6.857 Computer and Network Security	October 18, 2001

Lecture Notes 12 : OCB Mode and SSL

Lecturer: Ron Rivest *Scribe: Agarwal/Bull/Karlovich/Muller*

Outline

- OCB Mode
- Key Establishment (SSL)

1 Offset Codebook Mode

Traditionally, confidentiality and authentication are viewed as separate problems, and are treated independently. Specifically, there is no authentication in a typical encryption mode of operation. Most modes simply specify a way of getting plaintext bits from ciphertext.[1] Most authentication schemes also do not address confidentiality.

In 6.033, to achieve both authenticity and confidentiality, we just chose one algorithm from each category and layered them on top of each other. In other words, encrypt the message to get the ciphertext, apply a MAC function, and append the MAC to the end of the encrypted message. The recipient of the message reverses this process. Unfortunately, this requires a double pass over the message, once for encryption/decryption and once for authentication. Until recently (i.e. this year), most proposals to achieve both confidentiality and authenticity in one pass over a message have failed.

Offset Codebook Mode (OCB)[2] is a new proposal by Phillip Rogaway at UC Davis, with help from Mihir Bellare (UC San Diego), John Black (University of Nevada), and Ted Krovetz (Digital Fountain), as part of an ongoing process of the government to develop modes of operation that work well with AES[3]. OCB is a combined encryption and authentication mode which has so far survived analysis. It produces both authentication and confidentiality with about the same workload of confidentiality alone.

OCB assumes that you begin with a fixed length block cipher (such as AES). In the case of AES-128, the input plaintext is 128 bits, the output ciphertext is 128 bits, and the block cipher requires a 128-bit key. This strategy is fine by itself for just encrypting blocks of specified lengths. However, what about messages that are shorter or longer? What do you do about authentication?

[1] Some authentication modes closely resemble encryption modes. For instance, you can tweak CBC to get a nice authentication mode.
[2] `http://www.cs.ucdavis.edu/~rogaway/ocb/ocb.htm`
[3] `http://csrc.nist.gov/encryption/modes/`

1.1 Design Goals

1. **Encryption of Arbitrary Length Messages**

   OCB should be able to encrypt a message M of arbitrary length, not just multiples of the block length. The ciphertext should have the same size as M, excluding the variable length authentication tag that is appended.

   Appending an authentication tag to the ciphertext has several advantages. With this scheme, the size of the tag controls the level of authentication. Although an adversary could try to forge a message by guessing a tag at random, they would only have a $\frac{1}{2^\tau}$, (where τ = number of authentication bits) chance of succeeding.

   On the decryption side, the decryptor will check the tag, possibly rejecting the whole thing as invalid. Unlike confidentiality alone, OCB could reject the message as unauthentic even if it decrypts properly. Maybe a bit got trashed during transmission, a forgery is being attempted, or it's invalid for some other reason.

   If the ciphertext passes the test, then OCB produces the plaintext normally.

2. **Simplicity**

   When no other goal takes precedence, we would like the mode of operation to be as easy to understand and implement as possible.

3. **Efficiency**

   - **Achieves a nearly optimal number of block cipher calls**

     Since invoking our block cipher is expensive, it becomes the primitive operation, and performance directly depends on how many block cipher calls are necessary. A message of N blocks requires at least N block cipher calls to encrypt the entire message. OCB exceeds this minimum requirement by 2 (explained later), so the number of block cipher calls that OCB makes = 2 + (number of blocks in message).

   - **Parallelizable**

     Unlike cipherblock chaining, in which each block of ciphertext is XORed to the next block of the message, OCB can encrypt every block of the message simultaneously. This feature becomes significant as processors become more parallel.

   - **Single-key**

     Using a standard cipher followed by a MAC can require two different keys, and therefore two key setup routines. OCB uses the same key for encryption and authentication.

   In practice, OCB is 6% slower than CBC (when only confidentiality is achieved). However, when compared to CBC with a MAC added, OCB is 50% faster.

4. **Provable Security**

   OCB has provable security, which we will cover after our description.

1.2 How it Works

Setup

Each message requires a nonce N, similar to an initialization vector in schemes like CBC. However, unlike other schemes that require random procedures, the only requirement on N is that it be non-repeating. Even a simple counter may be used to generate N. (Bonus feature: If both sender and receiver have synchronized counters, N does not need to be communicated explicitly.) N is used in the setup of OCB, which generates L and R as in Figure 1. L and R are used to produce the "offsets" for each message block.

