CHAPTER 18

Certificates, Key Transport and Key Agreement

Chapter Goals

- To understand the problems associated with managing and distributing secret keys.
- To introduce the notion of digital certificates and a PKI.
- To show how an implicit certificate scheme can operate.
- To learn about key distribution techniques based on symmetric-key-based protocols.
- To introduce key transport based on public key encryption.
- To introduce Diffie–Hellman key exchange, and its various variations.
- To introduce the symbolic and computational analysis of protocols.

18.1. Introduction

We can now perform encryption and authentication using either public key techniques or symmetric key techniques. However, we have not addressed how parties actually obtain the public key of an entity, or obtain a shared symmetric key. When using hybrid ciphers in Section 16.3 we showed how a symmetric key could be transported to another party, and then that symmetric key used to encrypt a *single* message. However, we did not address what happens if we want to use the symmetric key many times, or use it for authentication etc. Nor did we address how the sender would know that the public key of the receiver he was using was genuine.

In this chapter we present methodologies to solve all of these problems. In doing so we present our first examples of what can be called "cryptographic protocols". A cryptographic protocol is an exchange of messages which achieves some cryptographic goal. Up until now we have created single shot mechanisms (encryption, MAC, signatures etc.) which have not required interaction between parties.

In dealing with cryptographic protocols we still need to define what we mean by something being secure, and when presenting a protocol we need to present a proof as to why we believe the protocol meets our security definition. However, unlike the proofs for encryption etc. that we have met before, the proofs for protocols are incredibly complex. They fall into one of two camps[1]:

- In the first camp are so-called "symbolic methods", these treat underlying primitives such as encryption as perfect black boxes and then try to show that a protocol is "secure". However, as they work at a very high level of abstraction they do not really prove security, they simply enable one to find attacks on a protocol relatively easily. This is because in this camp one shows a protocol is insecure by exhibiting an attack.
- The second camp is like our security games for encryption etc. We define a game, with an adversary. The adversary has certain powers (given by oracles), and has a certain goal. We then present proofs that the existence of such an adversary implies the existence of an algorithm which can solve a hard problem. These proofs provide a high degree of assurance

[1]There is a third camp called the *simulation paradigm*, typified by something called the Universal Composability (UC) framework. This camp is beyond the scope of this book however.

N.P. Smart, *Cryptography Made Simple*, Information Security and Cryptography, DOI 10.1007/978-3-319-21936-3_18

that the protocol is sound, and any security flaws will come from implementation aspects as opposed to protocol design issues.

Before we continue we need to distinguish between different types of keys. The following terminology will be used throughout this chapter and beyond:

- **Static (or Long-Term) Keys:** These are keys which are to be in use for a long time period. The exact definition of long will depend on the application, but this could mean from a few hours to a few years. The compromising of a static key is usually considered to be a major problem, with potentially catastrophic consequences.
- **Ephemeral, or Session (or Short-Term) Keys:** These are keys which have a short lifetime, maybe a few seconds or a day. They are usually used to provide confidentiality for a given time period. The compromising of a session key should only result in the compromising of that session's secrecy and it should not affect the long-term security of the system.

Key distribution is one of the fundamental problems of cryptography. There are a number of solutions to this problem; which solution one chooses depends on the overall system.

- **Physical Distribution:** Using trusted couriers or armed guards, keys can be distributed using traditional physical means. Until the 1970s this was in effect the only secure way of distributing keys at system setup. It has a large number of physical problems associated with it, especially scalability, but the main drawback is that security no longer rests with the key but with the courier. If we can bribe, kidnap or kill the courier then we have broken the system.
- **Distribution Using Symmetric Key Protocols:** Once some secret keys have been distributed between a number of users and a trusted central authority, we can use the trusted authority to help generate keys for any pair of users as the need arises. Protocols to perform this task will be discussed in this chapter. They are usually very efficient but have some drawbacks. In particular they usually assume that both the trusted authority and the two users who wish to agree on a key are both online. They also still require a physical means to set up the initial keys.
- **Distribution Using Public Key Protocols:** Using public key cryptography, two parties, who have never met or who do not trust any one single authority, can produce a shared secret key. This can be done in an online manner, using a key exchange protocol. Indeed this is the most common application of public key techniques for encryption. Rather than encrypting large amounts of data by public key techniques we agree a key by public key techniques and then use a symmetric cipher to actually do the encryption, a methodology we saw earlier when we discussed hybrid encryption.

To understand the scale of the problem, if our system is to cope with n separate users, and each user may want to communicate securely with any other user, then we require

$$\frac{n \cdot (n-1)}{2}$$

separate symmetric keys. This soon produces huge key management problems; a small university with around 10 000 students would need to have around fifty million separate secret keys.

With a large number of keys in existence one finds a large number of problems. For example what happens when your key is compromised? In other words, if someone else has found your key. What can you do about it? What can they do? Hence, a large number of keys produces a large key management problem.

One solution is for each user to hold only one key with which they communicate with a central authority, hence a system with n users will only require n keys. When two users wish to communicate, they generate a secret key which is only to be used for that message, a so-called session key or ephemeral key. This session key can be generated with the help of the central authority using one of the protocols that appear later in this chapter.

As we have mentioned already the main problem is one of managing the secure distribution of keys. Even a system which uses a trusted central authority needs some way of getting the keys shared between the centre and each user out to the user. One possible solution is key splitting (more formally called *secret sharing*) where we divide the key into a number of shares

$$k = k_1 \oplus k_2 \oplus \cdots \oplus k_r.$$

The shares are then distributed via separate routes. The beauty of this is that an attacker needs to attack all the routes so as to obtain the key. On the other hand attacking one route will stop the legitimate user from recovering the key. We will discuss secret sharing in more detail in Chapter 19.

One issue one needs to consider when generating and storing keys is the key lifetime. A general rule is that the longer the key is in use the more vulnerable it will be and the more valuable it will be to an attacker. We have already touched on this when mentioning the use of session keys. However, it is important to destroy keys properly after use. Relying on an operating system to delete a file by typing `del` or `rm` does not mean that an attacker cannot recover the file contents by examining the hard disk. Usually deleting a file does not destroy the file contents, it only signals that the file's location is now available for overwriting with new data. A similar problem occurs when deleting memory in an application.

This (rather lengthy) chapter is organized in the following sections. We first introduce the notion of certificates and a Public Key Infrastructure (PKI); such techniques allow parties to obtain authentic public keys. We then discuss a number of protocols which allow parties to agree new symmetric ephemeral keys, given already deployed static symmetric keys. Then we discuss how to obtain new symmetric ephemeral keys, given existing public keys (which can be authenticated via a PKI). Then to give a flavour of how protocols are analysed we present a simple symbolic method called BAN Logic and apply it to one of our symmetric-key-based protocols. We then give a detailed exposition of how one uses game style security definitions to show that the public-key-based protocols are secure. Note that we can use symbolic methods to analyse public-key-based techniques, and game style security to analyse symmetric-key-based techniques. However, for reasons of space we only give a limited set of examples.

18.2. Certificates and Certificate Authorities

When using a symmetric key system we assume we do not have to worry about which key belongs to which party. It is tacitly assumed that if Alice holds a long-term secret key K_{ab} which she thinks is shared with Bob, then Bob really does have a copy of the same key. This assurance is often achieved using a trusted physical means of long-term key distribution, for example using armed couriers.

In a public key system the issues are different. Alice may have a public key which she thinks is associated with Bob, but we usually do not assume that Alice is one hundred percent certain that it really belongs to Bob. This is because we do not, in the public key model, assume a physically secure key distribution system. After all, that was one of the points of public key cryptography in the first place: to make key management easier. Alice may have obtained the public key she thinks belongs to Bob from Bob's web page, but how does she know the web page has not been spoofed?

The process of linking a public key to an entity or principal, be it a person, machine or process, is called binding. One way of binding, common in many applications where the principal really does need to be present, is by using a physical token such as a smart card. Possession of the token, and knowledge of any PIN/password needed to unlock the token, is assumed to be equivalent to being the designated entity. This solution has a number of problems associated with it, since cards can be lost or stolen, which is why we protect them using a PIN (or in more important applications by using biometrics). The major problem is that most entities are non-human; they are computers and computers do not carry cards. In addition many public key protocols are performed over networks where physical presence of the principal (if they are human) is not something one can test.

Hence, some form of binding is needed which can be used in a variety of very different applications. The main binding tool in use today is the *digital certificate*. In this a special trusted third party, or TTP, called a certificate authority, or CA, is used to vouch for the validity of the public keys. A certificate authority system works as follows:

- All users have a trusted copy of the public key of the CA. For example these come embedded in your browser when you buy your computer, and you "of course" trust the vendor of the computer and the manufacturer of the software on your computer.
- The CA's job is to digitally sign data strings containing the following information

  (Alice, Alice's public key).

  This data string and the associated signature is called a digital certificate. The CA will only sign this data if it truly believes that the public key really does belong to Alice.
- When Alice now sends you her public key, contained in a digital certificate, you now trust that the purported key really is that of Alice, since you trust the CA to do its job correctly.

This use of a digital certificate binds the name "Alice" with the key. Public key certificates will typically (although not always) be stored in repositories and accessed as required. For example, most browsers keep a list of the certificates that they have come across. The digital certificates do not need to be stored securely since they cannot be tampered with as they are digitally signed.

To see the advantage of certificates and CAs in more detail consider the following example of a world without a CA. In the following discussion we break with our colour convention for a moment and now use red to signal public keys which must be obtained in an authentic manner and blue to signal public keys which do not need to be obtained in an authentic manner.

In a world without a CA you obtain many individual public keys from each individual in some authentic fashion. For example

6A5DEF....A21	Jim Bean's public key,
7F341A....BFF	Jane Doe's public key,
B5F34A....E6D	Microsoft's update key.

Hence, each key needs to be obtained in an authentic manner, as does every new key you obtain.

Now consider the world with a CA. You obtain a single public key in an authentic manner, namely the CA's public key. We shall call our CA Ted since he is Trustworthy. You then obtain many individual public keys, signed by the CA, in possibly an unauthentic manner. For example they could be attached at the bottom of an email, or picked up whilst browsing the web.

A45EFB....C45	Ted's totally trustworthy key,
6A5DEF....A21	Ted says "This is Jim Bean's public key",
7F341A....BFF	Ted says "This is Jane Doe's public key",
B5F34A....E6D	Ted says "This is Microsoft's update key".

If you trust Ted's key and you trust Ted to do his job correctly then you trust all the public keys you hold to be authentic.

In general a digital certificate is not just a signature on the single pair (Alice, Alice's public key); one can place all sorts of other, possibly application specific, information into the certificate. For example it is usual for the certificate to contain the following information.

- User's name,
- User's public key,

- Is this an encryption or signing key?
- Name of the CA,
- Serial number of the certificate,
- Expiry date of the certificate,
- . . . .

Commercial certificate authorities exist that will produce a digital certificate for your public key, often after payment of a fee and some checks on whether you are who you say you are. The certificates produced by commercial CAs are often made public, so one could call them public "public key certificates", in that their use is mainly over open public networks. CAs are also used in proprietary closed systems, for example in debit/credit card systems or by large corporations.

It is common for more than one CA to exist. A quick examination of the properties of your web browser will reveal a large number of certificate authorities which your browser assumes you "trust" to perform the function of a CA. As there is more than one CA it is common for one CA to sign a digital certificate containing the public key of another CA, and vice versa, a process which is known as cross-certification.

