EXHIBIT 10

Let's look at this model in more detail for these different cryptographic primitives.

5.3.1 Random Functions: Hash Functions

The first type of random oracle is the *random function.* A random function accepts an input string of any length, and outputs a random string of fixed length, say n bits long. So the elf just has a simple list of inputs and outputs, which grows steadily as it works. (We'll ignore any effects of the size of the scroll and assume that all queries are answered in constant time.)

Random functions are our model for *one-way functions* or *cryptographic hash functions*, which have many practical uses. They were first used in computer systems for one-way encryption of passwords in the 1960s and—as mentioned in Chapter 2—are used today in a number of authentication systems. They are also used to compute *message digests;* given a message M, we can pass it through a pseudorandom function to get a digest, say $h(M)$, which can stand in for the message in various applications. One example is a digital signature: signature algorithms tend to be slow if the message is long, so it's usually convenient to sign a message digest rather than the message itself.

Another application is timestamping. If we want evidence that we possessed a given electronic document by a certain date, we might submit it to an online timestamping service. However, if the document is still secret—for example an invention that we plan to patent, and for which we merely want to establish a priority date—then we might not send the timestamping service the whole document, but just the message digest.

The output of the hash function is known as the *hash value* or *message digest;* an input corresponding to a given hash value is its *preimage;* the verb *to hash* is used to refer to computation of the hash value. Colloquially, the *hash* is also used as a noun to refer to the hash value.

5.3.1.1 Properties

The first main property of a random function is *one-wayness.* Given knowledge of an input x, we can easily compute the hash value $h(x)$; but it is very difficult given the hash value $h(x)$ to find a corresponding preimage x if one is not already known. (The elf will only pick outputs for given inputs, not the other way round.) As the output is random, the best an attacker who wants to invert a random function can do is to keep on feeding in more inputs until he or she gets lucky. A pseudorandom function will have the same property; or this could be used to distinguish it from a random function, contrary to our definition. It follows that a pseudorandom function will also be a *one-way function*, provided there are enough possible outputs that the opponent can't find a desired target output by chance. This means choosing the output to be an n-bit number where the opponent can't do anything near 2" computations.

A second property of pseudorandom functions is that the output will not give any information at all about even part of the input. Thus, one-way encryption of the value x can be accomplished by concatenating it with a secret key k and computing $h(x, k)$. If the hash function isn't random enough though, using it for one-way encryption in this manner is asking for trouble. A topical example comes from the authentication in GSM mobile phones, where a 16-byte challenge from the base station is concatenated with a 16-byte secret key known to the phone into a 32-byte number, and passed through a hash function to give an 11-byte output [138]. The idea is that the phone company also knows k and can check this computation, while someone who eavesdrops on the radio link can only get a number of values of the random challenge x and corresponding output from $h(x, k)$. The eavesdropper must not be able to get any information about k or be able to compute $h(y, k)$ for a new input y. But the one-way function used by most phone companies isn't one-way enough, with the result that an eavesdropper who can pretend to be a base station and send a phone about 60,000 suitable challenges and get the responses can compute the key. I'll discuss this failure in more detail in Chapter 17, Section 17.3.3.

A third property of pseudorandom functions with sufficiently long outputs is that it is hard to find *collisions*, that is, different messages $M_1 \neq M_2$ with $h(M_1)=h(M_2)$. Unless the opponent can find a shortcut attack (which would mean the function wasn't really pseudorandom), then the best way of finding a collision is to collect a large set of messages M_i and their corresponding hashes $h(M_i)$, sort the hashes, and look for a match. If the hash function output is an n-bit number, so that there are 2^n possible hash values, then the number of hashes the enemy will need to compute before he or she can expect to find a match will be about the square root of this, namely $2^{n/2}$ hashes. This fact is of major importance in security engineering, so let's look at it more closely.

5.3.1.2 The Birthday Theorem

The *birthday theorem*, first known as *capture-recapture statistics*, was invented in the 1930s to count fish [679]. Suppose there are N fish in a lake, and you catch m of them, tag them, and throw them back; then when you first catch a fish you've tagged already, m should be "about" the square root of N. The intuitive reason this holds is that once you have $\sqrt{N}$ samples, each could potentially match any of the others, so the number of possible matches is about $\sqrt{N}\sqrt{N}$ or N, which is what you need.[1]

The birthday theorem has many applications for the security engineer. For example, if we have a biometric system that can authenticate a person's claim to identity with a probability of only one in a million that two randomly selected subjects will be falsely identified as the same person, this doesn't mean that we can use it as a reliable means of identification in a university with a user population of twenty thousand staff and students. This is because there will be almost two hundred million possible pairs. In fact, you can expect to find the first *collision*—the first pair of people who can be mistaken for each other by the system—once you have somewhat over a thousand people enrolled.

In some applications collision search attacks aren't a problem, such as in challenge response protocols where an attacker would have to be able to find the answer to the challenge just issued, and where you can prevent challenges repeating. (For example, the challenge might not be really random but generated by encrypting a counter.) In identify-friend-or-foe (IFF) systems, for example, common equipment has a response length of 48 to 80 bits.

[1] More precisely, the probability that m fish chosen randomly from N fish are different is $\beta = N(N - 1) \cdots (N - m + 1)/N^m$ which is asymptotically solved by $N \approx m^2/2 \log(1/\beta)$[451].

