called the MMO scheme after its inventors (Matyas, Meyer, and Oseas). Assume a block cipher E with block size n-bits and key length $k = n$ bits. Given any message M whose length is a multiple of n, break M into j bit-strings $M_1 M_2 \cdots M_j$ each with length n. Set $h_0 = 0$ and compute $h_i = E(h_{i-1}, M_i)$ for each $1 \leq i \leq j$. Set $h(M) = h_j$.

It is believed that the MMO scheme has the two properties asserted above when used with a secure block cipher such as AES. However, this scheme is significantly slower in practice than other methods which were designed specifically as primitives rather than as schemes which utilitize primitives. The most well-known and often-used algorithm is called the Secure Hash Algorithm (**SHA-1**). SHA-1 accepts messages of any bit-length and produces a 160-bit output. The best-known attack on SHA-1 (in an attempt to find a collision, as defined above) requires an expected 2^{80} invocations of SHA-1, which is most-likely infeasible into the distant future.

2.4 Public-Key Cryptography

As mentioned above, public-key cryptography uses the asymmetric-key model which does *not* assume that each party already possesses a copy of some random key. Instead the parties must establish the key in plain view of the eavesdropper. This is quite a task: how is one to design such a scheme? Indeed, for many years it was thought to be impossible to do so; only in recent times (1970's) did cryptographers begin to propose that it could in fact be done. Witfield Diffie and Martin Hellman, along with ideas independently proposed by Ralph Merkle, proposed systems which achieved exactly this aim. A more versatile system, **RSA**, was quickly discovered soon thereafter by Ron Rivest, Adi Shamir, and Len Adleman. (The British intelligence agency has since announced that it knew of the RSA system before it was published in the open literature.)

We did not explore the internal workings of the block ciphers discussed in the previous section. They are invariably written as "confusion/diffusion" primitives meaning that their internals attempt to "mix" together the input bits and key bits in highly inter-dependent ways so that the output blocks "look random" for any input and any key. This is the way that difficulty is designed into block ciphers. Public-key systems, however, tend to use different types of difficult problems to achieve their goal. They draw from hard problems in number theory, group theory, and various other algebraic domains.

2.4.1 A Hard Problem from Number Theory: Factoring

Number theory is a fertile area for finding hard problems. Perhaps the simplest to state is the **factoring problem.** The idea is this: while it is a relatively simple procedure to multiply two distinct prime numbers p and q (a prime number is an integer greater than 1 which is divisible only by itself and 1) together to produce their product n, it may not be simple to recover p and q given just n. In fact, it seems that the larger p and q are, the more difficult it is to find them when presented only with their product. The best known methods require time proportional to $e^{c\sqrt{\ln n \ln \ln n}}$ for the number n, where c is a constant. (Actually there is now a class of algorithms which does slightly better than this.) This is prohibitively expensive for large values of n; for example, if we generate two random distinct 1000-bit primes p and q and then give the product $n = pq$ to the best known factoring algorithm, the algorithm would not, in all likelihood, find p and q within our lifetimes.

Many public-key techniques are based on the presumed difficulty of the factoring problem. And just saying that the best known algorithms take too long does *not* mean that better algorithms do not exist. Indeed, no one has been able to prove a superpolynomial lower bound on factoring, so it is entirely possible that there does exist an efficient algorithm for factoring despite our inability to discover one. Of course, the

discovery of such an algorithm would quickly render ineffective most of the public-key cryptosystems in use today.

2.4.2 Quantum Computers

Quantum computers (machines which use the properties of quantum mechanics to create computational and informational computing systems) have few known applications at present. It seems that they are far more difficult to design and engineer than conventional computers, and indeed none yet exists outside of research labs, (and even these machines are not much use yet, beyond the uses of researchers). But one very important advantage they enjoy over conventional computers is the ability to solve the factoring problem efficiently. This means that when they do exist in an affordable form with enough speed to factor large numbers, they will require public-key cryptosystems to cease using the factoring problem as their underlying hardness assumption. (Such cryptosystems already do exist, it should be noted.) Many experts believe that quantum computers *will* reach this stage, though most likely not for the next 10 or 20 years.