Cross-certification is needed if more than one CA exists, since a user may not have a trusted copy of the CA's public key needed to verify another user's digital certificate. This is solved by cross-certificates, i.e. one CA's public key is signed by another CA. The user first verifies the appropriate cross-certificate, and then verifies the user certificate itself.

With many CAs one can get quite long certificate chains, as Figure 18.1 illustrates. Suppose Bob trusts the Root CA's public key and he obtains Alice's public key which is signed by the private key of CA3. He then obtains CA3's public key, either along with Alice's digital certificate or by some other means. CA3's public key comes in a certificate which is signed by the private key of CA1. Bob then needs to obtain the public key of CA1, which will be contained in a certificate signed by the Root CA. Hence, by verifying all the signatures he ends up trusting Alice's public key.

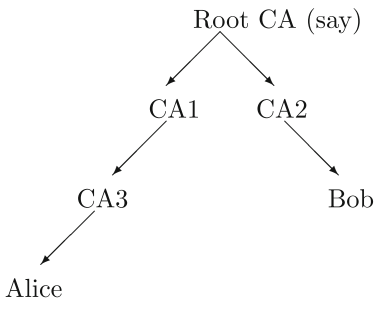

FIGURE 18.1. Example certification hierarchy

Often the function of a CA is split into two parts. One part deals with verifying the user's identity and one part actually signs the public keys. The signing is performed by the CA, whilst the identity of the user is parcelled out to a registration authority, or RA. This can be a good practice, with the CA implemented in a more secure environment to protect the long-term private key.

The main problem with a CA system arises when a user's public key is compromised or becomes untrusted for some reason. For example

- A third party has gained knowledge of the private key,
- An employee leaves the company.

As the public key is no longer to be trusted all the associated digital certificates are now invalid and need to be revoked. But these certificates can be distributed over a large number of users, each one of which needs to be told to no longer trust this certificate. The CA must somehow inform all users that the certificate(s) containing this public key is/are no longer valid, in a process called certificate revocation.

One way to accomplish this is via a Certificate Revocation List, or CRL, which is a signed statement by the CA containing the serial numbers of all certificates which have been revoked by that CA and whose validity period has not expired. One clearly need not include in this the serial numbers of certificates which have passed their expiry date. Users must then ensure they have the latest CRL. This can be achieved by issuing CRLs at regular intervals even if the list has not changed. Such a system can work well in a corporate environment when overnight background jobs are often used to make sure each desktop computer in the company is up to date with the latest software etc. For other situations it is hard to see how the CRLs can be distributed, especially if there are a large number of CAs trusted by each user.

The whole system of CAs and certificates is often called the Public Key Infrastructure, or PKI. This essentially allows a distribution of trust; the need to trust the authenticity of each individual public key in your possession is replaced by the need to trust a body, the CA, to do its job correctly.

18.2.1. Implicit Certificates: One issue with digital certificates is that they can be rather large. Each certificate needs to at least contain both the public key of the user and the signature of the certificate authority on that key. This can lead to quite large certificate sizes, as the following table demonstrates:

	RSA	DSA	EC-DSA
User's key	2024	2048	256
CA sig	2024	512	512

This assumes that for RSA keys one uses a 2048-bit modulus, for DSA one uses a 2048-bit prime p and a 256-bit prime q and for EC-DSA one uses a 256-bit curve. Hence, for example, if the CA is using 2048-bit RSA and they are signing the public key of a user using 2048-bit DSA then the total certificate size must be at least 4096 bits.

Implicit certificates enable these sizes to be reduced somewhat. An implicit certificate looks like $X|Y$ where

- X is the data being bound to the public key,
- Y is the implicit certificate on X.

From Y we need to be able to recover the public key being bound to X and implicit assurance that the certificate was issued by the CA. In the system we describe below, based on a DSA or EC-DSA, the size of Y will be 2048 or 256 bits respectively. Hence, the size of the certificate is reduced to the size of the public key being certified.

System Set-up: The CA chooses a public group G of known order n and an element $P \in G$. The CA then chooses a long-term private key c and computes the public key $Q \leftarrow P^c$. This public key should be known to all users.

Certificate Request: Suppose Alice wishes to request a certificate and the public key associated with the information ID, which could be her name. Alice computes an ephemeral secret key t and an ephemeral public key $R \leftarrow P^t$. Alice sends R and ID to the CA.

Processing of the Request: The CA checks that he wants to link ID with Alice. The CA picks another random number k and computes

$$g \leftarrow P^k R = P^k P^t = P^{k+t}.$$

Then the CA computes $s \leftarrow cH(ID\|g) + k \pmod{n}$. Then the CA sends back to Alice the pair (g, s). The implicit certificate is the pair (ID, g). We now have to convince you that, not only can Alice recover a valid public/private key pair, but also any other user can recover Alice's public key from this implicit certificate.

Alice's Key Discovery: Alice knows the following information: $t, s, R = P^t$. From this she can recover her private key via $a \leftarrow t + s \pmod{n}$. Note that Alice's private key is known only to Alice and not to the CA. In addition Alice has contributed some randomness t to her private key, as has the CA who contributed k. Her public key is then $P^a = P^{t+s} = P^t P^s = R \cdot P^s$.

User's Key Discovery: Since s and R are public, a user, say Bob, can recover Alice's public key from the above message flows via $R \cdot P^s$. But this says nothing about the linkage between the CA, Alice's public key and the ID information. Instead Bob recovers the public key from the implicit certificate (ID, g) and the CA's public key Q via the equation $P^a = Q^{H(ID\|g)} g$.

As soon as Bob sees Alice's key used in action, say he verifies a signature purported to have been made by Alice, he knows implicitly that it must have been issued by the CA, since otherwise Alice's signature would not verify correctly.

There are a number of problems with the above system which mean that implicit certificates are not used much in real life. For example:

(1) What do you do if the CA's key is compromised? Usually you pick a new CA key and re-certify the user's keys. But you cannot do this since the user's public key is chosen interactively during the certification process.
(2) Implicit certificates require the CA and users to work at the same security level. This is not considered good practice, as usually one expects the CA to work at a higher security level (say 4096-bit DSA) than the user (say 2048-bit DSA).

However for devices with restricted bandwidth implicit certificates can offer a suitable alternative where traditional certificates are not viable.

18.3. Fresh Ephemeral Symmetric Keys from Static Symmetric Keys

Recall that if we have n users each pair of whom wishes to communicate securely with each other then we would require

$$\frac{n \cdot (n-1)}{2}$$

separate static symmetric key pairs. As remarked earlier this leads to huge key management problems and issues related to the distribution of the keys. We have already mentioned that it is better to use session keys and few long-term keys, but we have not explained how one deploys the session keys.

To solve this problem a number of protocols which make use of symmetric key cryptography to distribute secret session keys have been developed, some of which we shall describe in this section. Later we shall look at public key techniques for this problem, which are often more elegant. We first need to set up some notation to describe the protocols. Firstly we set up the names of the parties and quantities involved.

- **Parties/Principals:** A, B, S.
  Assume the two parties who wish to agree a secret are A and B, for Alice and Bob. We assume that they will use a trusted third party, or TTP, which we shall denote by S.
- **Shared Secret Keys:** K_{ab}, K_{bs}, K_{as}.
  K_{ab} will denote a secret key known only to A and B.
- **Nonces:** N_a, N_b.
  Just as in Chapter 13 nonces are numbers used only once; they do not need to be random, just unique. The quantity N_a will denote a nonce originally produced by the principal A.

Note that other notations for nonces are possible and we will introduce them as the need arises.

- **Timestamps:** T_a, T_b, T_s.
  The quantity T_a is a timestamp produced by A. When timestamps are used we assume that the parties try to keep their clocks in synchronization using some other protocol.

The statement

$$A \longrightarrow B : M, A, B, \{N_a, M, A, B\}_{K_{as}}$$

means A sends to B the message to the right of the colon. The message consists of

- A nonce M,
- A the name of party A,
- B the name of party B,
- A message $\{N_a, M, A, B\}$ encrypted under the key K_{as} which A shares with S. Hence, the recipient B is unable to read the encrypted part of this message.

18.3.1. Wide-Mouth Frog Protocol: Our first protocol is the Wide-Mouth Frog protocol, which is a simple protocol invented by Burrows. The protocol transfers a key K_{ab} from A to B via S; it uses only two messages but has a number of drawbacks. In particular it requires the use of synchronized clocks, which can cause a problem in implementations. In addition the protocol assumes that A chooses the session key K_{ab} and then transports this key to user B. This implies that user A is trusted by user B to be competent in making and keeping keys secret. This is a very strong assumption and the main reason that this protocol is not used much in real life. However, it is very simple and gives a good example of how to analyse a protocol formally, which we shall come to later in this chapter. The protocol proceeds in the following steps, as illustrated in Figure 18.2,

$$A \longrightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}.$$

On obtaining the first message the trusted third party S decrypts the last part of the message and checks that the timestamp is recent. This decrypted message tells S it should forward the key to the party called B. If the timestamp is verified to be recent, S encrypts the key along with its timestamp and passes this encryption on to B. On obtaining this message B decrypts the message received and checks the time stamp is recent, then he can recover both the key K_{ab} and the name A of the person who wants to send data to him using this key. The checks on the timestamps mean the session key should be recent, in that it left user A a short time ago. However, user A could have generated this key years ago and stored it on her hard disk, in which time Eve broke in and took a copy of this key.

We already said that this protocol requires that all parties need to keep synchronized clocks. However, this is not such a big problem since S checks or generates all the timestamps used in the protocol. Hence, each party only needs to record the difference between its clock and the clock owned by S. Clocks are then updated if a clock drift occurs which causes the protocol to fail. This protocol is really too simple; much of the simplicity comes by assuming synchronized clocks and by assuming party A can be trusted with creating session keys.

18.3.2. Needham–Schroeder Protocol: We shall now look at more complicated protocols, starting with one of the most famous, namely the Needham–Schroeder protocol. This protocol was developed in 1978, and is one of most highly studied protocols ever; its fame is due to the fact that even a simple protocol can hide security flaws for a long time. We shall build the protocol up slowly, starting with a simple, obviously insecure, protocol and then addressing each issue we find until we reach the final protocol. Our goal is to come up with a protocol which does not require synchronized clocks.

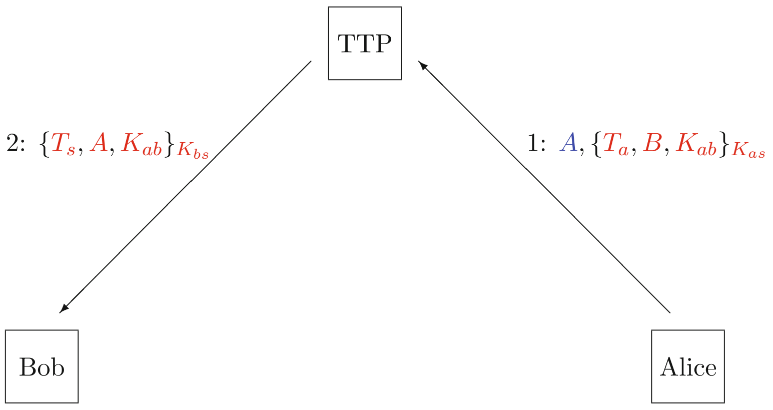
TTP
Bob
Alice
2: $\{T_s, A, K_{ab}\}_{K_{bs}}$
1: $A, \{T_a, B, K_{ab}\}_{K_{as}}$

Figure 18.2. Wide-Mouth Frog protocol

In all of our analysis we assume that the attacker has complete control of the network; she can delay, replay and delete messages. She can even masquarade as legitimate entities, until those entities prove who they are via cryptographic means. This model of the network and attacker is called the Dolev–Yao model.

