CHAPTER 4

Intermediate Protocols

4.1 TIMESTAMPING SERVICES

In many situations, people need to certify that a document existed on a certain date. Think about a copyright or patent dispute: The party that produces the earliest copy of the disputed work wins the case. With paper documents, notaries can sign and lawyers can safeguard copies. If a dispute arises, the notary or the lawyer testifies that the letter existed on a certain date.

In the digital world, it's far more complicated. There is no way to examine a digital document for signs of tampering. It can be copied and modified endlessly without anyone being the wiser. It's trivial to change the date stamp on a computer file. No one can look at a digital document and say: "Yes, this document was created before November 4, 1952."

Stuart Haber and W. Scott Stornetta at Bellcore thought about the problem [682, 683,92]. They wanted a digital timestamping protocol with the following properties:

— The data itself must be timestamped, without any regard to the physical medium on which it resides.
— It must be impossible to change a single bit of the document without that change being apparent.
— It must be impossible to timestamp a document with a date and time different from the present one.

Arbitrated Solution

This protocol uses Trent, who has a trusted timestamping service, and Alice, who wishes to timestamp a document.

(1) Alice transmits a copy of the document to Trent.

(2) Trent records the date and time he received the document and retains a copy of the document for safekeeping.

Now, if anyone calls into question Alice's claim of when the document was created, she just has to call up Trent. He will produce his copy of the document and verify that he received the document on the date and time stamped.

This protocol works, but has some obvious problems. First, there is no privacy. Alice has to give a copy of the document to Trent. Anyone listening in on the communications channel could read it. She could encrypt it, but still the document has to sit in Trent's database. Who knows how secure that database is?

Second, the database itself would have to be huge. And the bandwidth requirements to send large documents to Trent would be unwieldy.

The third problem has to do with the potential errors. An error in transmission, or an electromagnetic bomb detonating somewhere in Trent's central computers, could completely invalidate Alice's claim of a timestamp.

And fourth, there might not be someone as honest as Trent to run the timestamping service. Maybe Alice is using Bob's Timestamp and Taco Stand. There is nothing to stop Alice and Bob from colluding and timestamping a document with any time that they want.

Improved Arbitrated Solution

One-way hash functions and digital signatures can clear up most of these problems easily:

(1) Alice produces a one-way hash of the document.
(2) Alice transmits the hash to Trent.
(3) Trent appends the date and time he received the hash onto the hash and then digitally signs the result.
(4) Trent sends the signed hash with timestamp back to Alice.

This solves every problem but the last. Alice no longer has to worry about revealing the contents of her document; the hash is sufficient. Trent no longer has to store copies of the document (or even of the hash), so the massive storage requirements and security problems are solved (remember, one-way hash functions don't have a key). Alice can immediately examine the signed timestamped hash she receives in step (4), so she will immediately catch any transmission errors. The only problem remaining is that Alice and Trent can still collude to produce any timestamp they want.

Linking Protocol

One way to solve this problem is to link Alice's timestamp with timestamps previously generated by Trent. These timestamps will most probably be generated for people other than Alice. Since the order that Trent receives the different timestamp requests can't be known in advance, Alice's timestamp must have occurred after the previous one. And since the request that came after is linked with Alice's timestamp, then hers must have occurred before. This sandwiches Alice's request in time.

If A is Alice's name, the hash value that Alice wants timestamped is H_n, and the previous time stamp is T_{n-1}, then the protocol is:

(1) Alice sends Trent H_n and A.

(2) Trent sends back to Alice:

$$T_n = S_K(n, A, H_n, t_n, I_{n-1}, H_{n-1}, T_{n-1}, L_n)$$

where L_n consists of the following hashed linking information:

$$L_n = H(I_{n-1}, H_{n-1}, T_{n-1}, L_{n-1})$$

S_K indicates that the message is signed with Trent's private key. Alice's name identifies her as the originator of the request. The parameter n indicates the sequence of the request: This is the nth timestamp Trent has issued. The parameter t_n is the time. The additional information is the identification, original hash, time, and hashed timestamp of the previous document Trent stamped.

(3) After Trent stamps the next document, he sends Alice the identification of the originator of that document: I_{n+1}.

If someone challenges Alice's timestamp, she just contacts the originators of the previous and following documents: I_{n-1} and I_{n+1}. If their documents are called into question, they can get in touch with I_{n-2} and I_{n+2}, and so on. Every person can show that their document was timestamped after the one that came before and before the one that came after.

This protocol makes it very difficult for Alice and Trent to collude and produce a document stamped with a different time than the actual one. Trent cannot forward-date a document for Alice, since that would require knowing in advance what document request came before it. Even if he could fake that, he would have to know what document request came before that, and so on. He cannot back-date a document, because the timestamp must be embedded in the timestamps of the document issued immediately after, and that document has already been issued. The only possible way to break this scheme is to invent a fictitious chain of documents both before and after Alice's document, long enough to exhaust the patience of anyone challenging the timestamp.

Distributed Protocol

People die; timestamps get lost. Many things could happen between the timestamping and the challenge to make it impossible for Alice to get a copy of I_{n-1}'s timestamp. This problem could be alleviated by embedding the previous 10 people's timestamps into Alice's, and then sending Alice the identities of the next 10 people. Alice has a greater chance of finding people who still have their timestamps.

Along a similar line, the following protocol does away with Trent altogether.

(1) Using H_n as input, Alice generates a string of random values using a cryptographically secure pseudo-random-number generator:

$$V_1, V_2, V_3, \ldots V_k$$

(2) Alice interprets each of these values as the identification, I, of another person. She sends H_n to each of these people.

(3) Each of these people attaches the date and time to the hash, signs the result, and sends it back to Alice.

(4) Alice collects and stores all the signatures as the timestamp.

The cryptographically secure pseudo-random-number generator in step (1) prevents Alice from deliberately choosing corrupt Is as verifiers. Even if she makes trivial changes in her document in an attempt to construct a set of corrupt Is, her chances of getting away with this are negligible. The hash function randomizes the Is; Alice cannot force them.

This protocol works because the only way for Alice to fake a timestamp would be to convince all of the k people to cooperate. Since she chose them at random in step (1), the odds against this are very high. The more corrupt society is, the higher a number k should be.

Additionally, there should be some mechanism for dealing with people who can't promptly return the timestamp. Some subset of k is all that would be required for a valid timestamp. The details depend on the implementation.

Further Work

Further improvements to timestamping protocols are presented in [92]. The authors use binary trees to increase the number of timestamps that depend on a given timestamp, reducing even further the possibility that someone could create a chain of fictitious timestamps. They also recommend publishing a hash of the day's timestamps in a public place, such as a newspaper. This serves a function similar to sending the hash to random people in the distributed protocol. In fact, a timestamp has appeared in every Sunday's *New York Times* since 1992.