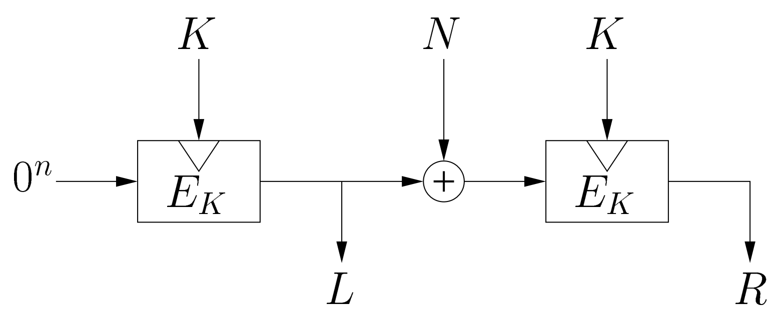

Figure 1: OCB setup.

The setup procedure for OCB is performed as follows. First, encrypt an all zero vector with the agreed on key K, producing L. Next, encrypt with K the XOR of L and the nonce N to get the value R. L and R are used to produce the offsets z_i for each message block M_i.

Encryption

Now we are ready to encrypt each message block M_i. In the encryption process, we must make sure that $E_K(M_i) \neq E_K(M_j)$ even if $M_i = M_j$. In order to achieve unique encryption of equivalent blocks, we XOR them with a constant derived from L and R, namely z_i. Then we encrypt the block and XOR the result with z_i to get our ciphertext block, as shown in Figure 2. Each message block M_i is encrypted in the same way (with different offsets z_i), for $1 \leq i \leq m-1$, where M_m is the last block (or partial block) of M.

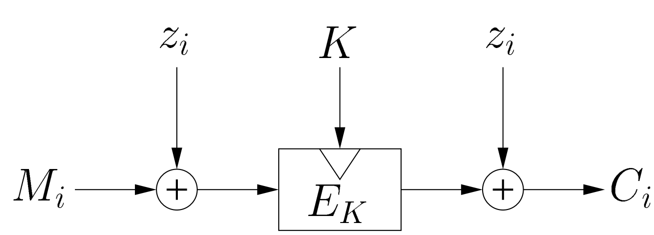

Figure 2: OCB cipher of message block i.

Offset z_i

A first approximation to the calculation of z_i for each i is $z_i = i \cdot L + R$. A calculation like this demonstrates the important features that each z_i is pairwise independent with any other z_j, which results in one z not being able to be predicted from the knowledge of another z. Also, given i, it is easy to compute z_i. Therefore, separate processors can compute the z_i's independently, resulting in increased parallelism.

In practice, z_i is computed using a set of γ's:

$$z_i = \gamma_i \cdot L + R, \text{ over a finite field } GF(2^n)$$

This equation is actually much more efficient. Adding two 128-bit numbers with C is difficult (or awkward at best). In a finite field the addition is really just XOR, and multiplication becomes a shift. However the calculation is done, z_i amounts to a sequence of constants that are used as offsets in OCB.

Question: Why do we XOR twice, wouldn't it be enough to just XOR before the encryption?

Answer: There's going to be a lot of questions like that, I'm not sure I'm going to be able to answer all of them. These details are subtle, they tried to do simpler things, they didn't work. Since it's simpler to have one XOR rather than both, they must have tried it, and it must have failed. But I am not sure since they didn't write up things that failed. There are many many proposals where people sort of guessed things, and didn't have proofs. This construction is provably secure. They probably tried to eliminate that second XOR and couldn't. Also, there's some history, other papers that used similar schemes. This could be leftover from similiar papers that tried to do things to get keylength extension.

Variable message lengths

The encryption scheme described earlier is only good for the first $m-1$ blocks, for which we can apply the block cipher without any tricks. Now consider the last message block M_m. To deal with this last block, and produce ciphertext that has the same length as the plaintext, OCB does the following:

Take the length of M_m, l (l is represented with n bits, where n is the size of a regular block). XOR it with another full-length constant $L \cdot x^{-1}$, then XOR with x_m (full-length). Encrypt that result, and XOR the leading l bits with M_m to get the ciphertext. This process is shown in Figure 3. The last block can be shorter than usual, or full-length. Either way, this method will produce a valid encryption. The ciphertext C_m is l bits long, so we end up with the same length ciphertext as the original message.