However, there are other applications in which collisions are unacceptable. In a digital signature application, if it were possible to find collisions with $h(M_1) = h(M_2)$ but $M_1 \neq M_2$, then a Mafia-owned bookstore's Web site might get you to sign a message M_1 saying something like, "I hereby order a copy of Rubber Fetish volume 7 for \$32.95," and then present the signature together with an M_2, saying something like, "I hereby mortgage my house for \$75,000; and please make the funds payable to Mafia Holdings Inc., Bermuda."

For this reason, hash functions used with digital signature schemes generally have n large enough to make them collision-free, that is, that $2^{n/2}$ computations are impractical for an opponent. The two most common are MD5, which has a 128-bit output and will thus require about 2^{64} computations to break, and SHA1 with a 160-bit output and a work factor for the cryptanalyst of about 2^{80}. MD5, at least, is starting to look vulnerable: already in 1994, a design was published for a \$10 million machine that would find collisions in 24 days, and SHA1 will also be vulnerable in time. So the U.S. National Institute of Standards and Technology (NIST) has recently introduced still wider hash functions—SHA256 with a 256-bit output, and SHA512 with 512 bits. In the absence of cryptanalytic *shortcut attacks*—that is, attacks requiring less computation than brute force search—these should require 2^{128} and 2^{256} effort respectively to find a collision. This should keep Moore's Law at bay for a generation or two. In general, a prudent designer will use a longer hash function where this is possible, and the use of the MD series hash functions in new systems should be avoided (MD5 had a predecessor MD4 which turned out to be cryptanalytically weak, with collisions and preimages being found).

Thus, a pseudorandom function is also often referred to as being *collision-free* or *collision-intractable*. This doesn't mean that collisions don't exist—they must, as the set of possible inputs is larger than the set of possible outputs—just that you will never find any of them. The (usually unstated) assumption is that the output must be long enough.

5.3.2 Random Generators: Stream Ciphers

The second basic cryptographic primitive is the *random generator*, also known as a *keystream generator* or *stream cipher*. This is also a random function, but unlike in the hash function case it has a short input and a long output. (If we had a good pseudorandom function whose input and output were a billion bits long, and we never wanted to handle any objects larger than this, we could turn it into a hash function by throwing away all but a few hundred bits of the output, and a stream cipher by padding all but a few hundred bits of the input with a constant.) At the conceptual level, however, it's common to think of a stream cipher as a random oracle whose input length is fixed while the output is a very long stream of bits, known as the *keystream*. It can be used quite simply to protect the confidentiality of backup data: we go to the keystream generator, enter a key, get a long file of random bits, and exclusive-or it with our plaintext data to get ciphertext, which we then send to our backup contractor. We can think of the elf generating a random tape of the required length each time he is presented with a new key as input, giving it to us and keeping a copy of it on his scroll for reference in case he's given the same input again. If we need to recover the data, we go back to the generator, enter the same key, get the same long file of random data, and exclusive-or it with our ciphertext to get our plaintext data back again. Other people with access to the keystream generator won't be able to generate the same keystream unless they know the key.

I mentioned the one-time pad, and Shannon's result that a cipher has perfect secrecy if and only if there are as many possible keys as possible plaintexts, and every key is equally likely. Such security is called *unconditional* (or *statistical*) security, as it doesn't depend either on the computing power available to the opponent or on there being no future advances in mathematics that provide a shortcut attack on the cipher.

One-time pad systems are a very close fit for our theoretical model, except that they are typically used to secure communications across space rather than time: there are two communicating parties who have shared a copy of the randomly generated keystream in advance. Vernam's original telegraph cipher machine used punched paper tape; of which two copies were made in advance, one for the sender and one for the receiver. A modern diplomatic system might use optical tape, shipped in a tamper-evident container in a diplomatic bag. Various techniques have been used to do the random generation. Marks describes how SOE agents' silken keys were manufactured in Oxford by little old ladies shuffling counters.

One important problem with keystream generators is that we want to prevent the same keystream being used more than once, whether to encrypt more than one backup tape or to encrypt more than one message sent on a communications channel. During World War II, the amount of Russian diplomatic traffic exceeded the quantity of one-time tape they had distributed in advance to their embassies, so it was reused. This was a serious blunder. If $M_1 + K = C_1$, and $M_2 + K = C_2$, then the opponent can combine the two ciphertexts to get a combination of two messages: $C_1 - C_2 = M_1 - M_2$; and if the messages M_i have enough redundancy, then they can be recovered. Text messages do in fact contain enough redundancy for much to be recovered; and in the case of the Russian traffic, this led to the Venona project in which the United States and United Kingdom decrypted large amounts of wartime Russian traffic afterward and broke up a number of Russian spy rings. The saying is: "Avoid the two-time tape!"

Exactly the same consideration holds for any stream cipher, and the normal engineering practice when using an algorithmic keystream generator is to have a *seed* as well as a key. Each time the cipher is used, we want it to generate a different keystream, so the key supplied to the cipher should be different. So, if the long-term key that two users share is K, they may concatenate it with a seed that is a message number N (or some other nonce), then pass it through a hash function to form a working key $h(K, N)$. This working key is the one actually fed to the cipher machine.

5.3.3 Random Permutations: Block Ciphers

The third type of primitive, and the most important in modern commercial cryptography, is the *block cipher*, which we model as a *random permutation*. Here, the function is invertible, and the input plaintext and the output ciphertext are of a fixed size. With Playfair, both input and output are two characters; with DES, they're both bit strings of 64 bits. Whatever the number of symbols and the underlying alphabet, encryption acts on a block of fixed length. (If you want to encrypt a shorter input, you have to pad it, as with the final z in our Playfair example.)