2.4.3 The RSA Primitive

Perhaps the most important and well-known public-key primitive is the **RSA** primitive (named after its inventors, Rivest, Shamir, and Adleman). It works as follows: suppose Alice wishes to send a message M to Bob, but Alice and Bob do *not* share any secret key. Bob does the following:

- He randomly chooses two large distinct primes p and q.
- He sets n equal to their product pq.
- He chooses some odd number e which shares no prime factors with $\phi = (p-1)(q-1)$.
- He computes the unique number d such that $ed = 1 \bmod \phi$.
- He sends (n, e) to Bob, in plain view of the adversary, but hides d, p, q, and ϕ, and keeps them private.

In the description above, we call (n, e) the **public key** of Bob; (n, d) is called the **private key** of Bob. Now Alice receives Bob's public key (n, e) provided no adversary intercepts it on the way to Alice (this is an entirely different problem, but an important one to be addressed later). Alice computes $C = M^e \bmod n$ and sends C to Bob. Finally, Bob computes $C^d \bmod n$ which recovers M. (Proving this requires some grounding in elementary number theory which is beyond the scope of this article.)

This description is quite simplistic, and several assumptions have been made here. First, M has been treated as an integer here whereas we previously treated it as a bit-string. This is a minor issue, however, since it is a simple matter to convert bit-strings into integers provided the bit-strings are sufficiently short. The integer M must lie in the range $0 \leq M < n$ for the above scheme to work properly. We see once again that, like block cipher primitives, the RSA primitive is useful as a building block but does not immediately handle messages of any bit-length.

Although the RSA primitive will properly recover M as described above, provided M lies in the proper range, it is also possible that M is so small that M^e does not exceed n and therefore the modulus is never used in the computation. In this case, it is quite easy to recover M without knowledge of the secret numbers d, p, q, and ϕ. We will shortly see how to remedy this problem.

If the adversary is able to factor n, that is, find p and q, then she will be able to find d and thus recover M given C. It is therefore of paramount importance that n be sufficiently large to prevent attackers from successfully factoring it. The current record for factoring a large number was set in 1999 when researchers successfully factored a 512-bit integer using an algorithm called the General Number Field Sieve. Using a large number of computers in parallel, they were able to harness a huge amount of computing power (8000 MIPS-years) over a few months' time. Some have taken this as an indication that 512-bit RSA moduli should be considered insufficient and have recommended using 1024-bit moduli instead, especially if one wishes to retain security into the near future.

It is an extremely compelling question to ask whether factoring n is the *best* attack against RSA. This is currently unknown, though it remains a vigorous area of research.

2.4.4 Public-Key Encryption with RSA

The RSA primitive must be handled *very* carefully in order to produce a secure public-key encryption scheme. RSA has many properties which are exploitable if the primitive is misused, and therefore extreme care must be employed when building protocols on top of it. For example, the primes p and q must not only be of sufficient size, they should also be of similar size (to avoid certain factoring algorithms), and they should be sufficiently far apart (to avoid other factoring algorithms). Some experts recommend they have additional properties (the so-called "strong prime" property). Also they should not be one more than any multiple of 3 if the encryption exponent e is 3. This is only a partial list of rules to be observed.

Unlike the simple modes of operation given above, public-key encryption with RSA (or any other public-key primitive) requires a great deal of expertise to implement correctly. We therefore do not treat the topic in depth in this article but refer the reader to more specialized sources listed at the end of the article. The typical methods used for public-key encryption schemes are provably-secure in the Chosen-Ciphertext Attack Model, a stronger model than was used in the symmetric-key setting.