To decrypt we just do everything backwards. For most of the blocks this is trivial, but the last one is a little different. We know the length of the last cipher block, which is the same as the length

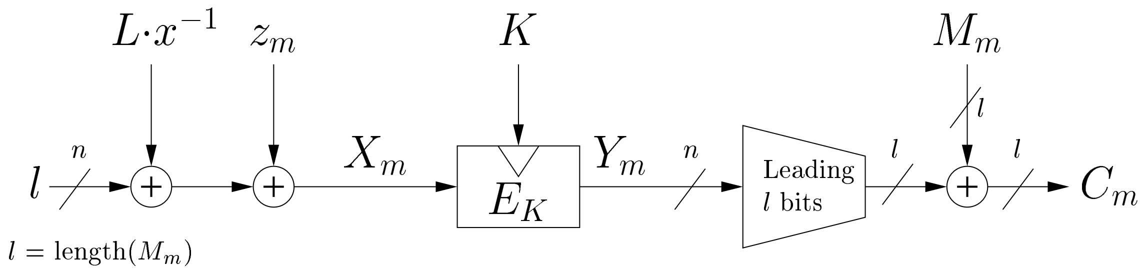

Figure 3: OCB cipher for the last block of M. Data buses with unusual bit lengths are marked.

of the last message block, so you know l. Then you can XOR it with $L \cdot x^{-1}$ and z_m, encrypt and finally XOR that with ciphertext result, to get the last M_m.

Authentication

For starters, we define τ to be the authentication tag length, where $0 <= \tau <= n$, $n =$ the length of the block use in the block cipher. The length of τ is equivalent to the bits of authentication method (i.e. hash). τ is variable length since some authentication methods (such as those for video encryption) might only need a few bits per block. τ is the OCB parameter that is used to control forgery. Similarly, the particular block cipher used controls the level of confidentiality.

The signature is computed as follows (see Figure 4):

To get the checksum we compute, $M_1 \bigoplus M_2 \bigoplus \cdots \bigoplus M_{m-1} \bigoplus C_m 0^{n-l} \bigoplus Y_m$. Note that when M_m is full-length, $C_m 0^{n-l} \bigoplus Y_m$ is equivalent to it.

Take the checksum, XOR it with Z_m. Encrypt that with same key K, then take the first τ bits to get the tag.

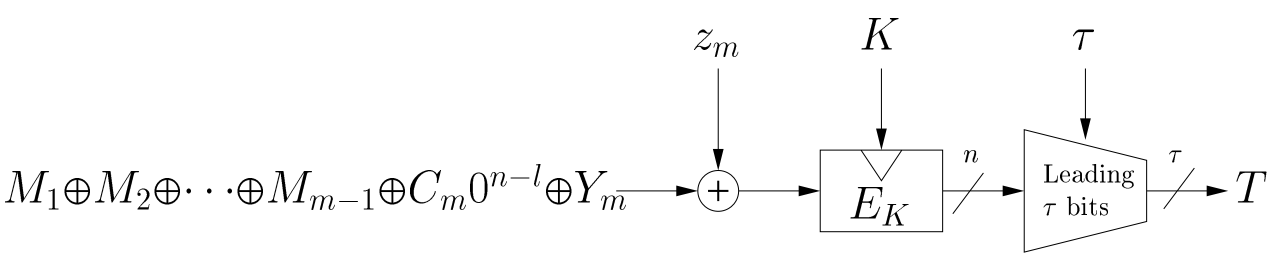

Figure 4: OCB authentication. The signature is the output T.

This provides authentication almost for free on top of encryption. The whole thing is done with two more block ciphers per message than regular encryption costs. The first is in encrypting $L \bigoplus N$ to get R, and the second is for the authentication tag.

To verify the signature, the decryptor can recompute the checksum, then recompute the tag, and finally check that it's equal to the one that was sent.

Question: Is Tau global or included?
Answer: Tau is negotiated ahead of time, along with other factors like the encryption scheme and length parameters.

Question: Why is this computation (truncating Tau) done? Why not just send the whole tag?
Answer: You could do that. Their goal was to create a scheme that could handle anything. Suppose you're doing character by character encryption. If your messages are 8 bits long, you don't want to send 128 bits of authentication with each one. You want to allow users to choose their own level of authentication. Banks often use 32 bits. One out of a billion chance of forging.