Our first simple protocol is given by the following message flows, and is illustrated in Figure 18.3,

$$\begin{aligned} A &\longrightarrow S : A, B, \\ S &\longrightarrow A : K_{ab}, \\ A &\longrightarrow B : K_{ab}, A. \end{aligned}$$

In this protocol the trusted server S generates the key, after it is told by Alice to generate a key

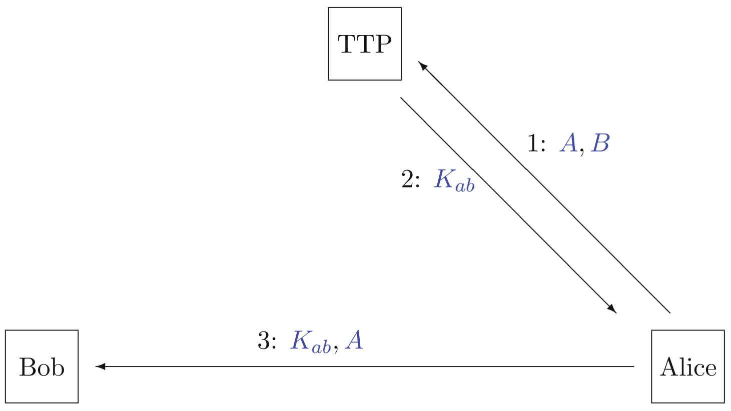

Figure 18.3. Protocol Version 1

for use between Alice and Bob. At the end of the protocol Bob is told by Alice that the key K_{ab} is one he should use for communication with Alice. However, it is immediately obvious that this protocol is not secure, since any eavesdropper can learn the secret key K_{ab}, since it is transmitted unencrypted.

So our first modification is to create a protocol which enables the key to remain secret, however we need to do this utilizing only the long-term static secret keys K_{as} and K_{bs}. With this modification our protocol is formed of the following message flows, and is illustrated in Figure 18.4,

$$\begin{aligned} A &\longrightarrow S : A, B, \\ S &\longrightarrow A : \{K_{ab}\}_{K_{bs}}, \{K_{ab}\}_{K_{as}}, \\ A &\longrightarrow B : \{K_{ab}\}_{K_{bs}}, A. \end{aligned}$$

This is slightly better, but now we notice that the attacker can take the last message from Alice

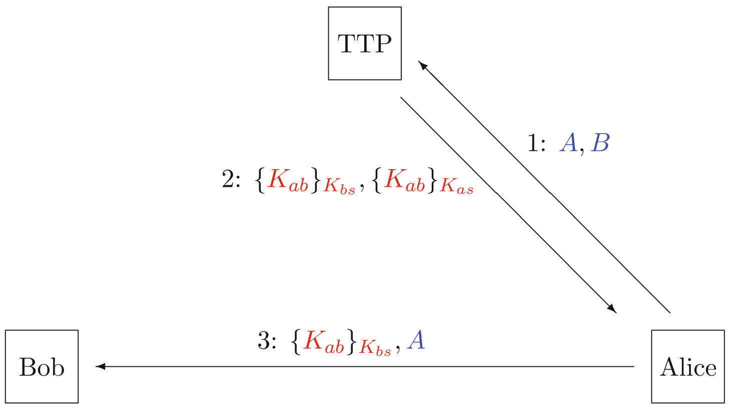
TTP
1: A, B
2: $\{K_{ab}\}_{K_{bs}}, \{K_{ab}\}_{K_{as}}$
3: $\{K_{ab}\}_{K_{bs}}, A$
Bob
Alice

FIGURE 18.4. Protocol Version 2

to Bob, and replace it with $\{K_{ab}\}_{K_{bs}}, D$. This means that Bob will think that the key K_{ab} is for communicating with Diana and not Alice. Thus we do not have the property that Bob knows to whom he is sending information. Imagine that Bob is in love with Diana and so encrypts a message under K_{ab} saying "I love you". This message can only be decrypted by Alice, so the adversary now redirects the ciphertext to Alice. Alice decrypts the message and apparently finds out that Bob is in love with her. We easily see that this could cause problems for both Alice, Bob and Diana.

A more fundamental problem with our second protocol attempt is that there is a man-in-the-middle attack. In the following message flows, the attacker Eve (E), masquerades as both the TTP and Bob to Alice, and so is able to learn the key that Alice thinks she is using to communicate solely with Bob.

$$\begin{aligned} A &\longrightarrow E : A, B, \\ E &\longrightarrow S : A, E, \\ S &\longrightarrow E : \{K_{ae}\}_{K_{es}}, \{K_{ae}\}_{K_{as}}, \\ E &\longrightarrow A : \{K_{ae}\}_{K_{es}}, \{K_{ae}\}_{K_{as}}, \\ A &\longrightarrow E : \{K_{ae}\}_{K_{es}}, A. \end{aligned}$$

To get around these two problems we get the TTP S to encrypt the identity components, thus preventing both of the above problems. Thus our third attempt at a protocol is given by Figure 18.5 and the following message flows:

$$\begin{aligned} A &\longrightarrow S : A, B, \\ S &\longrightarrow A : \{K_{ab}, A\}_{K_{bs}}, \{K_{ab}, B\}_{K_{as}}, \\ A &\longrightarrow B : \{K_{ab}, A\}_{K_{bs}}. \end{aligned}$$

This protocol however suffers from a replay attack. Assume the attacker can determine the key

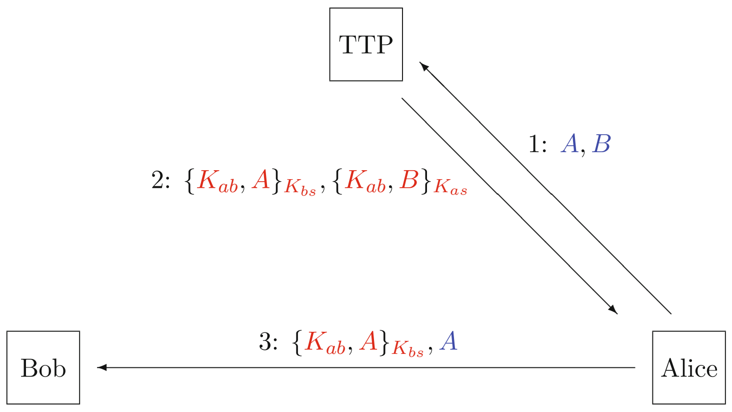
TTP
1: A, B
2: $\{K_{ab}, A\}_{K_{bs}}, \{K_{ab}, B\}_{K_{as}}$
3: $\{K_{ab}, A\}_{K_{bs}}, A$
Bob
Alice

FIGURE 18.5. Protocol Version 3

K_{ab} used in an old run of the protocol; this might be because users and systems are often not as careful with respect to keys meant for short-term use (after all they *are* for short-term use!). She can then masquerade as the TTP and deliver the same responses to a new run of a protocol as to the old run, and she can do this without needing to know either K_{as} or K_{bs}.

The basic problem is that Alice does not know whether the key she receives is fresh. In the Wide-Mouth Frog protocol we guaranteed freshness by the use of synchronized clocks; we are trying to avoid them in the design of this protocol. We avoid the need to use synchronized clocks by instead utilizing nonces. This leads us to our fourth protocol attempt, given by Figure 18.6 and the following message flows:

$$\begin{aligned} A &\longrightarrow S : A, B, N_a, \\ S &\longrightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}, \\ A &\longrightarrow B : \{K_{ab}, A\}_{K_{bs}}, \end{aligned}$$

By Alice checking the nonce N_a received in the response from S is the same as that sent in the initial message, Alice knows that the key K_{ab} will be fresh; assuming that S is following the protocol. In this fourth variant Bob knows that Alice once was alive and sent the same ephemeral symmetric key K_{ab} as he just received. But this could have been a long while in the past. Similarly, Alice knows that only Bob could have access to K_{ab}, but she does not know whether he does or not. Thus our last modification is to add a key confirmation step, which assures both Alice and Bob that they are both alive and are able to use the key. This is achieved by adding an encrypted nonce sent by Bob, which Alice decrypts, modifies and sends back to Bob encrypted. With this change we get the Needham–Schroeder protocol, see Figure 18.7, and the message flows:

$$\begin{aligned} A &\longrightarrow S : A, B, N_a, \\ S &\longrightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}, \\ A &\longrightarrow B : \{K_{ab}, A\}_{K_{bs}}, \\ B &\longrightarrow A : \{N_b\}_{K_{ab}}, \\ A &\longrightarrow B : \{N_b - 1\}_{K_{ab}}. \end{aligned}$$

We now look again at each message in detail, and summarize what it achieves:

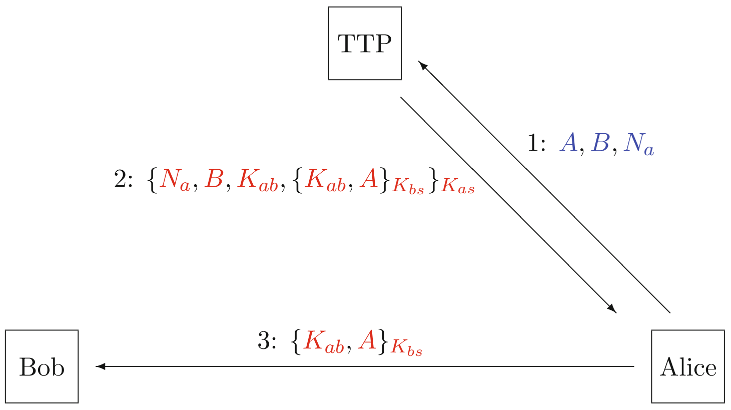
TTP
1: A, B, N_a
2: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3: $\{K_{ab}, A\}_{K_{bs}}$
Bob
Alice

FIGURE 18.6. Protocol Version 4

- The first message tells S that Alice wants a key to communicate with Bob.
- In the second message S generates the session key K_{ab} and sends it back to Alice. The nonce N_a is included so that Alice knows this was sent after her request of the first message. The session key is also encrypted under the key K_{bs} for sending to Bob.
- The third message conveys the session key to Bob.
- Bob needs to check that the third message was not a replay. So he needs to know whether Alice is still alive; hence, in the fourth message he encrypts a nonce back to Alice.
- In the final message, to prove to Bob that she is still alive, Alice encrypts a simple function of Bob's nonce back to Bob.