2.5 Integrating Private-Key and Public-Key Cryptosystems

Public-Key primitives are invariably much slower than block ciphers, but are more flexible in that no shared key is required. To exploit the best parts of this trade-off in practice, it is common to use a public-key scheme to select and encrypt (under the appropriate public key) a block cipher key K. Once this is accomplished, the parties then use a symmetric-key encryption scheme to communicate further. This method is used commonly in several network protocols, for example Transport Layer Security (**TLS**, also known as Secure-Socket Layer or **SSL**), the protocol used for secure web transactions.

Here, in brief, is how TLS works: consider a web server named "Alice" and a client named "Bob." The parties wish to establish a secure connection so that Bob can send his credit-card number to Alice without fear of its being stolen. The protocol proceeds as follows:

1. Bob acquires Alice's public key
2. Bob selects a random key K for use with some symmetric-key algorithm such as CBC Mode with Random IV
3. Bob encrypts K using Alice's public key yielding ciphertext C
4. Bob sends C to Alice, who decrypts C using her secret key
5. Bob and Alice now share K and can communicate efficiently using a secure symmetric-key algorithm

The above protocol is over-simplified; the actual TLS algorithm provides public-key certification and message authentication, each of which will be explained shortly. But the main point to appreciate here is that the public-key scheme is used only to distribute another key (often called a **session key**) for later use with a symmetric-key algorithm.

3 Authentication

An equally-important cryptographic problem is **authentication**. Here Alice wishes to send a message to Bob in such a way that Bob can be virtually certain that Alice sent the message. As with the privacy problem, we have two models: the symmetric-key model and the asymmetric-key model. In the symmetric-key model the algorithms are called Message Authentication Codes (or **MAC**s), and in the asymmetric-key model the algorithms are called **Digital Signatures**. Both MACs and digital signatures are widely-used in virtually every secure communications protocol yet invented. In fact, many experts believe that encryption without authentication achieves at best marginal communications security.

3.1 Message Authentication Codes

Message Authentication Codes (MACs) work as follows: Alice and Bob share some random secret key K. Alice wishes to send message M to Bob in such a way that Bob can be virtually certain that M was actually from Alice. Alice computes a function $\sigma = \text{MAC}(K, M)$ where MAC is a function from $\{0,1\}^*$ to $\{0,1\}^t$. We call σ the **tag** and t is the tag length. Alice now sends M and σ to Bob; note that no mention is made of encryption here: authentication is a completely different goal, and it is quite reasonable that Alice cares nothing about privacy in this setting. In fact, our current example has Alice sending M in the clear!

Bob receives M' and σ' and wishes to verify that Alice was the originator. It could be that M or σ were modified in transit, and therefore M' and σ' are used to indicate that what Bob receives may not be what Alice sent. Although the verification step can be different from the MAC-generation step that Alice used, it most often is the same: Bob re-computes the MAC tag and checks for a match. That is, he checks if $\sigma' = \text{MAC}(K, M')$. If yes, Bob accepts M' as valid; if no, Bob rejects the message as an attempted forgery.

3.1.1 The Model

The attack model most-often used for MACs is called the Adaptive Chosen Message Attack Model (**ACMA**). In this model, we once again provide the adversary with an oracle. This oracle has embedded within it a randomly-chosen key K which is inaccessible to the adversary. The adversary is given some amount of computational resources and is allowed to make some number of queries to the oracle. The adversary's goal is to make queries $m_1, m_2, \cdots, m_q$ receiving back MAC tags $\sigma_1, \sigma_2, \cdots, \sigma_q$ and then produce a *new* message m_* different from every previously queried message along with a valid tag σ^*. If she succeeds, she is said to have "forged." A good MAC algorithm denies even the best adversary much of a chance at success.