Question: Why does authentication work? Why can't somebody just forge the whole thing since they know K?
Answer: K is not public, it is a key known only to the sender and receiver. Receiver is counting on the fact that the only other person who could produce the checksum knows K. An adversary is presumed to not know K.

OCB review

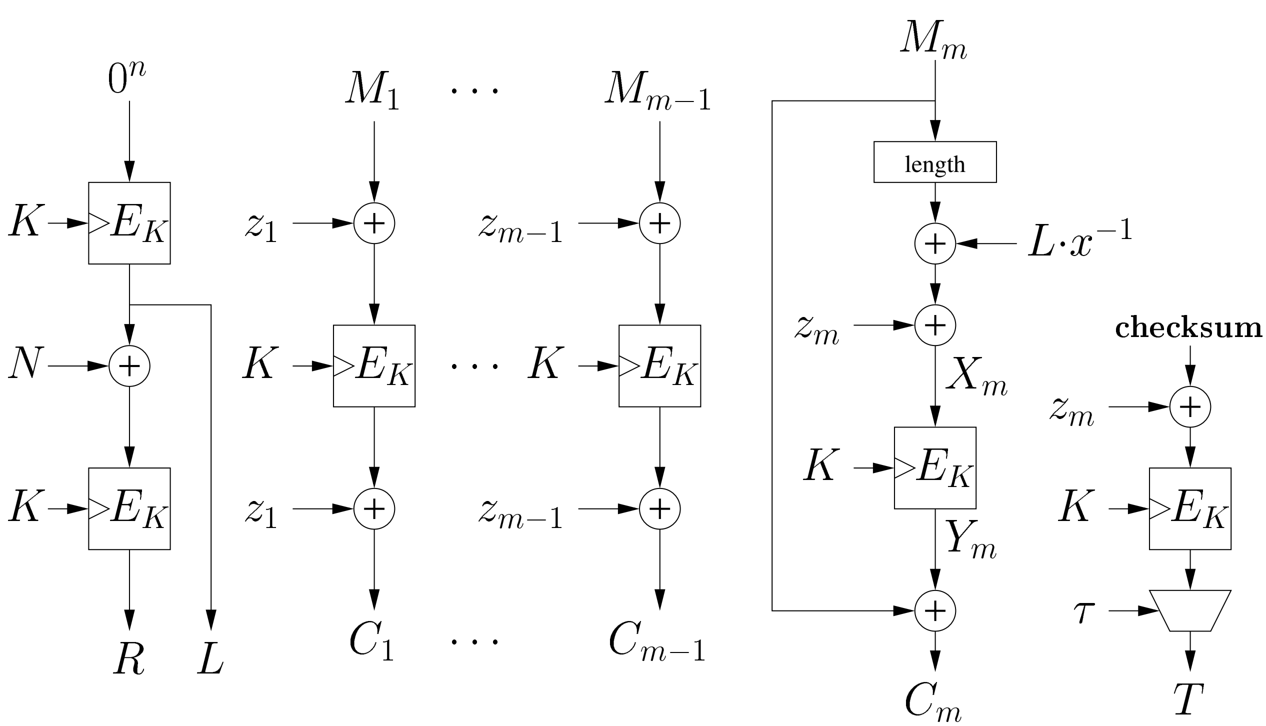

Figure 5: OCB.

Features:

- Arbitrary length messages.
- Single k for confidentiality and authentication.
- Nonce used once, no randomness required.
- Don't need to know the length of the message ahead of time to start working on it, can compute if you're being fed the language block by block. The same for decryption.
- Little endian versus big endian not a problem because no adding operations, only XORs because of finite field arithmetic.
- Can preprocess all Ns.
- Has provable security.

Provable Security

There are two things we are trying to achieve: Privacy and Authentication. What follows is an explanation of two theorems that they proved, but not the proofs themselves. These statements are examples of classic kinds of statements in this area of cryptography.

The proofs assume we have a secure block cipher. The test is whether the mode is indistinguishable from random text. They put this mode of operation to the most severe test possible: The adversary has block box access to it.

To do this test, we give the adversary access to a box that has a secret key, AES, etc inside of it. Adversary can put in any message, get out the appropriate ciphertext.

We set up two boxes: One has OCB implemented, takes a message, produces ciphertext plus tag. The other puts out random bits of the appropriate length (message plus tag). It does a random permutation, and properly memorizes messages in a table, so that if you submit the same message you get the same answer.