These timestamping protocols are patented [684,685,686]. A Bellcore spin-off company called Surety Technologies owns the patents and markets a Digital Notary System to support these protocols. In their first version, clients send "certify" requests to a central coordinating server. Following Merkle's technique of using hash functions to build trees [1066], the server builds a tree of hash values whose leaves are all the requests received during a given second, and sends back to each requester the list of hash values hanging off the path from its leaf to the root of the tree. The client software stores this locally, and can issue a Digital Notary "certificate" for any file that has been certified. The sequence of roots of these trees comprises the "Universal Validation Record" that will be available electronically at multiple repository sites (and also published on CD-ROM). The client software also includes a "validate" function, allowing the user to test whether a file has been certified in exactly its current form (by querying a repository for the appropriate tree root and comparing it against a hash value appropriately recomputed from the file and its certificate). For information contact Surety Technologies, 1 Main St., Chatham, NJ, 07928; (201) 701-0600; Fax: (201) 701-0601.

4.2 SUBLIMINAL CHANNEL

Alice and Bob have been arrested and are going to prison. He's going to the men's prison and she's going to the women's prison. Walter, the warden, is willing to let Alice and Bob exchange messages, but he won't allow them to be encrypted. Walter expects them to coordinate an escape plan, so he wants to be able to read everything they say.

Walter also hopes to deceive either Alice or Bob. He wants one of them to accept a fraudulent message as a genuine message from the other. Alice and Bob go along with this risk of deception, otherwise they cannot communicate at all, and they have to coordinate their plans. To do this they have to deceive the warden and find a way of communicating secretly. They have to set up a **subliminal channel**, a covert communications channel between them in full view of Walter, even though the messages themselves contain no secret information. Through the exchange of perfectly innocuous signed messages they will pass secret information back and forth and fool Walter, even though Walter is watching all the communications.

An easy subliminal channel might be the number of words in a sentence. An odd number of words in a sentence might correspond to "1," while an even number of words might correspond to "0." So, while you read this seemingly innocent paragraph, I have sent my operatives in the field the message "101." The problem with this technique is that it is mere steganography (see Section 1.2); there is no key and security depends on the secrecy of the algorithm.

Gustavus Simmons invented the concept of a subliminal channel in a conventional digital signature algorithm [1458,1473]. Since the subliminal messages are hidden in what looks like normal digital signatures, this is a form of obfuscation. Walter sees signed innocuous messages pass back and forth, but he completely misses the information being sent over the subliminal channel. In fact, the subliminal-channel signature algorithm is indistinguishable from a normal signature algorithm, at least to Walter. Walter not only cannot read the subliminal message, but he also has no idea that one is even present.

In general the protocol looks like this:

(1) Alice generates an innocuous message, pretty much at random.

(2) Using a secret key shared with Bob, Alice signs the innocuous message in such a way that she hides her subliminal message in the signature. (This is the meat of the subliminal channel protocol; see Section 23.3.)

(3) Alice sends this signed message to Bob via Walter.

(4) Walter reads the innocuous message and checks the signature. Finding nothing amiss, he passes the signed message to Bob.

(5) Bob checks the signature on the innocuous message, confirming that the message came from Alice.

(6) Bob ignores the innocuous message and, using the secret key he shares with Alice, extracts the subliminal message.

What about cheating? Walter doesn't trust anyone and no one trusts him. He can always prevent communication, but he has no way of introducing phony messages. Since he can't generate any valid signatures, Bob will detect his attempt in step (5). And since he does not know the shared key, he can't read the subliminal messages. Even more important, he has no idea that the subliminal messages are there. Signed messages using a digital signature algorithm look no different from signed messages with subliminal messages embedded in the signature.

Cheating between Alice and Bob is more problematic. In some implementations of a subliminal channel, the secret information Bob needs to read the subliminal message is the same information Alice needs to sign the innocuous message. If this is the case, Bob can impersonate Alice. He can sign messages purporting to come from her, and there is nothing Alice can do about it. If she is to send him subliminal messages, she has to trust him not to abuse her private key.

Other subliminal channel implementations don't have this problem. A secret key shared by Alice and Bob allows Alice to send Bob subliminal messages, but it is not the same as Alice's private key and does not allow Bob to sign messages. Alice need not trust Bob not to abuse her private key.

Applications of Subliminal Channel

The most obvious application of the subliminal channel is in a spy network. If everyone sends and receives signed messages, spies will not be noticed sending subliminal messages in signed documents. Of course, the enemy's spies can do the same thing.

Using a subliminal channel, Alice could safely sign a document under threat. She would, when signing the document, imbed the subliminal message, saying, "I am being coerced." Other applications are more subtle. A company can sign documents and embed subliminal messages, allowing them to be tracked throughout the documents' lifespans. The government can "mark" digital cash. A malicious signature program can leak secret information in its signatures. The possibilities are endless.

Subliminal-Free Signatures

Alice and Bob are sending signed messages to each other, negotiating the terms of a contract. They use a digital signature protocol. However, this contract negotiation has been set up as a cover for Alice's and Bob's spying activities. When they use the digital signature algorithm, they don't care about the messages they are signing. They are using a subliminal channel in the signatures to send secret information to each other. The counterespionage service, however, doesn't know that the contract negotiations and the use of signed messages are just cover-ups. This concern has led people to create **subliminal-free signature schemes**. These digital signature schemes cannot be modified to contain a subliminal channel. See [480,481] for details.

4.3 UNDENIABLE DIGITAL SIGNATURES

Normal digital signatures can be copied exactly. Sometimes this property is useful, as in the dissemination of public announcements. Other times it could be a problem. Imagine a digitally signed personal or business letter. If many copies of that document were floating around, each of which could be verified by anyone, this could lead to embarrassment or blackmail. The best solution is a digital signature that can be proven valid, but that the recipient cannot show to a third party without the signer's consent.

The Alice Software Company distributes DEW (Do-Everything-Word). To ensure that their software is virus-free, they include a digital signature with each copy. However, they want only legitimate buyers of the software, not software pirates, to be able to verify the signature. At the same time, if copies of DEW are found to contain a virus, the Alice Software Company should be unable to deny a valid signature.

Undeniable signatures [343,327] are suited to these sorts of tasks. Like a normal digital signature, an undeniable signature depends on the signed document and the signer's private key. But unlike normal digital signatures, an undeniable signature cannot be verified without the signer's consent. Although a better name for these signatures might be something like "nontransferable signatures," the name comes from the fact that if Alice is forced to either acknowledge or deny a signature—perhaps in court—she cannot falsely deny her real signature.

The mathematics are complicated, but the basic idea is simple:

(1) Alice presents Bob with a signature.

(2) Bob generates a random number and sends it to Alice.

(3) Alice does a calculation using the random number and her private key and sends Bob the result. Alice could only do this calculation if the signature is valid.

(4) Bob confirms this.

There is also an additional protocol so that Alice can prove that she did not sign a document, and cannot falsely deny a signature.