3.1.2 MAC Algorithms

There are several well-known MAC algorithms. One of the most-commonly used methods is called HMAC. HMAC employs a cryptographic hash function (Section 2.3) to do virtually all of the cryptographic work, and therefore is quite simple to describe. Let h be our cryptographic hash function, such as SHA-1, let K be the MAC key, and let M be the message to MAC. Then we write HMAC as

$$\text{HMAC}_K(M) = h(K \parallel p_1 \parallel h(K \parallel p_2 \parallel M))$$

where p_1 and p_2 are repeated constants used to pad the inputs to h to the proper size, and $\parallel$ denotes string concatenation. The security of HMAC is directly related to the security of the underlying hash function.

Another common MAC algorithm is the CBC MAC. Given a block cipher E with block size n and key K, we MAC a message M (whose length is a multiple of n) by computing the CBC Mode Encryption with IV=0 and then use the final block as the MAC tag (Section 2.2.3). It is important to note that the CBC MAC is secure only when the message length is fixed for any given key; that is, for any key K we must fix a constant ℓ such that every message to be MACed has length ℓn, no more and no less. Although this is certainly an inconvenience in practice, there are several well-known ways to extend the CBC MAC to accept messages of varying lengths.

3.1.3 High-Performance MAC Algorithms

MACs are needed in virtually all secure communications protocols, including those used on the Internet. High volume servers might be required to process extremely large packet rates while, at the same time, producing MACs on the packet contents. Therefore several researchers have sought very efficient MAC algorithms. In software the best MACs are the so-called Carter-Wegman MACs which employ a novel idea: rather than processing the input message M with a cryptographic algorithm (as both HMAC and CBC MAC do), use a non-cryptographic hash function to shrink M to short string, then use a cryptographic primitive on this resulting short string. The advantage is that the non-cryptographic hash function can be very simple and very efficient, more efficient than cryptographic primitives tend to be.

3.2 Integrated Privacy and Authentication in the Symmetric-Key Model

Symmetric-key encryption and MACs often are done together, so it is natural to ask whether algorithms could be devised which yield both encryption and authentication simultaneously with performance better than the sum of the two tasks done separately. Recently several researchers have been exploring this domain resulting in several new schemes to achieve this objective. It is likely that in the coming years one or more of these primitives will be widely-used in high-performance secure communications systems.

3.3 Digital Signatures

A digital signature scheme provides authentication in the asymmetric-key model. The setting is completely analogous to the above setting for MACs except here Alice and Bob do not have a shared key. Alice would like to send a message M to Bob such that Bob can be virtually certain that Alice is the originator of M. Therefore, as in the case for public-key encryption (Section 2.4), a public-key and a private-key are generated. However this time it is *Alice* who generates these keys, and she designates the private-key as her "signing key" and the public-key as her "verification key." Once again she sends her verification key to Bob but keeps her signing key secret. To sign a message M she computes some signature function using her signing key to yield a signature σ and sends M and σ to Bob. Bob receives M' and σ' and computes some verification algorithm using the verfication key of Alice; the verification algorithm outputs VALID or INVALID.

Since public-key primitives are typically quite slow, it is inefficient to actually compute the signature on M as just described. More common is to first apply a cryptographic hash function (Section 2.3) to M and then generate a signature on the hash output. This is provably just as good as signing M itself, assuming the cryptographic hash function is secure.

3.3.1 Signature Schemes

As with public-key encryption (Section 2.4), signature schemes are quite delicate and must be implemented with care. Instead of giving a detailed description of any one scheme, we explore the general framework of one scheme: RSA signatures.

RSA was introduced as a public-key encryption primitive (Section 2.4.3). But the primitive can also be used to generated signatures as follows: Alice generates an RSA public key (e, n) and private key (d, n) as before. To sign a message M, she RSA encrypts M under her private key. In other words, she computes $\sigma = M^d \bmod n$. To verify the signature, Bob decrypts with the public key of Alice; that is, when receiving M' and σ', Bob checks if $\sigma' = M'^e \bmod n$. If yes, he accepts M' as a valid signed message, else he rejects M' as an attempted forgery. Once again, M must be treated as a number in the range $0 \leq M < n$, and, as noted above, M is usually the *hash* of a message rather than the message itself. This description of RSA signatures is quite oversimplified, but the crux idea is that the RSA public-key primitive is quite well-suited for use as a signature primitive as well.