We allow the adversary to do encryption or decryption as he desires. The adversary is trying to be able to learn about the box. Can he tell the two boxes apart? He could be right half the time by guessing. The question is could he be right more than half the time?

The claim is that the advantage is bounded by

$$\frac{1.5 \cdot \bar{\sigma}^2}{2^n}$$

where

$$\bar{\sigma} = \sigma + 2 \cdot q + 3$$

Here σ is the total number of blocks asked and q is the total number of queries. $\bar{\sigma}^2$ is going to be very small compared to 2^n, so quite close to zero. This means an adversary can't distinguish ciphertext from randomtext. They prove it with a secure block cipher, not AES in particular.

Similarly for authentication:

Probability of forgery is at most:

$$\frac{1.5 \cdot \bar{\sigma}^2}{2^n} + \frac{1}{2^\tau}$$

To break an authentication, the adversary is allowed to ask a certain number of questions, then required to forge a message. Forgery means to come up with an authentication tag for some other message (that he hasn't already asked) that will be accepted. The chance of a successful forgery occurring at random is $\frac{1}{2^\tau}$.

The proof is that the amount you can be better than random is bounded by $\frac{1.5 \cdot \bar{\sigma}^2}{2^n}$. Again quite small compared to 2^τ. So it provides both good privacy and good protection against forgery.

2 Secure Socket Layer

2.1 Why do we need SSL?

OCB assumes that the sender and receiver have a shared key K, however we have not yet discussed how both parties secretly obtain a common K. Thus, we need some way to establish shared secret keys, a subject that we're only going to touch on. Secure Socket Layer (SSL)[4] is one of the solutions to this problem.

The simplest solution is to have a prearranged key. Imagine that we meet at a party and I give you a cocktail napkin with some random numbers. However, this cannot work in all scenarios. Often, the second party we are dealing with is an online party, such as Amazon.com. We need some form of negotiation through which both parties obtain the secret key, but no third party can figure it out.

There are different assumptions about which schemes are reliable. We can definitely rely on cryptography and mathematics, and we also might want to rely on certificate schemes. Within these certificate schemes, we must find a way of relying on authenticating a certificate.

Certificates establish public keys of the parties involved. For example, if Party B simply posted his public key, why should you believe him? Certificates, signed statements from some trusted third party affirming someone's public key, are used in many of these key exchange protocols.

The simplest protocol, which is what much of SSL is based on, is simply encrypting data with the other's public key. However, public key cryptography is slow, so we need to combine it with symmetric algorithms to make systems that work in practice.

[4]Key establishment in SSL: SSL and TLS: Designing and Building Secure System by Eric Rescorla. Available in reading room.

2.2 How does SSL work?

The SSL protocol is layered on top of TCP. There is the assumption of a reliable transport method.

Now, we describe the SSL protocol as diagramed in Figure 6.

SSL Protocol:

1. Client sends an unencrypted *ClientHello* message containing a list of possible cipher algorithms it can use, along with a random number to be used in the cipher algorithm.
2. Server replies with its own *ServerHello* message containing its choice from Client's list of cipher algorithms and its own random number.
3. Server sends a *Certificate* message, consisting of its public key signed by a third party which Client is presumed to trust.
4. Server sends a Done message.
5. Client specifies and sends premaster secret key, after it is encrypted with Server's public key in a *ClientKeyExchange* message.
6. Client sends a *ChangeCipher* message, indicating that further conversation will be encrypted.
7. Client sends a *HandshakeFinished* message.
8. Server sends a *ChangeCipher* message.
9. Server sends a *HandshakeFinished* message.

From this point on, every message is encrypted with the shared key K, which is determined from the premaster secret. *HandshakeFinished* messages contain the MAC of all previous messages to prevent an adversary from playing games in mid-transmission. One example of a man-in-the-middle attack is that an adversary could change the cipher negotiation messages to only accept broken ciphers (which would probably be supported by both server and client). Then encryption would take place normally, and without tampering, but would use a scheme that the adversary can break.

Question: Is there a reason for having the three separate messages going the same direction?
Answer: There are a number of options, you could pack them all into the same message. But maybe the server could specify other messages to implement custom options. Using separate messages has been found to be a convenient way to allow flexibilty. To get the same flexibility with one you would have to have a big message with multiple labeled parts.