Bob can't turn around and convince Carol that Alice's signature is valid, because Carol doesn't know that Bob's numbers are random. He could have easily worked the protocol backwards on paper, without any help from Alice, and then shown Carol the result. Carol can be convinced that Alice's signature is valid only if she completes the protocol with Alice herself. This might not make much sense now, but it will once you see the mathematics in Section 23.4.

This solution isn't perfect. Yvo Desmedt and Moti Yung show that it is possible, in some applications, for Bob to convince Carol that Alice's signature is valid [489].

For instance, Bob buys a legal copy of DEW. He can validate the signature on the software package whenever he wants. Then, Bob convinces Carol that he's a salesman from the Alice Software Company. He sells her a pirated copy of DEW. When Carol tries to validate the signature with Bob, he simultaneously validates the signature with Alice. When Carol sends him the random number, he then sends it on to Alice. When Alice replies, he then sends the reply on to Carol. Carol is convinced that she is a legitimate buyer of the software, even though she isn't. This attack is an instance of the chess grandmaster problem and is discussed in detail in Section 5.2.

Even so, undeniable signatures have a lot of applications; in many instances Alice doesn't want anyone to be able to verify her signature. She might not want personal correspondence to be verifiable by the press, be shown and verified out of context, or even to be verified after things have changed. If she signs a piece of information she sold, she won't want someone who hasn't paid for the information to be able to verify its authenticity. Controlling who verifies her signature is a way for Alice to protect her personal privacy.

A variant of undeniable signatures separates the relation between signer and message from the relation between signer and signature [910]. In one signature scheme, anyone can verify that the signer actually created the signature, but the cooperation of the signer is required to verify that the signature is valid for the message.

A related notion is an **entrusted undeniable signature** [1229]. Imagine that Alice works for Toxins, Inc., and sends incriminating documents to a newspaper using an undeniable signature protocol. Alice can verify her signature to the newspaper reporter, but not to anyone else. However, CEO Bob suspects that Alice is the source of the documents. He demands that Alice run the disavowal protocol to clear her name, and Alice refuses. Bob maintains that the only reason Alice has to refuse is that she is guilty, and fires her.

Entrusted undeniable signatures are like undeniable signatures, except that the disavowal protocol can only be run by Trent. Bob cannot demand that Alice run the disavowal protocol; only Trent can. And if Trent is the court system, then he will only run the protocol to resolve a formal dispute.

4.4 DESIGNATED CONFIRMER SIGNATURES

The Alice Software Company is doing a booming business selling DEW—so good, in fact, that Alice is spending more time verifying undeniable signatures than writing new features.

Alice would like a way to designate one particular person in the company to be in charge of signature verification for the whole company. Alice, or any other programmer, would be able to sign documents with an undeniable protocol. But the verifications would all be handled by Carol.

As it turns out, this is possible with **designated confirmer signatures** [333,1213]. Alice can sign a document such that Bob is convinced the signature is valid, but he cannot convince a third party; at the same time Alice can designate Carol as the future confirmer of her signature. Alice doesn't even need to ask Carol's permission beforehand; she just has to use Carol's public key. And Carol can still verify Alice's signature if Alice is out of town, has left the company, or just upped and died.

Designated confirmer signatures are kind of a compromise between normal digital signatures and undeniable signatures. There are certainly instances where Alice might want to limit who can verify her signature. On the other hand, giving Alice complete control undermines the enforceability of signatures: Alice might refuse to cooperate in either confirming or denying, she might claim the loss of keys for confirming or denying, or she might just be unavailable. Designated confirmer signatures can give Alice the protection of an undeniable signature while not letting her abuse that protection. Alice might even prefer it that way: Designated confirmer signatures can help prevent false applications, protect her if she actually does lose her key, and step in if she is on vacation, in the hospital, or even dead.

This idea has all sorts of possible applications. Carol can set herself up as a notary public. She can publish her public key in some directory somewhere, and people can designate her as a confirmer for their signatures. She can charge a small fee for confirming signatures for the masses and make a nice living.

Carol can be a copyright office, a government agency, or a host of other things. This protocol allows organizations to separate the people who sign documents from the people who help verify signatures.

4.5 Proxy Signatures

Designated confirmer signatures allows a signer to designate someone else to verify his signature. Alice, for instance, needs to go on a business trip to someplace which doesn't have very good computer network access—to the jungles of Africa, for example. Or maybe she is incapacitated after major surgery. She expects to receive some important e-mail, and has instructed her secretary Bob to respond accordingly. How can Alice give Bob the power to sign messages for her, without giving him her private key?

Proxy signatures is a solution [1001]. Alice can give Bob a proxy, such that the following properties hold:

— ***Distinguishability***. Proxy signatures are distinguishable from normal signatures by anyone.

— ***Unforgeability.*** Only the original signer and the designated proxy signer can create a valid proxy signature.

— ***Proxy signer's deviation.*** A proxy signer cannot create a valid proxy signature not detected as a proxy signature.

— ***Verifiability.*** From a proxy signature, a verifier can be convinced of the original signer's agreement on the signed message.

— ***Identifiability.*** An original signer can determine the proxy signer's identity from a proxy signature.

— ***Undeniability.*** A proxy signer cannot disavow an accepted proxy signature he created.

In some cases, a stronger form of identifiability is required—that anyone can determine the proxy signer's identity from the proxy signature. Proxy signature schemes, based on different digital signature schemes, are in [1001].

4.6 GROUP SIGNATURES

David Chaum introduces this problem in [330]:

> A company has several computers, each connected to the local network. Each department of that company has its own printer (also connected to the network) and only persons of that department are allowed to use their department's printer. Before printing, therefore, the printer must be convinced that the user is working in that department. At the same time, the company wants privacy; the user's name may not be revealed. If, however, someone discovers at the end of the day that a printer has been used too often, the director must be able to discover who misused that printer, and send him a bill.

The solution to this problem is called a **group signature**. Group signatures have the following properties:

— Only members of the group can sign messages.
— The receiver of the signature can verify that it is a valid signature from the group.
— The receiver of the signature cannot determine which member of the group is the signer.
— In the case of a dispute, the signature can be "opened" to reveal the identity of the signer.

Group Signatures with a Trusted Arbitrator

This protocol uses a trusted arbitrator:

(1) Trent generates a large pile of public-key/private-key key pairs and gives every member of the group a different list of unique private keys. No keys on any list are identical. (If there are n members of the group, and each member gets m key pairs, then there are $n*m$ total key pairs.)
(2) Trent publishes the master list of all public keys for the group, in random order. Trent keeps a secret record of which keys belong to whom.
(3) When group members want to sign a document, he chooses a key at random from his personal list.
(4) When someone wants to verify that a signature belongs to the group, he looks on the master list for the corresponding public key and verifies the signature.
(5) In the event of a dispute, Trent knows which public key corresponds to which group member.

The problem with this protocol is that it requires a trusted party. Trent knows everyone's private keys and can forge signatures. Also, m must be long enough to preclude attempts to analyze which keys each member uses.