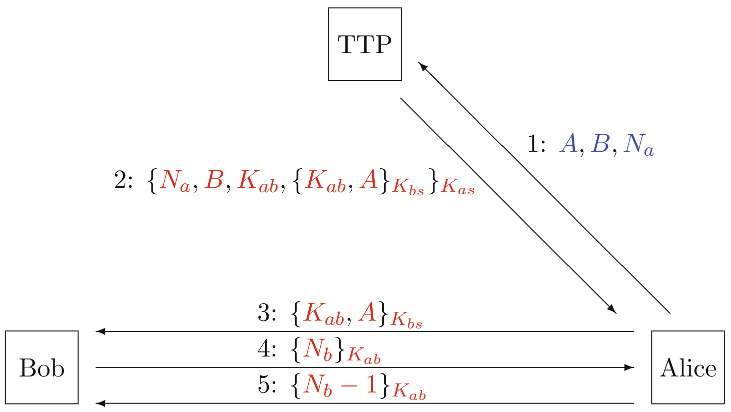
TTP
1: A, B, N_a
2: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3: $\{K_{ab}, A\}_{K_{bs}}$
4: $\{N_b\}_{K_{ab}}$
5: $\{N_b - 1\}_{K_{ab}}$
Bob
Alice

FIGURE 18.7. Needham–Schroeder protocol

The main problem with the Needham–Schroeder protocol is that Bob does not know that the key he shares with Alice is fresh, a fact which was not spotted until some time after the original protocol was published. An adversary who finds an old session transcript can, after finding the old session key by some other means, use the old session transcript in the last three messages involving Bob. Hence, the adversary can get Bob to agree to a key with the adversary, which Bob thinks he is sharing with Alice.

Note that Alice and Bob have their secret session key generated by the TTP and so neither party needs to trust the other to produce "good" keys. They of course trust the TTP to generate good keys since the TTP is an authority trusted by everyone. In some applications this last assumption is not valid and more involved algorithms, or public key algorithms, are required.

18.3.3. Kerberos: We end this section by looking at Kerberos. Kerberos is an authentication system based on symmetric encryption, with keys shared with an authentication server; it is based on ideas underlying the Needham–Schroeder protocol. Kerberos was developed at MIT around 1987 as part of Project Athena. A modified version of this original version of Kerberos is now used in many versions of the Windows operating system, and in many other systems.

The network is assumed to consist of clients and a server, where the clients may be users, programs or services. Kerberos keeps a central database of clients including a secret key for each client, hence Kerberos requires a key space of size $O(n)$ if we have n clients. Kerberos is used to provide authentication of one entity to another and to issue session keys to these entities.

In addition Kerberos can run a ticket-granting system to enable access control to services and resources. This division mirrors what happens in real companies. For example, in a company the personnel department administers who you are, whilst the computer department administers what resources you can use. This division is also echoed in Kerberos with an authentication server and a ticket generation server TGS. The TGS grants tickets to enable users to access resources, such as files, printers, etc.

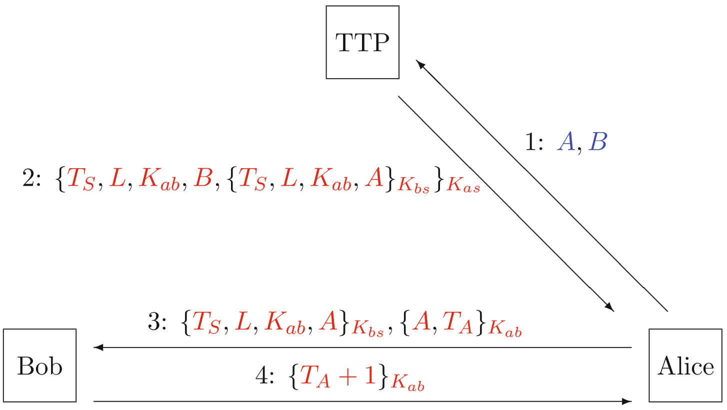
TTP
1: A, B
2: $\{T_S, L, K_{ab}, B, \{T_S, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3: $\{T_S, L, K_{ab}, A\}_{K_{bs}}, \{A, T_A\}_{K_{ab}}$
Bob
Alice
4: $\{T_A + 1\}_{K_{ab}}$

FIGURE 18.8. Kerberos

Suppose A wishes to access a resource B. First A logs in to the authentication server using a password. The user A is given a ticket from this server encrypted under her password. This ticket contains a session key K_{as}. She now uses K_{as} to obtain a ticket from the TGS S to access the resource B. The output of the TGS is a key K_{ab} , a timestamp T_S and a lifetime L. The output of the TGS is used to authenticate A in subsequent traffic with B. The flows look something like those given in Figure 18.8,

$$\begin{aligned}
A &\longrightarrow S : A, B,\\
S &\longrightarrow A : \{T_S, L, K_{ab}, B, \{T_S, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}},\\
A &\longrightarrow B : \{T_S, L, K_{ab}, A\}_{K_{bs}}, \{A, T_A\}_{K_{ab}},\\
B &\longrightarrow A : \{T_A + 1\}_{K_{ab}}.
\end{aligned}$$

Again we describe what each message flow is trying to achieve:

- The first message is A telling S that she wants to access B.
- If S allows this access then a ticket $\{T_S, L, K_{ab}, A\}$ is created. This is encrypted under K_{bs} and sent to A for forwarding to B. The user A also gets a copy of the key in a form readable by her.
- The user A wants to verify that the ticket is valid and that the resource B is alive. Hence, she sends an encrypted nonce/timestamp T_A to B.
- The resource B sends back the encryption of $T_A + 1$, after checking that the timestamp T_A is recent, thus proving that he knows the key and is alive.

Thus we have removed the problems associated with the Needham–Schroeder protocol by using timestamps, but this has created a requirement for synchronized clocks.

18.4. Fresh Ephemeral Symmetric Keys from Static Public Keys

Recall that the main drawback with the use of fast bulk encryption based on block or stream ciphers was the problem of key distribution. We have already seen a number of techniques to solve this problem, using protocols which are themselves based on symmetric key techniques. These, however, also have problems associated with them. For example, the symmetric key protocols required the use of already deployed long-term keys between each user and a trusted central authority. In this section we look at two public-key-based techniques. The first, called *key transport*, uses public key encryption to transmit a symmetric key from one user to the other; the second, called *key agreement*, is a protocol which as output produces a symmetric key, and which uses public key signatures to authenticate the parties.

18.4.1. Key Transport: Let $(e_{\mathfrak{pk}}, d_{\mathfrak{sk}})$ be a public key encryption scheme with public/private key pair $(\mathfrak{pk}, \mathfrak{sk})$ associated with a user Bob, via a certificate. Suppose Alice wants to send a symmetric key over to Bob, she first looks up Bob's public key in a directory, she generates the required symmetric key K_{ab} from the required space and then encrypts this and sends it to Bob. This is a very simple scheme whose flow is given pictorially in Figure 18.9 and in symbols by

$$A \longrightarrow B : e_{\mathfrak{pk}}(K_{ab}).$$

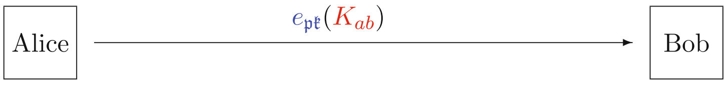

FIGURE 18.9. Public-key-based key transport: Version 1

This protocol is very much like the first part of a hybrid encryption scheme. However, it does not achieve all that we require. Firstly, whilst Alice knows that only Bob can decrypt the ciphertext to obtain the new key K_{ab}, Bob does not know that the ciphertext came from Alice. So Bob does not know to whom the key K_{ab} should be associated. One way around this is for Alice to also append a digital signature to the ciphertext. So now we have two public/private key pairs: a signature pair for Alice $(\mathfrak{pk}_A, \mathfrak{sk}_A)$ for some public key signature algorithm Sig, and an encryption pair for Bob $(\mathfrak{pk}_B, \mathfrak{sk}_B)$ for some encryption algorithm $e_{\mathfrak{pk}_B}$. The resulting protocol flow is given in Figure 18.10. However, this protocol suffers from another weakness; we can get Bob to think he is sharing a key with Eve, when he is actually sharing a key with Alice. The attack goes as follows:

$$A \longrightarrow E : c := e_{\mathfrak{pk}_B}(K_{ab}), \mathsf{Sig}_{\mathfrak{sk}_A}(c)$$
$$E \longrightarrow B : c := e_{\mathfrak{pk}_B}(K_{ab}), \mathsf{Sig}_{\mathfrak{sk}_E}(c).$$

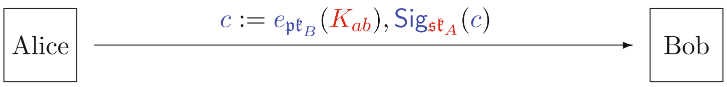
Alice
$c := e_{\mathfrak{pk}_B}(K_{ab}), \mathsf{Sig}_{\mathfrak{sk}_A}(c)$
Bob

FIGURE 18.10. Public-key based key transport: Version 2

This works because the identity of the sender is not bound to the ciphertext. Following the lead from our examples in Section 18.3, the simple solution to this problem is to encrypt the sender's identity along with the transmitted key, as in Figure 18.11. But even this cannot be considered secure due to a replay attack which we will outline later.

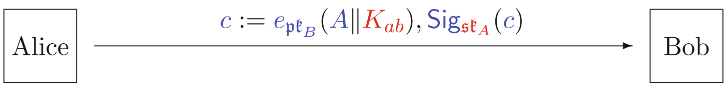
Alice
$c := e_{\mathfrak{pk}_B}(A \| K_{ab}), \mathsf{Sig}_{\mathfrak{sk}_A}(c)$
Bob

FIGURE 18.11. Public-key based key transport: Version 3

Forward Secrecy: All of the systems based on symmetric key encryption given in Section 18.3, and the previous method of key transport using public key-encryption-based key transport are not *forward secure*. A system is said to have forward secrecy if compromising of a long-term private key, i.e. $\mathfrak{sk}_B$ in the above protocol, at some point, in the future does not compromise the security of communications made using that key in the past. Notice in Figure 18.11 that if the recipient's private key $\mathfrak{sk}_B$ is compromised in the future, then all communications in the past are also compromised.

In addition using key transport implies that the recipient trusts the sender to be able to generate, in a sensible way, the session key. Sometimes the recipient may wish to contribute some randomness of their own to the session key. However, this can only be done if both parties are online at the same moment in time. Key transport is thus more suited to the case where only the sender is online, as in applications like email, for example.

The idea of both parties contributing to the entropy of the session key not only aids in creating perfectly secure schemes, it also avoids the replay attack we have on our key transport protocol. The next set of protocols, based on Diffie–Hellman key exchange, does indeed contribute entropy from both sides.

18.4.2. Diffie–Hellman Key Exchange: To avoid the fact that key transport based on public key encryption is not forward secure, and the problem of one party generating the key, the modern way of using public key techniques to create symmetric keys between two parties is based on a process called *key exchange*.

Key exchange was introduced in the same seminal paper by Diffie and Hellman in which they introduced public key cryptography. Their protocol for key distribution, called *Diffie–Hellman key exchange*, allows two parties to agree a secret key over an insecure channel without having met before. Its security is based on the discrete logarithm problem in a finite abelian group G of prime order q. In the original paper the group is taken to be a subgroup G of $\mathbb{F}_p^*$, but now more efficient versions can be produced by taking G to be a subgroup of an elliptic curve, in which case the protocol is called EC-DH.

In Diffie–Hellman the two parties each have their own ephemeral secrets a and b, which are elements in the group $\mathbb{Z}/q\mathbb{Z}$. The basic message flows for the Diffie–Hellman protocol are given by

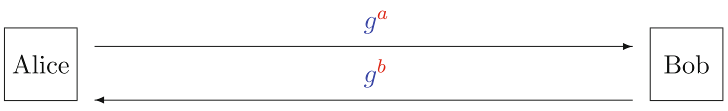

FIGURE 18.12. Diffie–Hellman key exchange

Figure 18.12 and the following notational representation:

$$A \longrightarrow B : \mathfrak{ek}_A = g^a,$$
$$B \longrightarrow A : \mathfrak{ek}_B = g^b.$$

From these exchanged ephemeral key values, $\mathfrak{ek}_A$ and $\mathfrak{ek}_B$, and their respective ephemeral secret keys, a and b, both parties can agree on the same secret session key:

- Alice can compute $K \leftarrow {\mathfrak{ek}_B}^a = (g^b)^a$, since she knows a and was sent $\mathfrak{ek}_B = g^b$ by Bob,
- Bob can also compute $K \leftarrow {\mathfrak{ek}_A}^b = (g^a)^b$, since he knows b and was sent $\mathfrak{ek}_A = g^a$ by Alice.