Question: You mentioned RSA. Is that in the standard?
Answer: That you get to specify. The server can say, "I've got an RSA key," so the client sends the premaster key with RSA public key. There is a wide range of possibilities.

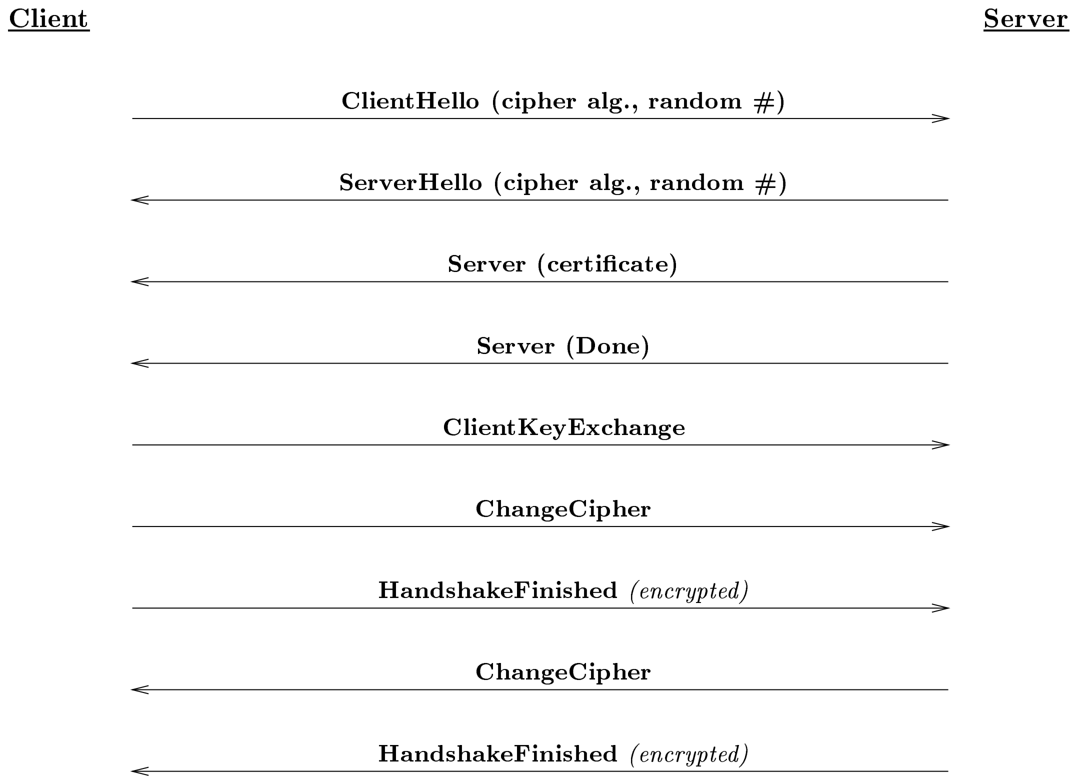

Figure 6: SSL protocol.

Question: Where in the graph do you actually send the premaster key?
Answer: In the *ClientKeyExchange* step of the protocol. The premaster key is really just random numbers. From the master secret you get the encryption key and authentication key.

Question: There's a seperate key in each direction?
Answer: Yes.

Question: When you need to set up authentication, can't you just use the server key every time?
Answer: That would be great for hardware acceleration manufacturers. It would be expensive, but it works for smaller applications. However, for larger things like webpages, the public key takes too long. Symmetric operations are much faster. Thus, you use the public key for security and then get a symmetric key to do the faster stuff. Perhaps one day public key operations will be able to compete with AES, but not at present.

2.3 Forward Secrecy

The framework we have laid out for SSL has a fundamental flaw. If the scheme is broken sometime in the future and the secret key is found, then an adversary who recorded messages passed between client and server can now decrypt the entire transaction.

What we want is a property known as *forward secrecy*. We'd like it to be the case that even if an adversary records all messages and later gets the private keys, she should not be able to decrypt recorded messages.

A modification to the Diffie-Hellman protocol includes a scheme to avoid this problem. Therefore, even though RSA is faster, it is better to use the modified Diffie-Hellman.

2.4 Diffie-Hellman

Suppose two parties want to setup a secret key K. Here we describe the Diffie-Hellman protocol to establish K as diagramed in Figure 7.