Chaum [330] lists a number of other protocols, some in which Trent is unable to fake signatures and others in which Trent is not even required. Another protocol [348] not only hides the identity of the signer, but also allows new members to join the group. Yet another protocol is [1230].

4.7 Fail-Stop Digital Signatures

Let's say Eve is a very powerful adversary. She has vast computer networks and rooms full of Cray computers—orders of magnitude more computing power than Alice. All of these computers chug away, day and night, trying to break Alice's private key. Finally—success. Eve can now impersonate Alice, forging her signature on documents at will.

Fail-stop digital signatures, introduced by Birgit Pfitzmann and Michael Waidner [1240], prevent this kind of cheating. If Eve forges Alice's signatures after a brute-force attack, then Alice can prove they are forgeries. If Alice signs a document and then disavows the signature, claiming forgery, a court can verify that it is not a forgery.

The basic idea behind fail-stop signatures is that for every possible public key, many possible private keys work with it. Each of these private keys yields many different possible signatures. However, Alice has only one private key and can compute just one signature. Alice doesn't know any of the other private keys.

Eve wants to break Alice's private key. (Eve could also be Alice, trying to compute a second private key for herself.) She collects signed messages and, using her array of Cray computers, tries to recover Alice's private key. Even if she manages to recover a valid private key, there are so many possible private keys that it is far more likely that she has a different one. The probability of Eve's recovering the proper private key can be made so small as to be negligible.

Now, when Eve forges a signed document using the private key she generated, it will have a different signature than if Alice signs the document herself. When Alice is hauled off to court, she can produce two different signatures for the same message and public key (corresponding to her private key and to the private key Eve created) to prove forgery. On the other hand, if Alice cannot produce the two different signatures, there is no forgery and Alice is still bound by her signature.

This signature scheme protects against Eve breaking Alice's signature scheme by sheer computational power. It does nothing against Mallory's much more likely attack of breaking into Alice's house and stealing her private key or Alice's attack of signing a document and then conveniently losing her private key. To protect against the former, Alice should buy herself a good guard dog; that kind of thing is beyond the scope of cryptography.

Additional theory and applications of fail-stop signatures can be found in [1239, 1241,730,731].

4.8 Computing with Encrypted Data

Alice wants to know the solution to some function $f(x)$, for some particular value of x. Unfortunately, her computer is broken. Bob is willing to compute $f(x)$ for her, but

Alice isn't keen on letting Bob know her x. How can Alice let Bob compute $f(x)$ for her without telling him x?

This is the general problem of **computing with encrypted data**, also called **hiding information from an oracle**. (Bob is the oracle; he answers questions.) There are ways to do this for certain functions; they are discussed in Section 23.6.

4.9 BIT COMMITMENT

The Amazing Alice, magician extraordinaire, will now perform a mystifying feat of mental prowess. She will guess the card Bob will choose before he chooses it! Watch as Alice writes her prediction on a piece of paper. Marvel as Alice puts that piece of paper in an envelope and seals it shut. Thrill as Alice hands that sealed envelope to a random member of the audience. "Pick a card, Bob, any card." He looks at it and shows it to Alice and the audience. It's the seven of diamonds. Alice now takes the envelope back from the audience. She rips it open. The prediction, written before Bob chose his card, says "seven of diamonds"! Applause.

To make this work, Alice had to switch envelopes at the end of the trick. However, cryptographic protocols can provide a method immune from any sleight of hand. Why is this useful? Here's a more mundane story:

Stockbroker Alice wants to convince investor Bob that her method of picking winning stocks is sound.

BOB: "Pick five stocks for me. If they are all winners, I'll give you my business."
ALICE: "If I pick five stocks for you, you could invest in them without paying me. Why don't I show you the stocks I picked last month?"
BOB: "How do I know you didn't change last month's picks after you knew their outcome? If you tell me your picks now, I'll know that you can't change them. I won't invest in those stocks until after I've purchased your method. Trust me."
ALICE: "I'd rather show you my picks from last month. I didn't change them. Trust me."

Alice wants to commit to a prediction (i.e., a bit or series of bits) but does not want to reveal her prediction until sometime later. Bob, on the other hand, wants to make sure that Alice cannot change her mind after she has committed to her prediction.

Bit Commitment Using Symmetric Cryptography

This bit-commitment protocol uses symmetric cryptography:

(1) Bob generates a random-bit string, R, and sends it to Alice.

R

(2) Alice creates a message consisting of the bit she wishes to commit to, b (it can actually be several bits), and Bob's random string. She encrypts it with some random key, K, and sends the result back to Bob.

$E_K(R,b)$

That is the commitment portion of the protocol. Bob cannot decrypt the message, so he does not know what the bit is.

When it comes time for Alice to reveal her bit, the protocol continues:

(3) Alice sends Bob the key.

(4) Bob decrypts the message to reveal the bit. He checks his random string to verify the bit's validity.

If the message did not contain Bob's random string, Alice could secretly decrypt the message she handed Bob with a variety of keys until she found one that gave her a bit other than the one she committed to. Since the bit has only two possible values, she is certain to find one after only a few tries. Bob's random string prevents her from using this attack; she has to find a new message that not only has her bit inverted, but also has Bob's random string exactly reproduced. If the encryption algorithm is good, the chance of her finding this is minuscule. Alice cannot change her bit after she commits to it.

Bit Commitment Using One-Way Functions

This protocol uses one-way functions:

(1) Alice generates two random-bit strings, R_1 and R_2.

R_1,R_2

(2) Alice creates a message consisting of her random strings and the bit she wishes to commit to (it can actually be several bits).

(R_1,R_2,b)

(3) Alice computes the one-way function on the message and sends the result, as well as one of the random strings, to Bob.

$H(R_1,R_2,b),R_1$

This transmission from Alice is evidence of commitment. Alice's one-way function in step (3) prevents Bob from inverting the function and determining the bit.

When it comes time for Alice to reveal her bit, the protocol continues:

(4) Alice sends Bob the original message.

(R_1,R_2,b)

(5) Bob computes the one-way function on the message and compares it and R_1, with the value and random string he received in step (3). If they match, the bit is valid.

The benefit of this protocol over the previous one is that Bob does not have to send any messages. Alice sends Bob one message to commit to a bit and another message to reveal the bit.

Bob's random string isn't required because the result of Alice's commitment is a message operated on by a one-way function. Alice cannot cheat and find another message (R_1,R_2',b'), such that $H(R_1,R_2',b') = H(R_1,R_2,b)$. By sending Bob R_1 she is committing to the value of b. If Alice didn't keep R_2 secret, then Bob could compute both $H(R_1,R_2,b)$ and $H(R_1,R_2,b')$ and see which was equal to what he received from Alice.

Bit Commitment Using Pseudo-Random-Sequence Generators

This protocol is even easier [1137]:

(1) Bob generates a random-bit string and sends it to Alice.