Eve, the attacker, can see the messages g^a and g^b and then needs to recover the secret key $K = g^{a \cdot b}$ which is exactly the Diffie–Hellman problem considered in Chapter 3. Hence, the security of the above protocol rests not on the difficulty of solving the discrete logarithm problem, DLP, but on the difficulty of solving the Diffie–Hellman problem, DHP. Recall that it may be the case that it is easier to solve the DHP than the DLP, although no one believes this to be true for the groups that are currently used in real-life protocols.

In practice one does not want the agreed key to be an element of a group, one requires it to be a bit string of a given length. Hence, in real systems one applies a key derivation function to the agreed secret group element, to obtain the required key for future use. It turns out that not only is this operation important for functional reasons, but in addition when we model the KDF as a random oracle one can prove the security of Diffie–Hellman-related protocols in the random oracle model. Thus we almost always use the derived key $k = H(K)$, for some random oracle H.

Notice that the Diffie–Hellman protocol can be performed both online (in which case both parties contribute to the randomness in the shared session key) or offline, where one of the parties uses a long-term key of the form g^a instead of an ephemeral key. Hence, the Diffie–Hellman protocol can be used as a key exchange or as a key transport protocol. We shall focus however on its use as a key exchange protocol.

Finite Field Example: The following is a very small example; we let the domain parameters be given by

$$p = 2\,147\,483\,659, \quad q = 2\,402\,107, \quad \text{and } g = 509\,190\,093,$$

but in real life one would take $p \approx 2^{2048}$. Note that g has prime order q in the field $\mathbb{F}_p$. The following diagram indicates a possible message flow for the Diffie–Hellman protocol:

$$\begin{array}{lcl}
\text{Alice} & & \text{Bob} \\
a = 12\,345 & & b = 654\,323, \\
\mathfrak{ek}_A = g^a = 382\,909\,757 & \longrightarrow & \mathfrak{ek}_A = 382\,909\,757, \\
\mathfrak{ek}_B = 1\,190\,416\,419 & \longleftarrow & \mathfrak{ek}_B = g^b = 1\,190\,416\,419.
\end{array}$$

The shared secret group element is then computed via

$${\mathfrak{ek}_A}^b = 382\,909\,757^{654\,323} \pmod{p} = 881\,311\,606,$$
$${\mathfrak{ek}_B}^a = 1\,190\,416\,419^{12\,345} \pmod{p} = 881\,311\,606,$$

with the actual secret key being given by $k = H(881\,311\,606)$ for some KDF H, which we model as a random oracle.

Notice that group elements are transmitted in the protocol, hence when using a finite field such as $\mathbb{F}_p^*$ for the Diffie–Hellman protocol the communication costs are around 2048 bits in each direction, since it is prudent to choose $p \approx 2^{2048}$. However, when one uses an elliptic curve group $E(\mathbb{F}_q)$ one can choose $q \approx 2^{256}$, and so the communication costs are much less, namely around 256 bits in each direction. In addition the group exponentiation step for elliptic curves can be done more efficiently than that for finite prime fields.

Elliptic Curve Example: As a baby example of EC-DH consider the elliptic curve

$$E : Y^2 = X^3 + X - 3$$

over the field $\mathbb{F}_{199}$. Let the base point be given by $G = (1, 76)$, which has prime order $q = \#E(\mathbb{F}_{199}) = 197$. Then a possible EC-DH message flow is given by

$$\begin{array}{lcl} \text{Alice} & & \text{Bob} \\ a = 23 & & b = 86, \\ \mathfrak{ek}_A = [a]G = (2, 150) & \longrightarrow & \mathfrak{ek}_A = (2, 150), \\ \mathfrak{ek}_B = (123, 187) & \longleftarrow & \mathfrak{ek}_B = [b]G = (123, 187). \end{array}$$

The shared secret key is then computed via

$$[b]\mathfrak{ek}_A = [86](2, 150) = (156, 75),$$
$$[a]\mathfrak{ek}_B = [23](123, 187) = (156, 75).$$

The shared key is then usually taken to be the x-coordinate 156 of the computed point. In addition, instead of transmitting the points, we transmit the compression of the point, which results in a significant saving in bandwidth.

18.4.3. Signed Diffie–Hellman: So we seem to have solved the key distribution problem. But there is an important problem: you need to be careful *who* you are agreeing a key with. Alice has no assurance that she is agreeing a key with Bob, which can lead to the following man-in-the-middle attack:

$$\begin{array}{ccccc} \text{Alice} & & \text{Eve} & & \text{Bob} \\ a & \longrightarrow & g^a, & & \\ g^m & \longleftarrow & m, & & \\ g^{am} & & g^{am}, & & \\ & & n & \longrightarrow & g^n, \\ & & g^b & \longleftarrow & b, \\ & & g^{bn} & & g^{bn}. \end{array}$$

In the man-in-the-middle attack

- Alice agrees a key with Eve, thinking it is Bob with whom she is agreeing a key with,
- Bob agrees a key with Eve, thinking it is Alice,
- Eve can now examine communications as they pass through her i.e. she acts as a router. She does not alter the plaintext, so her actions go undetected.

So we can conclude that the Diffie–Hellman protocol on its own is not enough. For example how does Alice know with whom she is agreeing a key? Is it Bob or Eve? One way around the man-in-the-middle attack on the Diffie–Hellman protocol is for Alice to sign her message to Bob and Bob to sign his message to Alice. In this way both parties know who they are talking to. This produces the protocol called *signed-Diffie–Hellman* given in Figure 18.13 and with message flows:

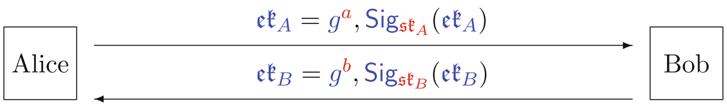
Alice
$\mathfrak{ek}_A = g^a, \mathsf{Sig}_{\mathfrak{sk}_A}(\mathfrak{ek}_A)$
$\mathfrak{ek}_B = g^b, \mathsf{Sig}_{\mathfrak{sk}_B}(\mathfrak{ek}_B)$
Bob

FIGURE 18.13. Signed Diffie–Hellman key exchange

$$\begin{aligned} A \longrightarrow B &: \mathfrak{ek}_A = g^a, \mathsf{Sig}_{\mathfrak{sk}_A}(\mathfrak{ek}_A), \\ B \longrightarrow A &: \mathfrak{ek}_B = g^b, \mathsf{Sig}_{\mathfrak{sk}_B}(\mathfrak{ek}_B). \end{aligned}$$

The problem is that we again have the attack of stripping the signature from Alice's message, replacing it with Eve's; then Bob will think he shares a key with Eve, whereas actually he shares a key with Alice. In Figure 18.11 this was solved by encrypting the identity of the sender, so that it could not be tampered with. However, in Diffie–Hellman key exchange there is no encryption used into which we can embed the identity.

18.4.4. Station-to-Station Protocol: To get around this latter problem the following protocol was invented, called the *station-to-station* (STS) protocol. The original presentation dates back to 1987. The basic idea is that the two parties encrypt their signatures using some symmetric key encryption algorithm (e_k, d_k), and the key, $k \leftarrow H(g^{a \cdot b})$, derived from the key exchange protocol. In particular this means that the initiator's, in our case Alice's, signature needs to be sent in a third message flow. Thus we have the flows:

$$\begin{aligned} A \longrightarrow B &: \mathfrak{ek}_A = g^a, && \\ B \longrightarrow A &: \mathfrak{ek}_B = g^b, e_k\left(\mathsf{Sig}_{\mathfrak{sk}_B}(\mathfrak{ek}_B, \mathfrak{ek}_A)\right) && \text{where } k \leftarrow H({\mathfrak{ek}_A}^b), \\ A \longrightarrow B &: e_k\left(\mathsf{Sig}_{\mathfrak{sk}_A}(\mathfrak{ek}_A, \mathfrak{ek}_B)\right) && \text{where } k \leftarrow H({\mathfrak{ek}_B}^a). \end{aligned}$$

Notice that the messages signed by each party have the group elements in different orders. Another variant on the STS protocol is for the parties to authenticate their signatures using a MAC function $(\mathsf{Mac}_{k'}, \mathsf{Verify}_{k'})$. In this variant the key derivation function is used to derive a key k to use in the following protocol (for which we require key agreement scheme), and a separate key k' to perform the authentication of the signature values. The reason for this is to provide a clear separation between k and the protocol used to derive it. So in this variant of the STS protocol the message flows become:

$$\begin{aligned} A \longrightarrow B &: \mathfrak{ek}_A = g^a, && \\ B \longrightarrow A &: \mathfrak{ek}_B = g^b, S_B := \mathsf{Sig}_{\mathfrak{sk}_B}(\mathfrak{ek}_B, \mathfrak{ek}_A), \mathsf{Mac}_{k'}\left(S_B\right) && \text{where } k \| k' \leftarrow H({\mathfrak{ek}_A}^b) \\ A \longrightarrow B &: S_A := \mathsf{Sig}_{\mathfrak{sk}_A}(\mathfrak{ek}_A, \mathfrak{ek}_B), \mathsf{Mac}_{k'}\left(S_A\right) && \text{where } k \| k' \leftarrow H({\mathfrak{ek}_B}^a) \end{aligned}$$

18.4.5. Blake-Wilson–Menezes Protocol: One can ask whether one can obtain authentication without the need for signatures and/or MACs as in the station-to-station protocol, and without the need for additional data to be sent, or the additional third message flow. The answer to all of these questions is *yes*, as the following protocol, due to Blake-Wilson and Menezes, and the MQV protocol of the next section will show.

Originally the Diffie–Hellman protocol was presented as a way to provide authentic keys given shared static public keys $\mathfrak{pk}_A = g^a$ and $\mathfrak{pk}_B = g^b$. In other words these values were not exchanged, but used to produce a static authenticated shared symmetric key. Then people realized by exchanging ephemeral versions of such keys one could obtain new symmetric keys for each iteration. It turns out that we can combine both the static and the public variants of Diffie–Hellman key exchange to obtain an authenticated key agreement protocol, without the need for digital signatures.