Original Diffie-Hellman Protocol:

1. Client picks a random $x \pmod p$. It computes $g^x \pmod p$ and sends it to Server.
2. Server symmetrically computes some y at random and sends back $g^y \pmod p$.
3. A key that both parties can agree on is $K = g^{xy} \pmod p$. Both parties can compute K. x and y are transient; they are erased from memory after K is computed.
4. Further traffic is encrypted with K. It's presumed the adversary can't get x and y from logs, and cannot compute g^{xy} from g^x and g^y.

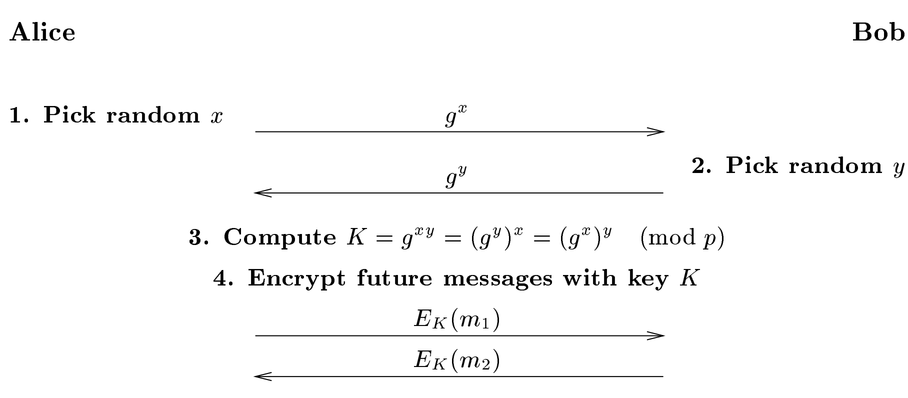

Figure 7: Original Diffie-Hellman protocol.

However, the protocol described above is not forward-secret. If an adversary can obtain x or y, he can compute the key K used to encrypt all of the messages.

Modified Diffie-Hellman Protocol:

In the modified scheme of Diffie-Hellman, both parties know who they are talking with and can establish an encryption key while maintaining forward secrecy. Since the long-term secrets, A and B, are only used to sign the initial key-exchange messages, even if A or B is discovered at some point in the future, the messages still cannot be decoded. Figure 8 below illustrates the steps involved in this modified protocol.

Alice — **with long-term secret** A

Bob — **with long-term secret** B

1. Pick random x

$$\xrightarrow{\quad g^x \quad}$$

2. Pick random y
Compute $K = g^{xy} \pmod p$

$$\xleftarrow{\quad g^y,\ \textbf{certificate},\ E_K(\sigma_B(g^y, g^x)) \quad}$$

3. Compute $K = g^{xy} \pmod p$

$$\xrightarrow{\quad \textbf{certificate},\ E_K(\sigma_A(g^x, g^y)) \quad}$$

4. Encrypt future messages with key K

$$\xleftrightarrow{\quad E_K(m) \quad}$$

...

Figure 8: Modified Diffie-Hellman protocol.

Because of forward secrecy, long term secrets are only used for signature keys. No long term secrets are used for encryptions. It is important that long term secrets are not used for encryption, only authentication.

We now have a forward-secure scheme in which A and B establish a shared secret and know who they are talking to. Even if the signing keys used by A or B are discovered, an adversary still will not be able to determine the transaction.

Question: Doesn't that mean that using a faster public key scheme (instead of a symmetric one, as mentioned previously in lecture) would be bad because it would stay around?

Answer: Yes, you have to be careful. It may be that your scheme has ephemeral public keys that are only good for one session.

Question: If you removed certificates, would this scheme then fail?

Answer: Good question. SSL is asymmetric; there is no certificate from client to server. The server does not need as high a level of authentication because it has your credit card number. Similarly here, if you did not care about authenticating A to B, you could eliminate a portion of the scheme. A would not be sure it was sharing the key only with Amazon, and B could not be completely sure who it was talking with.

Question: I don't understand why you have to encrypt the signature.
Answer: There are various attacks. Suppose we didn't have these encryptions here. You could mount a man in the middle attack. Very subtle, good midterm question. I'll answer it next lecture.

This protocol is not recommended because it has a minor defect. State of the art has advanced significantly since then and protocols have become more complex. Many protocols are proposed, and many get broken. It's hard to tell which ones will be broken. Sometimes it takes as long as a decade for someone to break a protocol.