R_B

(2) Alice generates a random seed for a pseudo-random-bit generator. Then, for every bit in Bob's random-bit string, she sends Bob either:

(a) the output of the generator if Bob's bit is 0, or

(b) the XOR of output of the generator and her bit, if Bob's bit is 1.

When it comes time for Alice to reveal her bit, the protocol continues:

(3) Alice sends Bob her random seed.

(4) Bob completes step (2) to confirm that Alice was acting fairly.

If Bob's random-bit string is long enough, and the pseudo-random-bit generator is unpredictable, then there is no practical way Alice can cheat.

Blobs

These strings that Alice sends to Bob to commit to a bit are sometimes called **blobs**. A blob is a sequence of bits, although there is no reason in the protocols why it has to be. As Gilles Brassard said, "They could be made out of fairy dust if this were useful" [236]. Blobs have these four properties:

1. Alice can commit to blobs. By committing to a blob, she is committing to a bit.
2. Alice can open any blob she has committed to. When she opens a blob, she can convince Bob of the value of the bit she committed to when she committed to the blob. Thus, she cannot choose to open any blob as either a zero or a one.
3. Bob cannot learn how Alice is able to open any unopened blob she has committed to. This is true even after Alice has opened other blobs.
4. Blobs do not carry any information other than the bit Alice committed to. The blobs themselves, as well as the process by which Alice commits to and opens them, are uncorrelated to anything else that Alice might wish to keep secret from Bob.

4.10 Fair Coin Flips

It's story time with Joe Kilian [831]:

> Alice and Bob wanted to flip a fair coin, but had no physical coin to flip. Alice offered a simple way of flipping a fair coin mentally.
>
> "First, you think up a random bit, then I'll think up a random bit. We'll then exclusive-or the two bits together," she suggested.
>
> "But what if one of us doesn't flip a coin at random?" Bob asked.
>
> "It doesn't matter. As long as one of the bits is truly random, the exclusive-or of the bits should be truly random," Alice replied, and after a moment's reflection, Bob agreed.
>
> A short while later, Alice and Bob happened upon a book on artificial intelligence, lying abandoned by the roadside. A good citizen, Alice said, "One of us must pick this book up and find a suitable waste receptacle." Bob agreed, and suggested they use their coin-flipping protocol to determine who would have to throw the book away.
>
> "If the final bit is a 0, then you will pick the book up, and if it is a 1, then I will," said Alice. "What is your bit?"
>
> Bob replied, "1."
>
> "Why, so is mine," said Alice, slyly, "I guess this isn't your lucky day."
>
> Needless to say, this coin-flipping protocol had a serious bug. While it is true that a truly random bit, x, exclusive-*OR*ed with any independently distributed bit, y, will yield a truly random bit, Alice's protocol did not ensure that the two bits were distributed independently. In fact, it is not hard to verify that no mental protocol can allow two infinitely powerful parties to flip a fair coin. Alice and Bob were in trouble until they received a letter from an obscure graduate student in cryptography. The information in the letter was too theoretical to be of any earthly use to anyone, but the envelope the letter came in was extremely handy.
>
> The next time Alice and Bob wished to flip a coin, they played a modified version of the original protocol. First, Bob decided on a bit, but instead of announcing it immediately, he wrote it down on a piece of paper and placed the paper in the envelope. Next, Alice announced her bit. Finally, Alice and Bob took Bob's bit out of the envelope and computed the random bit. This bit was indeed truly random whenever at least one of them played honestly. Alice and Bob had a working protocol, the cryptographer's dream of social relevance was fulfilled, and they all lived happily ever after.

Those envelopes sound a lot like bit-commitment blobs. When Manuel Blum introduced the problem of flipping a fair coin over a modem [194], he solved it using a bit-commitment protocol:

(1) Alice commits to a random bit, using any of the bit-commitment schemes listed in Section 4.9.

(2) Bob tries to guess the bit.
(3) Alice reveals the bit to Bob. Bob wins the flip if he correctly guessed the bit.

In general, we need a protocol with these properties:

— Alice must flip the coin before Bob guesses.
— Alice must not be able to re-flip the coin after hearing Bob's guess.
— Bob must not be able to know how the coin landed before making his guess.

There are several ways in which we can do this.

Coin Flipping Using One-Way Functions

If Alice and Bob can agree on a one-way function, this protocol is simple:

(1) Alice chooses a random number, x. She computes $y = f(x)$, where $f(x)$ is the one-way function.
(2) Alice sends y to Bob.
(3) Bob guesses whether x is even or odd and sends his guess to Alice.
(4) If Bob's guess is correct, the result of the coin flip is heads. If Bob's guess is incorrect, the result of the coin flip is tails. Alice announces the result of the coin flip and sends x to Bob.
(5) Bob confirms that $y = f(x)$.

The security of this protocol rests in the one-way function. If Alice can find x and x', such that x is even and x' is odd, and $y = f(x) = f(x')$, then she can cheat Bob every time. The least significant bit of $f(x)$ must also be uncorrelated with x. If not, Bob can cheat Alice at least some of the time. For example, if $f(x)$ produces even numbers 75 percent of the time if x is even, Bob has an advantage. (Sometimes the least significant bit is not the best one to use in this application, because it can be easier to compute.)

Coin Flipping Using Public-Key Cryptography

This protocol works with either public-key cryptography or symmetric cryptography. The only requirement is that the algorithm commute. That is:

$$D_{K_1}(E_{K_2}(E_{K_1}(M))) = E_{K_2}(M)$$

In general, this property is not true for symmetric algorithms, but it is true for some public-key algorithms (RSA with identical moduli, for example). This is the protocol:

(1) Alice and Bob each generate a public-key/private-key key pair.
(2) Alice generates two messages, one indicating heads and the other indicating tails. These messages should contain some unique random string, so that

she can verify their authenticity later in the protocol. Alice encrypts both messages with her public key and sends them to Bob in a random order.

$E_A(M_1)$, $E_A(M_2)$

(3) Bob, who cannot read either message, chooses one at random. (He can sing "eeny meeny miney moe," engage a malicious computer intent on subverting the protocol, or consult the *I Ching*—it doesn't matter.) He encrypts it with his public key and sends it back to Alice.

$E_B(E_A(M))$

M is either M_1 or M_2.

(4) Alice, who cannot read the message sent back to her, decrypts it with her private key and then sends it back to Bob.

$D_A(E_B(E_A(M))) = E_B(M_1)$ if $M = M_1$, or

$E_B(M_2)$ if $M = M_2$

(5) Bob decrypts the message with his private key to reveal the result of the coin flip. He sends the decrypted message to Alice.

$D_B(E_B(M_1)) = M_1$ or $D_B(E_B(M_2)) = M_2$

(6) Alice reads the result of the coin flip and verifies that the random string is correct.

(7) Both Alice and Bob reveal their key pairs so that both can verify that the other did not cheat.

This protocol is self-enforcing. Either party can immediately detect cheating by the other, and no trusted third party is required to participate in either the actual protocol or any adjudication after the protocol has been completed. To see how this works, let's try to cheat.