To do this we assume that Alice has a long-term *static* public/private key pair given by $(\mathfrak{pk}_A = g^a, a)$, and Bob has a similar long-term public/private key pair given by $(\mathfrak{pk}_B = g^b, b)$. We first assume that Alice has obtained an authentic version of Bob's public key $\mathfrak{pk}_B$, say via a digital certificate, and vice versa. We can then obtain an authenticated key agreement protocol using only two message flows, as follows:

$$\begin{aligned} A &\longrightarrow B : \mathfrak{ek}_A = g^x, \\ B &\longrightarrow A : \mathfrak{ek}_B = g^y. \end{aligned}$$

Notice that the message flows are *identical* to the original Diffie–Hellman protocol. The key difference is in how the shared secret key k is derived. Alice derives it via the equation

$$k \leftarrow H({\mathfrak{pk}_B}^x, {\mathfrak{ek}_B}^a) = H(g^{b\cdot x}, g^{y\cdot a}).$$

The same key is derived by Bob using the equation

$$k \leftarrow H({\mathfrak{ek}_A}^b, {\mathfrak{pk}_A}^y) = H(g^{x\cdot b}, g^{a\cdot y}).$$

18.4.6. MQV Protocol: The only problem with the previous protocol is that one must perform three exponentiations per key agreement per party. Alice needs to compute $\mathfrak{ek}_A = g^x$, ${\mathfrak{ek}_B}^x$ and ${\mathfrak{pk}_B}^a$. This led Menezes, Qu and Vanstone to invent the following protocol, called the MQV protocol. Once again, being based on the Diffie–Hellman protocol, security is based on the discrete logarithm problem in a group G generated by g. Like the Blake-Wilson–Menezes protocol, MQV works by assuming that both parties, Alice and Bob, first generate long-term public/private key pairs which we shall denote by $(\mathfrak{pk}_A = g^a, a)$ and $(\mathfrak{pk}_B = g^b, b)$. Again, we shall assume that Bob knows that $\mathfrak{pk}_A$ is the authentic public key belonging to Alice and that Alice knows that $\mathfrak{pk}_B$ is the authentic public key belonging to Bob.

Assume Alice and Bob now want to agree on a secret session key; they execute the same message flows as the Blake-Wilson–Menezes protocol, namely:

$$\begin{aligned} A &\longrightarrow B : \mathfrak{ek}_A = g^x, \\ B &\longrightarrow A : \mathfrak{ek}_B = g^y. \end{aligned}$$

So this does not look much different to the standard un-authenticated Diffie–Hellman protocol or the Blake-Wilson–Menezes protocol. However, the key difference is in how the final session key is created from the relevant values.

Assume you are Alice, so you know

$$\mathfrak{pk}_A, \mathfrak{pk}_B, \mathfrak{ek}_A, \mathfrak{ek}_B,\ a \text{ and } x.$$

Let l denote half the bit size of the order of the group G, for example if we are using a group with order $q \approx 2^{256}$ then we set $l = 256/2 = 128$. To determine the session key, Alice now computes

(1) Convert $\mathfrak{ek}_A$ to an integer i.
(2) $s_A \leftarrow (i \pmod{2^l}) + 2^l$.
(3) Convert $\mathfrak{ek}_B$ to an integer j.
(4) $t_A \leftarrow (j \pmod{2^l}) + 2^l$.
(5) $h_A \leftarrow x + s_A \cdot a \pmod{q}$.
(6) $K_A \leftarrow (\mathfrak{ek}_B \cdot {\mathfrak{pk}_B}^{t_A})^{h_A}$.

Notice how s_A and t_A are exactly l bits in length and, assuming the conversion of a group element to an integer results in a "random-looking" integer, these values will also behave like random l-bit integers. Thus Alice needs to compute one full exponentiation, to produce $\mathfrak{ek}_A$, and one multi-exponentiation to produce K_A.

Bob runs exactly the same calculation but with the roles of the public and private keys swapped around in the obvious manner, namely:

(1) Convert $\mathfrak{ek}_B$ to an integer i.
(2) $s_B \leftarrow (i \pmod{2^l}) + 2^l$.
(3) Convert $\mathfrak{ek}_A$ to an integer j.
(4) $t_B \leftarrow (j \pmod{2^l}) + 2^l$.
(5) $h_B \leftarrow y + s_B \cdot b \pmod{q}$.
(6) $K_B \leftarrow (\mathfrak{ek}_A \cdot \mathfrak{pk}_A{}^{t_B})^{h_B}$.

Then $K_A = K_B$ is the shared secret. To see why the K_A computed by Alice and the K_B computed by Bob are the same we notice that the s_A and t_A seen by Alice, are swapped when seen by Bob, i.e. $s_A = t_B$ and $s_B = t_A$. We see that

$$\begin{aligned}
\mathrm{dlog}_g(K_A) &= \mathrm{dlog}_g\left((\mathfrak{ek}_B \cdot \mathfrak{pk}_B{}^{t_A})^{h_A}\right) = (y + b \cdot t_A) \cdot h_A \\
&= y \cdot (x + s_A \cdot a) + b \cdot t_A \cdot (x + s_A \cdot a) = y \cdot (x + t_B \cdot a) + b \cdot s_B \cdot (c + t_B \cdot a) \\
&= x \cdot (y + s_B \cdot b) + a \cdot t_B \cdot (d + s_B \cdot b) = (x + a \cdot t_B) \cdot h_B \\
&= \mathrm{dlog}_g\left((\mathfrak{ek}_A \cdot \mathfrak{pk}_A^{t_B})^{h_B}\right) = \mathrm{dlog}_g(K_B).
\end{aligned}$$

18.5. The Symbolic Method of Protocol Analysis

One can see that the above protocols are very intricate; spotting flaws in them can be a very subtle business. A number of different approaches have been proposed to try and make the design of these protocols more scientific. The first school is based on so-called formal methods, and treats protocols via means of a symbolic algebra. The second school is closer to our earlier modelling of encryption and signature schemes, in that it is based on a cryptographic game between a challenger and an adversary.

The most influential of the methods in the first school is the BAN logic invented by Burrows, Abadi and Needham. The BAN logic has a large number of drawbacks compared to more modern logical analysis tools, but was very influential in the design and analysis of symmetric-key-based key agreement protocols such as Kerberos and the Needham–Schroeder protocol. It has now been supplanted by more complicated logics and formal methods, but it is of historical importance and the study of the BAN logic can still be very instructive for protocol designers. The main benefit in using symbolic methods and logics is that the analysis can usually be semi-automated via theorem provers and the like; this should be compared to the cryptographic analysis method which is still done mainly by hand.

First we really need to pause and decide what are the goals of key agreement and key transport, and what position the parties start from. In the symmetric key setting we assume all parties, A and B say, only share secret keys K_{as} and K_{bs} with the trusted third party S. In the public key setting we assume that all parties have a public/private key pair $(\mathfrak{pk}_A, \mathfrak{sk}_A)$, and that the public key $\mathfrak{pk}_A$ is bound to an entity's identity A, via some form of certificate.

In both cases parties A and B want to agree and/or transport a symmetric session key K_{ab} for use in some further protocol. This new session key should be fresh, i.e. it has not been used by any other party before and has been recently created. The freshness property will stop attacks whereby the adversary replays messages so as to use an old key again. Freshness can also be useful in deducing that the party with which you are communicating is still alive.

We also need to decide what capabilities an attacker has. As always we assume the worst possible situation in which an attacker can intercept any message flow over the network. She can then stop a message, alter it or change its destination. An attacker is also able to distribute her own messages over the network. With such a high-powered attacker it is often assumed that the attacker *is* the network.

The main idea of BAN logic is that one should concentrate on what the parties believe is happening. It does not matter what is actually happening; we need to understand exactly what each party can logically deduce, from its own view of the protocol, about what is actually happening.

In the BAN logic, complex statements are made up of some basic atomic statements which are either true or false. These atomic statements can be combined into more complex ones using conjunction, which is denoted by a comma. The basic atomic statements are given by:

- $P| \equiv X$ means P *believes* (or is entitled to believe) X.
  The principal P may act as though X is true.
- $P \lhd X$ means P *sees* X.
  Someone has sent a message to P containing X, so P can now read and repeat X.
- $P| \sim X$ means P *once said* X and P believed X when it was said.
  Note this tells us nothing about whether X was said recently or in the distant past.
- $P| \Rightarrow X$ means P has *jurisdiction* over X.
  This means P is an authority on X and should be trusted on this matter.
- $\# X$ means the formula X is *fresh.*
  This is usually used for nonces.
- $P \overset{K}{\leftrightarrow} Q$ means P and Q may use the *shared key* K to communicate.
  The key is assumed good and it will never be discovered by anyone other than P and Q, unless the protocol itself makes this happen.
- $\{X\}_K$, means as usual that X is *encrypted* under the key K.
  The encryption is assumed to be perfect in that X will remain secret unless deliberately disclosed by a party at some other point in the protocol.

We start with a set of statements which are assumed to be true at the start of the protocol. When executing the protocol we infer the truth of new statements via the BAN logic postulates, or rules of inference. The format we use to specify rules of inference is as follows:

$$\frac{A, B}{C}$$

which means that if A and B are true then we can conclude C is also true. This is a standard notation used in many areas of logic within computer science.

- **Message Meaning Rule:**
  $$\frac{A| \equiv A \overset{K}{\leftrightarrow} B, A \lhd \{X\}_K}{A| \equiv B| \sim X}.$$
  In words, if both
  - A believes she shares the key K with B,
  - A sees X encrypted under the key K,

  we can deduce that A believes that B once said X. Note that this implicitly assumes that A never said X.
- **Nonce Verification Rule:**
  $$\frac{A| \equiv \# X, A| \equiv B| \sim X}{A| \equiv B| \equiv X}.$$
  In words, if both
  - A believes X is fresh (i.e. recent),

– A believes B once said X,

then we can deduce that A believes that B still believes X.

- **Jurisdiction Rule:**
$$\frac{A| \equiv B| \Rightarrow X, A| \equiv B| \equiv X}{A| \equiv X}.$$
In words, if both
  – A believes B has jurisdiction over X, i.e. A trusts B on X,
  – A believes B believes X,

  then we conclude that A also believes X.
- **Other Rules:** The belief operator and conjunction can be manipulated as follows:
$$\frac{P| \equiv X, P| \equiv Y}{P| \equiv (X,Y)}, \quad \frac{P| \equiv (X,Y)}{P| \equiv X}, \quad \frac{P| \equiv Q| \equiv (X,Y)}{P| \equiv Q| \equiv X}.$$
A similar rule also applies to the "once said" operator
$$\frac{P| \equiv Q| \sim (X,Y)}{P| \equiv Q| \sim X}.$$
Note that $P| \equiv Q| \sim X$ and $P| \equiv Q| \sim Y$ does not imply $P| \equiv Q| \sim (X,Y)$, since that would imply X and Y were said at the same time. Finally, if part of a formula is fresh then so is the whole formula
$$\frac{P| \equiv \#X}{P| \equiv \#(X,Y)}.$$

We wish to analyse a key agreement protocol between A and B using the BAN logic. But what is the goal of such a protocol when expressed in this formalism? The minimum we want to achieve is
$$A| \equiv A \overset{K}{\leftrightarrow} B \text{ and } B| \equiv A \overset{K}{\leftrightarrow} B,$$
i.e. both parties believe they share a secret key with each other. However, we could expect to achieve more, for example
$$A| \equiv B| \equiv A \overset{K}{\leftrightarrow} B \text{ and } B| \equiv A| \equiv A \overset{K}{\leftrightarrow} B,$$
which is called key confirmation. In words, we may want to achieve that, after the protocol has run, A is assured that B knows he is sharing a key with A, and it is the same key A believes she is sharing with B.

Before analysing a protocol using the BAN logic we convert the protocol into logical statements. This process is called *idealization*, and is the most error prone part of the procedure since it cannot be automated. We also need to specify the assumptions, or axioms, which hold at the beginning of the protocol. To see this process in "real life" we analyse the Wide-Mouth Frog protocol for key agreement using synchronized clocks.