Notice that, because of the manner in which public-key cryptosystems work, a signature of M by Alice can be verified by not only by Bob but by *any* party in possession of Alice's public key. This flexibility is endowed in particular because of the minimal setup requirements in public-key cryptography.

Another well-known and widely-used algorithm is the Digital Signature Algorithm (**DSA**). DSA is a type of **ElGamal Signature** which uses a different type of difficulty called the Discrete Logarithm Problem (**DLP**). The DLP is as follows: we start with a prime p and a generator α (a generator is a number between 1 and $p-1$ such that $p-1$ is the smallest positive integer such that α^{p-1} is one more than a multiple of p). Now it is quite a simple matter given any x between 0 and p to compute the value $\alpha^x \bmod p$. However, if we are instead given a number β with $1 \leq \beta \leq p-1$, it is difficult (in general) to find a power x such that $\alpha^x = \beta \bmod p$. The "onewayness" of the DLP is what is exploited in ElGamal signatures to obtain their security.

3.4 Certificates and Key Distribution

Although the primitives and protocols described in the previous sections all have very interesting properties and are secure in their models, there remains a significant problem alluded to earlier. When describing

TLS (Section 2.5), the first step was for Bob to acquire Alice's public key. The obvious problem here is this: what if, in an attempt to acquire Alice's public key, Bob is fooled into accepting the public key of an impersonator instead? This impersonator would submit his own public key in place of Alice's (and of course he has the corresponding private key in his possession). If Bob then encrypted a random session key under the impersonator's public key, the impersonator would then enter into a communication with Bob where Bob falsely believed he was talking with Alice.

To circumvent this problem, TLS mandates that the public key of Alice be digitally signed by some "trusted" entity. Such an entity is called a Certification Authority (**CA**). The CA is an entity which is in the business of digitally signing the public keys of other entities which wish to have validated public keys. The idea here is that rather than expect Bob to already have a reliable copy of every public key of every possible Alice he might interact with, he instead just has one widely-distributed, widely-known public key from the CA. Then Alice must go to the CA and, after paying a fee, have her public key signed. The resulting TLS protocol looks like this:

1. Bob acquires Alice's public key with an accompanying signature from some CA
2. Bob verifies that Alice's public key is properly signed by the CA
3. Bob selects a random key K for use with some symmetric-key algorithm such as CBC Mode with Random IV; he also selects a key K' for use with some MAC algorithm
4. Bob encrypts K and K' using Alice's public key yielding ciphertext C
5. Bob sends C to Alice, who decrypts C using her secret key
6. Bob and Alice now share keys K and K' and can communicate efficiently using a secure symmetric-key algorithm and a secure MAC algorithm

This is, once again, quite an oversimplification. In practice there is more than one CA, and sometimes we might see signatures on signatures, the so-called "Web of Trust." Key distribution and identity authentication remain fertile grounds of active research.

One final question perhaps remains after reading this section: how does Bob get the well-known public key of the CA when he is, say, perusing the web? The answer is this: the public key of the CA is built in to his web browser, which implements the TLS protocol. So how does Bob ensure that his web browser has not been tampered with? Currently, he doesn't.

4 Other Cryptographic Protocols

As we saw in the introduction, privacy and authentication are the most-important cryptographic problems in some sense. But by no means does this complete the list of problems cryptographers work on. Other very interesting protocols built using these protocols include digital cash, electronic voting, and many, many others. We give an example of one further very useful protocol as an example of a cryptographic solution which has a very different flavor to what we have seen thus far.