If Alice wanted to cheat and force heads, she has three potential ways of affecting the outcome. First, she could encrypt two "heads" messages in step (2). Bob would discover this when Alice revealed her keys at step (7). Second, she could use some other key to decrypt the message in step (4). This would result in gibberish, which Bob would discover in step (5). Third, she could lie about the validity of the message in step (6). Bob would also discover this in step (7), when Alice could not prove that the message was not valid. Of course, Alice could refuse to participate in the protocol at any step, at which point Alice's attempted deception would be obvious to Bob.

If Bob wanted to cheat and force "tails," his options are just as poor. He could incorrectly encrypt a message at step (3), but Alice would discover this when she looked at the final message at step (6). He could improperly perform step (5), but this would also result in gibberish, which Alice would discover at step (6). He could claim that he could not properly perform step (5) because of some cheating on the part of Alice, but this form of cheating would be discovered at step (7). Finally, he could send a "tails" message to Alice at step (5), regardless of the message he decrypted, but Alice would immediately be able to check the message for authenticity at step (6).

Flipping Coins into a Well

It is interesting to note that in all these protocols, Alice and Bob don't learn the result of the coin flip at the same time. Each protocol has a point where one of the parties (Alice in the first two protocols and Bob in the last one) knows the result of the coin flip but cannot change it. That party can, however, delay disclosing the result to the other party. This is known as **flipping coins into a well**. Imagine a well. Alice is next to the well and Bob is far away. Bob throws the coin and it lands in the well. Alice can now look into the well and see the result, but she cannot reach down to change it. Bob cannot see the result until Alice lets him come close enough to look.

Key Generation Using Coin Flipping

A real application for this protocol is session-key generation. Coin-flipping protocols allow Alice and Bob to generate a random session key such that neither can influence what the session key will be. And assuming that Alice and Bob encrypt their exchanges, this key generation is secure from eavesdropping as well.

4.11 MENTAL POKER

A protocol similar to the public-key fair coin flip protocol allows Alice and Bob to play poker with each other via electronic mail. Instead of Alice making and encrypting two messages, one for heads and one for tails, she makes 52 messages, M_1, M_2, . . . , M_{52}, one for each card in the deck. Bob chooses five messages at random, encrypts them with his public key, and then sends them back to Alice. Alice decrypts the messages and sends them back to Bob, who decrypts them to determine his hand. He then chooses five more messages at random and sends them back to Alice as he received them; she decrypts these and they become her hand. During the game, additional cards can be dealt to either player by repeating the procedure. At the end of the game, Alice and Bob both reveal their cards and key pairs so that each can be assured that the other did not cheat.

Mental Poker with Three Players

Poker is more fun with more players. The basic mental poker protocol can easily be extended to three or more players. In this case, too, the cryptographic algorithm must be commutative.

(1) Alice, Bob, and Carol each generate a public-key/private-key key pair.

(2) Alice generates 52 messages, one for each card in the deck. These messages should contain some unique random string, so that she can verify their authenticity later in the protocol. Alice encrypts all the messages with her public key and sends them to Bob.

$E_A(M_n)$

(3) Bob, who cannot read any of the messages, chooses five at random. He encrypts them with his public key and sends them back to Alice.

$E_B(E_A(M_n))$

(4) Bob sends the other 47 messages to Carol.

$E_A(M_n)$

(5) Carol, who cannot read any of the messages, chooses five at random. She encrypts them with her public key and sends them to Alice.

$E_C(E_A(M_n))$

(6) Alice, who cannot read any of the messages sent back to her, decrypts them with her private key and then sends them back to Bob or Carol (depending on where they came from).

$D_A(E_B(E_A(M_n))) = E_B(M_n)$

$D_A(E_C(E_A(M_n))) = E_C(M_n)$

(7) Bob and Carol decrypt the messages with their keys to reveal their hands.

$D_B(E_B(M_n)) = M_n$

$D_C(E_C(M_n)) = M_n$

(8) Carol chooses five more messages at random from the remaining 42. She sends them to Alice.

$E_A(M_n)$

(9) Alice decrypts the messages with her private key to reveal her hand.

$D_A(E_A(M_n)) = M_n$

(10) At the end of the game Alice, Bob, and Carol all reveal their hands and all of their keys so that everyone can make sure that no one has cheated.

Additional cards can be dealt in the same manner. If Bob or Carol wants a card, either one can take the encrypted deck and go through the protocol with Alice. If Alice wants a card, whoever currently has the deck sends her a random card.

Ideally, step (10) would not be necessary. All players shouldn't be required to reveal their hands at the end of the protocol; only those who haven't folded. Since step (10) is part of the protocol designed only to catch cheaters, perhaps there are improvements.

In poker, one is only interested in whether the winner cheated. Everyone else can cheat as much as they want, as long as they still lose. (Actually, this is not really true. Someone can, while losing, collect data on another player's poker style.) So, let's look at cases in which different players win.

If Alice wins, she reveals her hand and her keys. Bob can use Alice's private key to confirm that Alice performed step (2) correctly—that each of the 52 messages corresponded to a different card. Carol can confirm that Alice is not lying about her hand by encrypting the cards with Alice's public key and verifying that they are the same as the encrypted messages she sent to her in step (8).

If either Bob or Carol wins, the winner reveals his hand and keys. Alice can confirm that the cards are legitimate by checking her random strings. She can also confirm that the cards are the ones dealt by encrypting the cards with the winner's public key and verifying that they are the same as the encrypted messages she received in step (3) or (5).

This protocol isn't secure against collusion among malicious players. Alice and another player can effectively gang up on the third and together swindle that player out of everything without raising suspicion. Therefore, it is important to check all the keys and random strings every time the players reveal their hands. And if you're sitting around the virtual table with two people who never reveal their hands whenever one of them is the dealer (Alice, in the previous protocol), stop playing.

Understand that while this is all interesting theory, actually implementing it on a computer is an arduous task. A Sparc implementation with three players on separate workstations takes eight hours to shuffle a deck of cards, let alone play an actual game [513].

Attacks against Poker Protocols

Cryptographers have shown that a small amount of information is leaked by these poker protocols if the RSA public-key algorithm is used [453,573]. Specifically, if the binary representation of the card is a quadratic residue (see Section 11.3), then the encryption of the card is also a quadratic residue. This property can be used to "mark" some cards—all the aces, for example. This does not reveal much about the hands, but in a game such as poker even a tiny bit of information can be an advantage in the long run.

Shafi Goldwasser and Silvio Micali [624] developed a two-player mental-poker protocol that fixes this problem, although its complexity makes it far more theoretical than practical. A general n-player poker protocol that eliminates the problem of information leakage was developed in [389].

Other research on poker protocols can be found in [573,1634,389]. A complicated protocol that allows players to not reveal their hands can be found in [390]. Don Coppersmith discusses two ways to cheat at mental poker using the RSA algorithm [370].