Example: Wide-Mouth Frog Protocol: Recall the Wide-Mouth Frog protocol
$$A \longrightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}.$$
This becomes the idealized protocol
$$A \longrightarrow S : \{T_a, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A| \equiv A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}.$$
One should read the idealization of the first message as telling S that
- T_a is a timestamp/nonce,
- K_{ab} is a key which is meant as a key to communicate with B.

So what assumptions exist at the start of the protocol? Clearly A, B and S share secret keys, which in BAN logic becomes

$$\begin{array}{cc} A| \equiv A \overset{K_{as}}{\leftrightarrow} S, & S| \equiv A \overset{K_{as}}{\leftrightarrow} S, \\ B| \equiv B \overset{K_{bs}}{\leftrightarrow} S, & S| \equiv B \overset{K_{bs}}{\leftrightarrow} S. \end{array}$$

There are a couple of nonce assumptions,

$$S| \equiv \# T_a \text{ and } B| \equiv \# T_s.$$

Finally, we have the following three assumptions

- B trusts A to invent good keys,

$$B| \equiv (A| \Rightarrow A \overset{K_{ab}}{\leftrightarrow} B),$$

- B trusts S to relay the key from A,

$$B| \equiv (S| \Rightarrow A| \equiv A \overset{K_{ab}}{\leftrightarrow} B),$$

- A knows the session key in advance,

$$A| \equiv A \overset{K_{ab}}{\leftrightarrow} B.$$

Notice how these last three assumptions specify the problems we associated with this protocol in the earlier section. Using these assumptions we can now analyse the protocol.

- Let us see what we can deduce from the first message

$$A \longrightarrow S : \{T_a, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{as}}.$$

  - Since S sees the message encrypted under K_{as} he can deduce that A said the message.
  - Since T_a is believed by S to be fresh he concludes the whole message is fresh.
  - Since the whole message is fresh, S concludes that A currently believes the whole of it.
  - S then concludes

$$S| \equiv A| \equiv A \overset{K_{ab}}{\leftrightarrow} B,$$

    which is what we need to conclude so that S can send the second message of the protocol.

- We now look at what happens when we analyse the second message

$$S \longrightarrow B : \{T_s, A| \equiv A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}.$$

  - Since B sees the message encrypted under K_{bs} he can deduce that S said the message.
  - Since T_s is believed by B to be fresh he concludes the whole message is fresh.
  - Since the whole message is fresh, B concludes that S currently believes the whole of it.
  - So B believes that S believes the second part of the message.
  - But B believes S has authority on whether A knows the key and B believes A has authority to generate the key.

- From the analysis of both messages we can conclude

$$B| \equiv A \overset{K_{ab}}{\leftrightarrow} B$$

  and

$$B| \equiv A| \equiv A \overset{K_{ab}}{\leftrightarrow} B.$$

  Combining this with our axiom, $A| \equiv A \overset{K_{ab}}{\leftrightarrow} B$, we conclude that the key agreement protocol is sound. The only requirement we have not met is that

$$A| \equiv B| \equiv A \overset{K_{ab}}{\leftrightarrow} B,$$

i.e. A does not achieve confirmation that B has received the key.

Notice what the application of the BAN logic has done is to make the axioms clearer, so it is easier to compare which assumptions each protocol needs to make it work. In addition it clarifies what the result of running the protocol is from all parties' points of view. However, it does not *prove* the protocol is secure in a cryptographic sense. To do that we need a more complex method of analysis, which is less amenable to computer application.

18.6. The Game-Based Method of Protocol Analysis

To provide a more rigorous analysis of protocols (which does not assume, for example, that encryption works as a perfect black box), we now present a method of analysis which resembles the games we used to introduce encryption, signatures and MACs. Recall that there we had an adversary A who tried to achieve a certain goal, using a set of powers. The goal was presented as some winning condition, and the powers were presented as giving the adversary access to some oracles. For key agreement protocols we will adopt a winning condition which is akin to the Real-or-Random winning condition for the encryption games, i.e. an adversary should not be able to tell the difference between a real key agreed during a key agreement protocol and a key chosen completely at random.

What is more complicated is the definition of the oracles. In our previous examples the oracles were relatively simple; they either encrypted, decrypted, signed or verified some data given some hidden key. As a key agreement protocol is a *protocol*, data is passed between players; in addition there could be many keys within the protocol and so one piece of data could be sent to be processed by different entities using different keys (recall we assume the adversary has control of the network). In addition we have seen attacks on key agreement protocols which rely on messages being passed to additional entities, and not just two. Thus we need to model security where there are many participants. In addition participants may be interacting with many parties at the same time, and could be interacting with the same party many times (e.g. a client connecting to a web server in multiple sessions).

Modelling the Participants: We first set up some parties. These are going to be users $U \in \mathcal{U}$ who start our game being honest participants in the protocol. For symmetric key based protocols each entity will have a list of secret keys k_{US} of keys shared with a special trusted party S. For public key protocols each party will have a public/private key pair $(\mathfrak{pk}_U, \mathfrak{sk}_U)$, where $\mathfrak{sk}_U$ will be held by the party and the public key $\mathfrak{pk}_U$ will be held (in a certified form) by all other parties. It may be the case that parties have two public/private key pairs, one for public key encryption and one for public key signatures, in which case we will denote them by $(\mathfrak{pk}_U^{(e)}, \mathfrak{sk}_U^{(e)})$ and $(\mathfrak{pk}_U^{(s)}, \mathfrak{sk}_U^{(s)})$. Each party U will have a state $\mathsf{state}_U = \{\mathfrak{k}_U, \kappa_U\}$, which consists of the secret data above $\mathfrak{k}_U$ and a Boolean variable κ_U. The variable κ_U denotes whether the party is corrupted or not. At the start of the game κ_U is set to false.

For public key protocols we are also going to have a set of users $V \in \mathcal{V}$ for whom the adversary can register its own public keys. For the users in $\mathcal{V}$ we do not assume that the adversary knows the underlying private key, only that it registers public keys of its choice.

The adversary can interact with these *parties* in the following manner. It has an oracle $\mathcal{O}_U$, for each user $U \in \mathcal{U}$, to which it can pass a single command $\mathsf{corrupt}$; on receiving this command the game returns the value $\mathfrak{k}_U$ to the adversary and sets $\kappa_U \leftarrow \mathsf{true}$. Thus this oracle allows the adversary to take control of any party she desires. In addition there is an oracle $\mathcal{O}_V$, for each user $V \in \mathcal{V}$, to which it can pass the single command $(\mathsf{register}, \mathfrak{pk}_V)$ which registers $\mathfrak{pk}_V$ as the public key of the user V.

Modelling the Sessions: As well as the data associated with each party we have data associated with the *view* of a party of a session in which it is engaging. Note that this is about the view of the party and may not correspond to the truth. Each party $U \in \mathcal{U}$ may have multiple sessions which it thinks it is having with party $U' \in \mathcal{U} \cup \mathcal{V}$; party U will index these sessions by an integer variable i. The goal of the session is to agree an ephemeral secret key $\mathfrak{ek}_{U,U',i}$, and as the key agreement protocol proceeds the session will have different states, which we shall denote by $\Sigma_{U,U',i}$. The possible states are:

- $\perp$: This is the initial value of $\Sigma_{U,U',i}$ and indicates that nothing has happened.
- accept: This indicates that party U thinks that a key has been agreed, and party U thinks it is equal to $\mathfrak{ek}_{U,U',i}$.
- reject: This indicates that party U has rejected this session of the protocol.
- $\mathsf{revealed}$: This indicates that the adversary has obtained the key $\mathfrak{ek}_{U,U',i}$; see below for how she did this.

Within a session we also maintain a list of messages received and sent, a so-called transcript $\mathcal{T}_{U,U',i} = \{m_1, r_1, m_2, r_2, \ldots, m_n, r_n\}$, where m_j is the jth message received by party U in its ith session with U', and r_n is the associated response which it made. The transcript is initially set to be the empty set. Thus the session state is given by $\mathsf{session}_{U,U',i} = \{\mathfrak{ek}_{U,U',i}, \Sigma_{U,U',i}, \mathcal{T}_{U,U',i}, s_{U,U',i}\}$, where $s_{U,U',i}$ is some protocol specific state information.

To obtain information about the sessions, and to drive the protocol, the adversary has a message oracle $\mathcal{O}$ which takes four arguments, $\mathcal{O}(U, U', i, m)$, which is processed as follows:

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then start a new protocol session with U' and let r be the initial message, set $\mathcal{T}_{U,U',i} \leftarrow \{m, r\}$ and return r to the adversary.
- Otherwise if $\mathcal{T}_{U,U',i} = \{m_1, r_1, m_2, r_2, \ldots, m_n, r_n\}$ and the message m is the $(n+1)$st input message in the protocol. The oracle computes the response r_{n+1} and adds the message and response to $\mathcal{T}_{U,U',i}$ and returns r_{n+1} to the adversary. The final response in a protocol, to signal the end, is to set $r_{n+1} \leftarrow \perp$.

Notice that messages are not sent between participants; we allow the adversary to do that. She can decide which message gets sent where, or if it even gets sent at all.

Matching Conversations: We now need to define when two such sessions, run by different participants, correspond to the same protocol execution. This is done using the concept of a matching conversation. Suppose we have two transcripts

$$\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1, m_2, r_2, \ldots, m_n, r_n\},$$
$$\mathcal{T}_{W,W',i'} = \{m'_1, r'_1, m'_2, r'_2, \ldots, m_k, r_k\},$$

such that

- $m'_i = r_{i-1}$ for $i \geq 1$,
- $m_i = r'_{i-1}$ for $i > 1$,
- n even: $r_n = \perp$ and $k = n - 1$,
- n odd: $r_k = \perp$ and $k = n$,
- $U = W'$ and $U' = W$.

In such a situation we say the two transcripts are matching conversations.

Winning the Game: Recall that our winning condition is going to be akin to the Real-or-Random definition for encryption security. So the game selects a hidden bit b; if $b = 0$ then the adversary will be given a random key for its challenge, otherwise it will be given a real key. So we need to decide on which key agreement protocol execution the adversary will be challenged. Instead of defining one for the adversary we let the adversary choose its own session which which it wants to be queried, subject to some constraints which we shall discuss below. This is done by means of a so-called test oracle $\mathcal{O}_{\mathsf{Test}}$.

The test oracle takes as input a tuple (U, U', i), with $U, U' \in \mathcal{U}$ and if $b = 1$ it returns $\mathfrak{ek}_{U,U',i}$, and if $b = 0$ it returns a random key of the same size as $\mathfrak{ek}_{U,U',i}$. The test oracle may only be called once by the adversary, and when making the call the adversary has to obey certain rules of the game. The restrictions on the test oracle query are for two reasons; firstly to ensure that the query makes sense, and secondly to ensure that the game is not too easy to win. For our purposes the restrictions will be as follows:

- $\Sigma_{U,U',i} = \mathsf{accept}$. Otherwise the key is either not defined, or it has been revealed and the adversary already knows it.
- There should be no revealed session (W, W', i') which has a matching conversation with (U, U', i). Since a matching conversation should correspond to the same session key, this protects against a trivial break in these circumstances.
- $\kappa_{U'} = \kappa_U = \mathsf{false}$. In other words both U and its partner U' are not corrupted.

Note that we do not assume that the test session has any matching conversations at all. It could be a conversation with the adversary.