4.1 Secret Sharing

Suppose there is some private piece of information which is too sensitive for any one person to possess. (The typical example is launch codes for nuclear missiles.) You may wish to partition this "secret" into pieces and distribute these pieces to various different people in such a way that all these individuals must contribute their pieces to reconstruct the original secret. This scheme can be realized quite easily. Consider an example:

Suppose we wish to share the secret "1131" among five world leaders: Halonen, Putin, Zemin, Toledo, and Mbeki. First we would select some $n > 1131$, say 5000. Then we would choose 4 random numbers from 0 to $n - 1$ assigning one to each of the first 4 members, say

$$\mathsf{Halonen} \leftarrow 1109$$

$$\mathsf{Putin} \leftarrow 199$$

$$\mathsf{Zemin} \leftarrow 4012$$

$$\mathsf{Toledo} \leftarrow 2392$$

and then sum these mod5000 to get 2712, and finally assign $\mathsf{Mbeki} \leftarrow 1131 - 2712 = 3419$. Notice that the five pieces just dealt out sum to the secret, mod 5000. Therefore, if all 5 members of the group agree to recover the shared secret, they would "pool their shares" by adding up their pieces mod 5000. Clearly this can easily be generalized to any number of players and any size secret. Also, it should be obvious that if less than 5 players contribute shares, absolutely no information about the secret is revealed (since the remaining share could be any number from 0 to 4999 and each of these induces a different result mod 5000).

What are the drawbacks to the above scheme? There are several: (1) Requiring *all* players to contribute their shares is often too strong a restriction (what if one of the players loses his share?), (2) What if a player contributes a bad share? Then clearly the result is affected and there is no way to determine who the cheater is! (3) What if the dealer is corrupt and gives out bad shares? Can we allow the players to verify the validity of their shares without giving them any information?

The domain of Secret Sharing has addressed all of these questions. A particularly nice scheme for addressing the first problem above was invented by Shamir in 1979.

4.1.1 Shamir's Threshold Secret-Sharing Scheme

A far more reasonable access structure is called a k-out-of-ℓ or "threshold" scheme. The idea is that the secret is dealt to ℓ players but any k of them can contribute their shares to recover the secret. And of course if any $k-1$ or fewer players contribute their shares, nothing is learned. The first scheme to achieve this behavior was devised by Adi Shamir.

Shamir's scheme is based on the "Fundamental Theorem of Algebra": that a degree $t-1$ univariate polynomial $f(x)$ is uniquely determined by t points (x_i, y_i), $1 \leq i \leq t$ where the x_i are distinct. This theorem is true in any algebraic field, including, in particular, the field of integers modulo some prime p. We denote this field Z_p. Shamir's idea is as follows: suppose we would like to share a secret s with n players. We choose some prime $p > \max(s, n)$ to induce a field Z_p. We set $a_0 = s$ and select $t-1$ random values between 0 and $p-1$; call these $a_1, \cdots a_{t-1}$. Let $f(x) = a_0 + a_1 x + \cdots + a_{t-1}x^{t-1}$ over Z_p. Now for each player i from 1 to n we compute $f(i)$ and hand $(i, f(i))$ to player i securely. Now if any t players pool their shares, they can recover $f(\cdot)$ by a simple technique called LaGrange interpolation and then compute $f(0)$ to recover $a_0 = s$. Computing in Z_p we get

$$s = \sum_{i=1}^{t} c_i y_i, \qquad \text{where } c_j = \prod_{1 \leq j \leq t, j \neq i} \frac{x_j}{x_j - x_i}.$$

And again if $t-1$ or fewer players contribute their shares, nothing is learned. This scheme is quite elegant, and may suffice for many situations, but one can imagine even more flexible access structures or requirements on the scheme, such as providing for the possible corruption of the various players. Secret Sharing remains an area of on-going research.

5 Summary

Cryptography can be described as the domain of deliberate difficulty. Using the hardness of various problems, cryptographers tackle problems such as protecting the privacy of information, authenticating entities on a network, sharing a secret among various parties, electronic voting, digital cash, and much much more. As these algorithms become more widely-used in a variety of settings, it becomes ever more important to ensure the methods used are fast and secure. Much research continues in both the practical and theoretical areas to extend ideas in this direction.