Anonymous Key Distribution

While it is unlikely that anyone is going to use this protocol to play poker via modem, Charles Pfleeger discusses a situation in which this type of protocol would come in handy [1244].

Consider the problem of key distribution. If we assume that people cannot generate their own keys (they might have to be of a certain form, or have to be signed by some organization, or something similar), we have to set up a Key Distribution Center to generate and distribute keys. The problem is that we have to figure out some way of distributing keys such that no one, including the server, can figure out who got which key.

This protocol solves the problem:

(1) Alice generates a public-key/private-key key pair. For this protocol, she keeps both keys secret.

(2) The KDC generates a continuous stream of keys.

(3) The KDC encrypts the keys, one by one, with its own public key.

(4) The KDC transmits the encrypted keys, one by one, onto the network.
(5) Alice chooses a key at random.
(6) Alice encrypts the chosen key with her public key.
(7) Alice waits a while (long enough so the server has no idea which key she has chosen) and sends the double-encrypted key back to the KDC.
(8) The KDC decrypts the double-encrypted key with its private key, leaving a key encrypted with Alice's public key.
(9) The server sends the encrypted key back to Alice.
(10) Alice decrypts the key with her private key.

Eve, sitting in the middle of this protocol, has no idea what key Alice chose. She sees a continuous stream of keys go by in step (4). When Alice sends the key back to the server in step (7), it is encrypted with her public key, which is also secret during this protocol. Eve has no way of correlating it with the stream of keys. When the server sends the key back to Alice in step (9), it is also encrypted with Alice's public key. Only when Alice decrypts the key in step (10) is the key revealed.

If you use RSA, this protocol leaks information at the rate of one bit per message. It's the quadratic residues again. If you're going to distribute keys in this manner, make sure this leakage isn't enough to matter. Also, the stream of keys from the KDC must be great enough to preclude a brute-force attack. Of course, if Alice can't trust the KDC, then she shouldn't be getting keys from it. A malicious KDC could presumably keep records of every key it generates. Then, it could search them all to determine which is Alice's.

This protocol also assumes that Alice is going to act fairly. There are things she can do, using RSA, to get more information than she might otherwise. This is not a problem in our scenario, but can be in other circumstances.

4.12 ONE-WAY ACCUMULATORS

Alice is a member of Cabal, Inc. Occasionally she has to meet with other members in dimly lit restaurants and whisper secrets back and forth. The problem is that the restaurants are so dimly lit that she has trouble knowing if the person across the table from her is also a member.

Cabal Inc. can choose from several solutions. Every member can carry a membership list. This has two problems. One, everyone now has to carry a large database, and two, they have to guard that membership list pretty carefully. Alternatively, a trusted secretary could issue digitally signed ID cards. This has the added advantage of allowing outsiders to verify members (for discounts at the local grocery store, for example), but it requires a trusted secretary. Nobody at Cabal, Inc. can be trusted to that degree.

A novel solution is to use something called a **one-way accumulator** [116]. This is sort of like a one-way hash function, except that it is commutative. That is, it is possible to hash the database of members in any order and get the same value. More-over, it is possible to add members into the hash and get a new hash, again without regard to order.

So, here's what Alice does. She calculates the accumulation of every member's name other than herself. Then she saves that single value along with her own name. Bob, and every other member, does the same. Now, when Alice and Bob meet in the dimly lit restaurant, they simply trade accumulations and names with each other. Alice confirms that Bob's name added to his accumulation is equal to Alice's name added to her accumulation. Bob does the same. Now they both know that the other is a member. And at the same time, neither can figure out the identities of any other member.

Even better, nonmembers can be given the accumulation of everybody. Now Alice can verify her membership to a nonmember (for membership discounts at their local counterspy shop, perhaps) without the nonmember being able to figure out the entire membership list.

New members can be added just by sending around the new names. Unfortunately, the only way to delete a member is to send everyone a new list and have them recompute their accumulations. But Cabal, Inc. only has to do that if a member resigns; dead members can remain on the list. (Oddly enough, this has never been a problem.)

This is a clever idea, and has applications whenever you want the same effect as digital signatures without a centralized signer.

4.13 ALL-OR-NOTHING DISCLOSURE OF SECRETS

Imagine that Alice is a former agent of the former Soviet Union, now unemployed. In order to make money, Alice sells secrets. Anyone who is willing to pay the price can buy a secret. She even has a catalog. All her secrets are listed by number, with tantalizing titles: "Where is Jimmy Hoffa?", "Who is secretly controlling the Trilateral Commission?", "Why does Boris Yeltsin always look like he swallowed a live frog?", and so on.

Alice won't give away two secrets for the price of one or even partial information about any of the secrets. Bob, a potential buyer, doesn't want to pay for random secrets. He also doesn't want to tell Alice which secrets he wants. It's none of Alice's business, and besides, Alice could then add "what secrets Bob is interested in" to her catalog.

A poker protocol won't work in this case, because at the end of the protocol Alice and Bob have to reveal their hands to each other. There are also tricks Bob can do to learn more than one secret.

The solution is called **all-or-nothing disclosure of secrets (ANDOS)** [246] because, as soon as Bob has gained any information whatsoever about one of Alice's secrets, he has wasted his chance to learn anything about any of the other secrets.

There are several ANDOS protocols in the cryptographic literature. Some of them are discussed in Section 23.9.

4.14 Key Escrow

This excerpt is from Silvio Micali's introduction to the topic [1084]:

> Currently, court-authorized line tapping is an effective method for securing criminals to justice. More importantly, in our opinion, it also prevents the further spread of crime by deterring the use of ordinary communication networks for unlawful purposes. Thus, there is a legitimate concern that widespread use of public-key cryptography may be a big boost for criminal and terrorist organizations. Indeed, many bills propose that a proper governmental agency, under circumstances allowed by law, should be able to obtain the clear text of any communication over a public network. At the present time, this requirement would translate into coercing citizens to either (1) *using weak cryptosystems*—i.e., cryptosystems that the proper authorities (but also everybody else!) could crack with a moderate effort—or (2) *surrendering*, a priori, *their secret key* to the authority. It is not surprising that such alternatives have legitimately alarmed many concerned citizens, generating as reaction the feeling that privacy should come before national security and law enforcement.

Key escrow is the heart of the U.S. government's Clipper program and its Escrowed Encryption Standard. The challenge here is to develop a cryptosystem that both protects individual privacy but at the same time allows for court-authorized wiretaps.

The Escrowed Encryption Standard gets its security from tamperproof hardware. Each encryption chip has a unique ID number and secret key. This key is split into two pieces and stored, along with the ID number, by two different escrow agencies. Every time the chip encrypts a data file, it first encrypts the session key with this unique secret key. Then it transmits this encrypted session key and its ID number over the communications channel. When some law enforcement agency wants to decrypt traffic encrypted with one of these chips, it listens for the ID number, collects the appropriate keys from the escrow agencies, XORs them together, decrypts the session key, and then uses the session key to decrypt the message traffic. There are more complications to make this scheme work in the face of cheaters; see Section 24.16 for details. The same thing can be done in software, using public-key cryptography [77,1579,1580,1581].

Micali calls his idea **fair cryptosystems** [1084,1085]. (The U.S. government reportedly paid Micali $1,000,000 for the use of his patents [1086,1087] in their Escrowed Encryption Standard; Banker's Trust then bought Micali's patent.) In these cryptosystems, the private key is broken up into pieces and distributed to different authorities. Like a secret sharing scheme, the authorities can get together and reconstruct the private key. However, the pieces have the additional property that they can be individually verified to be correct, without reconstructing the private key.

Alice can create her own private key and give a piece to each of n trustees. None of these trustees can recover Alice's private key. However, each trustee can verify that his piece is a valid piece of the private key; Alice cannot send one of the trustees a random-bit string and hope to get away with it. If the courts authorize a wiretap, the relevant law enforcement authorities can serve a court order on the *n* trustees to surrender their pieces. With all *n* pieces, the authorities reconstruct the private key and can wiretap Alice's communications lines. On the other hand, Mallory has to corrupt all *n* trustees in order to be able to reconstruct Alice's key and violate her privacy.

Here's how the protocol works:

(1) Alice creates her private-key/public-key key pair. She splits the private key into several public pieces and private pieces.

(2) Alice sends a public piece and corresponding private piece to each of the trustees. These messages must be encrypted. She also sends the public key to the KDC.

(3) Each trustee, independently, performs a calculation on its public piece and its private piece to confirm that they are correct. Each trustee stores the private piece somewhere secure and sends the public piece to the KDC.

(4) The KDC performs another calculation on the public pieces and the public key. Assuming that everything is correct, it signs the public key and either sends it back to Alice or posts it in a database somewhere.

If the courts order a wiretap, then each of the trustees surrenders his or her piece to the KDC, and the KDC can reconstruct the private key. Before this surrender, neither the KDC nor any individual trustee can reconstruct the private key; all the trustees are required to reconstruct the key.

Any public-key cryptography algorithm can be made fair in this manner. Some particular algorithms are discussed in Section 23.10. Micali's paper [1084,1085] discusses ways to combine this with a threshold scheme, so that a subset of the trustees (e.g., three out of five) is required to reconstruct the private key. He also shows how to combine this with oblivious transfer (see Section 5.5) so that the trustees do not know whose private key is being reconstructed.

Fair cryptosystems aren't perfect. A criminal can exploit the system, using a subliminal channel (see Section 4.2) to embed another secret key into his piece. This way, he can communicate securely with someone else using this subliminal key without having to worry about court-authorized wiretapping. Another protocol, called **failsafe key escrowing**, solves this problem [946,833]. Section 23.10 describes the algorithm and protocol.

The Politics of Key Escrow

Aside from the government's key-escrow plans, several commercial key-escrow proposals are floating around. This leads to the obvious question: What are the advantages of key-escrow for the user?

Well, there really aren't any. The user gains nothing from key escrow that he couldn't provide himself. He can already back up his keys if he wants (see Section 8.8). Key-escrow guarantees that the police can eavesdrop on his conversations or read his data files even though they are encrypted. It guarantees that the NSA can eavesdrop on his international phone calls—without a warrant—even though they are encrypted. Perhaps he will be allowed to use cryptography in countries that now ban it, but that seems to be the only advantage.

Key escrow has considerable disadvantages. The user has to trust the escrow agents' security procedures, as well as the integrity of the people involved. He has to trust the escrow agents not to change their policies, the government not to change its laws, and those with lawful authority to get his keys to do so lawfully and responsibly. Imagine a major terrorist attack in New York; what sorts of limits on the police would be thrown aside in the aftermath?

It is hard to imagine escrowed encryption schemes working as their advocates imagine without some kind of legal pressure. The obvious next step is a ban on the use of non-escrowed encryption. This is probably the only way to make a commercial system pay, and it's certainly the only way to get technologically sophisticated criminals and terrorists to use it. It's not clear how difficult outlawing non-escrowed cryptography will be, or how it will affect cryptography as an academic discipline. How can I research software-oriented cryptography algorithms without having software non-escrowed encryption devices in my possession; will I need a special license?

And there are legal questions. How do escrowed keys affect users' liability, should some encrypted data get out? If the U.S. government is trying to protect the escrow agencies, will there be the implicit assumption that if the secret was compromised by either the user or the escrow agency, then it must have been the user?

What if a major key-escrow service, either government or commercial, had its entire escrowed key database stolen? What if the U.S. government tried to keep this quiet for a while? Clearly, this would have an impact on users' willingness to use key escrow. If it's not voluntary, a couple of scandals like this would increase political pressure to either make it voluntary, or to add complex new regulations to the industry.

Even more dangerous is a scandal where it becomes public that political opponent of the current administration, or some outspoken critic of some intelligence or police agencies has been under surveillance for years. This could raise public sentiment strongly against escrowed encryption.

If signature keys are escrowed as well as encryption keys, there are additional issues. Is it acceptable for the authorities to use signature keys to run operations against suspected criminals? Will the authenticity of signatures based on escrowed keys be accepted in courts? What recourse do users have if the authorities actually do use their signature keys to sign some unfavorable contract, to help out a state-supported industry, or just to steal money?

The globalization of cryptography raises an additional set of questions. Will key-escrow policies be compatible across national borders? Will multi-national corporations have to keep separate escrowed keys in every country to stay in compliance with the various local laws? Without some kind of compatibility, one of the supposed advantages of key-escrow schemes (international use of strong encryption) falls apart.

What if some countries don't accept the security of escrow agencies on faith? How do users do business there? Are their digital contracts upheld by their courts, or is the fact that their signature key is held in escrow in the U.S. going to allow them to claim in Switzerland that someone else could have signed this electronic contract? Or will there be special waivers for people who do business in such countries?

And what about industrial espionage? There is no reason to believe that countries which currently conduct industrial espionage for their important or state-run companies will refrain from doing so on key-escrowed encryption systems. Indeed, since virtually no country is going to allow other countries to oversee its intelligence operations, widespread use of escrowed encryption will probably increase the use of wiretaps.

Even if countries with good civil rights records use key escrow only for the legitimate pursuit of criminals and terrorists, it's certain to be used elsewhere to keep track of dissidents, blackmail political opponents, and so on. Digital communications offer the opportunity to do a much more thorough job of monitoring citizens' actions, opinions, purchases, and associations than is possible in an analog world.

It's not clear how this will affect commercial key escrow, except that 20 years from now, selling Turkey or China a ready-made key-escrow system may look a lot like selling shock batons to South Africa in 1970, or building a chemical plant for Iraq in 1980. Even worse, effortless and untraceable tapping of communications may tempt a number of governments into tracking many of their citizens' communications, even those which haven't generally tried to do so before. And there's no guarantee that liberal democracies will be immune to this temptation.