At the end of the game (which we call the Authenticated Key Agreement game) the adversary A outputs its guess b' as to the hidden bit b, and we define the advantage of the adversary A against protocol Π in the usual way as

$$\mathrm{Adv}_{\Pi}^{\mathsf{AKA}}(A; n_P, n_S) = 2 \cdot \left| \Pr[A \text{ wins}] - \frac{1}{2} \right|,$$

where n_P upper bounds the number of participants with which A interacts and n_S upper bounds the number of sessions executable between two participants. To avoid trivial breaks of some simple protocols we also impose the restriction that after the test query on (U, U', i) is made we do not allow calls to $\mathcal{O}_U$ and $\mathcal{O}_{U'}$, i.e. at the end of the game U and U' must still be uncorrupted. If we want to model protocols with forward secrecy then we remove this restriction; thus the test session will consist of messages which were sent and received by U before the corruption of either party occured. We shall not discuss forward secrecy anymore, except to point out when protocols are not forward secure.

For a protocol to be deemed secure we need more than just that the advantage is small.

Definition 18.1 (AKA Security)**.** *An authenticated key agreement protocol* Π *is said to be secure if*

1. *There is a matching conversation between sessions* (U, U', i) *and* (U', U, j) *and* U *and* U' *are uncorrupted, then we have* $\Sigma_{U,U',i} = \Sigma_{U',U,j} = \mathsf{accept}$ *and* $\mathfrak{ek} = \mathfrak{ek}_{U,U',i} = \mathfrak{ek}_{U',U,j}$*, and* $\mathfrak{ek}$ *is distributed uniformly at random over the desired key space.*
2. $\mathrm{Adv}_{\Pi}^{\mathsf{AKA}}(A)$ *is "small".*

Public-Key-Encryption-Based Key Transport Example: To define a protocol within the above game framework we simply need to provide the information as to how the oracle $\mathcal{O}(U, U', i, m)$ should behave. We present a number of examples, all based on public key techniques; we leave symmetric-key-based protocols to the reader. We start with the basic key transport based on public key encryption, in particular the third version of the protocol given in Figure 18.11. The "code" for the $\mathcal{O}(U, U', i, m)$ oracle in this case is given by:

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then set $\mathfrak{ek}_{U,U',i} \leftarrow \mathbb{K}$, $c_U \leftarrow e_{\mathfrak{pk}_{U'}}^{(e)}(U \| \mathfrak{ek}_{U,U',i})$, $s_U \leftarrow \mathsf{Sig}_{\mathfrak{sk}_U^{(s)}}(c)$, $r \leftarrow (c_U, s_U)$, $\sigma_{U,U',i} \leftarrow \mathsf{accept}$. $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r\}$, and return r to the adversary.
- If $m = (c_{U'}, s_{U'})$, $\mathcal{T}_{U,U',i} = \emptyset$ and $\mathsf{Verify}_{\mathfrak{pk}_{U'}^{(s)}}(s_{U'}, c_{U'}) = \mathsf{true}$ then set $m^* \leftarrow e_{\mathfrak{sk}_{U'}^{(e)}}(c)$. If $m^* = \perp$ then abort. Parse m^* as $(A \| \mathfrak{ek}_{U,U',i})$ and abort if $A \neq U'$. Finally set $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{m, \perp\}$, and return $\perp$ to the adversary.

- Otherwise abort.

However, earlier we said that this had a replay attack. This replay attack can now be described in our AKA model. We take the two-user case, and pass the output from the initiator to the responder twice. Thus the responder thinks they are engaging with two sessions with the initiator. We then reveal on one of these sessions and by testing the other and so we can decide is the test oracle returns a random value or not.

- Assume two parties U and U'.
- $(c_U, s_U) \leftarrow \mathcal{O}(U, U', 1, \mathsf{init})$.
- $\mathcal{O}(U', U, 1, (c_U, s_U))$.
- $\mathcal{O}(U', U, 2, (c_U, s_U))$.
- $\mathfrak{k}' \leftarrow \mathcal{O}(U', U, 2, \mathsf{reveal})$.
- $\mathfrak{k} \leftarrow \mathcal{O}_{\mathsf{Test}}(U, U', 1)$.
- If $\mathfrak{k} = \mathfrak{k}'$ then return $b' = 1$.
- Return $b' = 0$.

An obvious fix is to remove the ability for replays, but without each party maintaining a large state this is relatively complex. As remarked above, the preferred fix would be to have each party contribute entropy to each protocol run.

Diffie–Hellman Example: We now turn to the basic Diffie–Hellman protocol from Figure 18.12. We assume we are working in a finite abelian group G of order q generated by g. The "code" for the $\mathcal{O}(U, U', i, m)$ oracle in this case is given by:

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $r \leftarrow g^{s_{U,U',i}}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r\}$, and return r to the adversary.
- If $m \in G$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $r \leftarrow g^{s_{U,U',i}}$, $\mathfrak{ek}_{U,U',i} \leftarrow m^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{m, r\}$, and return r to the adversary.
- If $m \in G$ and $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1\}$ then $\mathfrak{ek}_{U,U',i} \leftarrow m^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1, m, \perp\}$, and return $\perp$ to the adversary.
- Otherwise abort.

We already know this is not a secure authenticated key agreement protocol due to the man-in-the-middle attack. Indeed no public keys are even used within the protocol, so it does not have any authentication at all within it. However, to illustrate how this is captured in the AKA security model we present an attack, *within the model*. Our adversary performs the following steps:

- Assume four distinct parties U, U', K and L.
- $e_U \leftarrow \mathcal{O}(U, U', 1, \mathsf{init})$.
- $e_K \leftarrow \mathcal{O}(K, L, 1, e_U)$.
- $\mathcal{O}(U, U', 1, e_K)$.
- $\mathfrak{k}' \leftarrow \mathcal{O}(K, L, 1, \mathsf{reveal})$.
- $\mathfrak{k} \leftarrow \mathcal{O}_{\mathsf{Test}}(U, U', 1)$.
- If $\mathfrak{k} = \mathfrak{k}'$ then return $b' = 1$.
- Return $b' = 0$.

Note that this attack works since the test session $(U, U', 1)$ does not have a matching conversation with *any* other session. It does "match" with the session $(K, L, 1)$ in terms of message flows, but we do not have $U = L$ and $U' = K$. This allows the adversary to reveal the key for session $(K, L, 1)$, whilst still allowing session $(U, U', 1)$ to be passed to the test oracle. Also note that the sessions $(U, U', 1)$ and $(K, L, 1)$ agree on the same key, and that neither U nor U' have been corrupted. Thus the above method means the adversary can win the AKA security game with probability one for the Diffie–Hellman protocol.

Signed Diffie–Hellman Example: We now turn to the Signed Diffie–Hellman protocol from Figure 18.13. The "code" for the $\mathcal{O}(U, U', i, m)$ oracle is now given by:

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $e_U \leftarrow g^{s_{U,U',i}}$, $s_U \leftarrow \mathsf{Sig}_{\mathfrak{sk}_U}(e_U)$, $r \leftarrow (e_U, s_U)$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r\}$, and return r to the adversary.
- If $m = (e_{U'}, s_{U'})$ with $e_{U'} \in G$, $\mathcal{T}_{U,U',i} = \emptyset$ and $\mathsf{Verify}_{\mathfrak{pk}_{U'}}(s_{U'}, e_{U'}) = \mathsf{true}$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $e_U \leftarrow g^{s_{U,U',i}}$, $s_U \leftarrow \mathsf{Sig}_{\mathfrak{sk}_U}(e_U)$, $r \leftarrow (e_U, s_U)$, $\mathfrak{ek}_{U,U',i} \leftarrow {e_{U'}}^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{m, r\}$, and return r to the adversary.
- If $m = (e_{U'}, s_{U'})$ with $e_{U'} \in G$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, (e_U, s_U)\}$ and $\mathsf{Verify}_{\mathfrak{pk}_{U'}}(s_{U'}, e_{U'}) = \mathsf{true}$ then $\mathfrak{ek}_{U,U',i} \leftarrow {e_{U'}}^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, (e_U, s_U), m, \perp\}$, and return $\perp$ to the adversary.
- Otherwise abort.

Within the AKA security model we can present the following attack, which formalizes the replacement of signature attack discussed above. Our adversary performs the following steps:

- Assume three distinct parties U, U' and K.
- $\mathfrak{sk}_K \leftarrow \mathcal{O}_K(\mathsf{corrupt})$.
- $(e_U, s_U) \leftarrow \mathcal{O}(U, U', 1, \mathsf{init})$.
- $s_K \leftarrow \mathsf{Sig}_{\mathfrak{sk}_K}(e_U)$.
- $(e_{U'}, s_{U'}) \leftarrow \mathcal{O}(U', K, 1, (e_U, s_K))$.
- $\mathcal{O}(U, U', 1, (e_{U'}, s_{U'}))$.
- $\mathfrak{k}' \leftarrow \mathcal{O}(U', K, 1, \mathsf{reveal})$.
- $\mathfrak{k} \leftarrow \mathcal{O}_{\mathsf{Test}}(U, U', 1)$.
- If $\mathfrak{k} = \mathfrak{k}'$ then return $b' = 1$.
- Return $b' = 0$.

In this execution only party K is corrupted, and in particular neither U nor U' have been corrupted. However, party U thinks it is talking to party U', whereas party U' thinks it is talking to party K; yet in the sessions executed by U and U' they agree on the same ephemeral key. Hence, the above attack is valid within the AKA model and succeeds with probability one.

Blake-Wilson–Menezes Example: It turns out that the Blake-Wilson–Menezes protocol also has an attack on it, which makes use of the fact that we allow the adversary to register public keys of its own choosing. We describe the attack via the following pseudo-code, assuming the underlying group is of prime order q.

- Assume two legitimate parties U and U'.
- $t \leftarrow (\mathbb{Z}/q\mathbb{Z})^*$.
- $\mathfrak{pk}_V \leftarrow {\mathfrak{pk}_U}^t$.
- $\mathcal{O}_V(\mathsf{register}, \mathfrak{pk}_V)$.
- $(e_U) \leftarrow \mathcal{O}(U, U', 1, \mathsf{init})$.
- $(e_{U'}) \leftarrow \mathcal{O}(U', V, 1, e_U)$.
- $\mathcal{O}(U, U', 1, e_{U'}^t)$.
- $\mathfrak{k}' \leftarrow \mathcal{O}(U', V, 1, \mathsf{reveal})$.
- $\mathfrak{k} \leftarrow \mathcal{O}_{\mathsf{Test}}(U, U', 1)$.
- If $\mathfrak{k} = \mathfrak{k}'$ then return $b' = 1$.
- Return $b' = 0$.

Notice that U thinks it is talking to U', whereas U' thinks it is talking to V. Let x be the secret ephemeral key chosen by U in the above execution, and y is the secret ephemeral key chosen by U'. Then the secret key for the (test) session U is engaged in is equal to

$$\mathfrak{k} \leftarrow H({\mathfrak{pk}_{U'}}^x, {\mathfrak{ek}_{U'}}^{t \cdot s_U}) = H(g^{s_{U'} \cdot x}, g^{t \cdot y \cdot s_U}),$$

whilst the secret key for the (revealed) session U' is engaged in is equal to

$$\mathfrak{k}' \leftarrow H({\mathfrak{ek}_V}^{s_{U'}}, {\mathfrak{pk}_V}^{y}) = H(g^{x \cdot s_{U'}}, g^{t \cdot s_U \cdot y}).$$

So the two keys are identical, and yet the session executed by U' is not a matching session with that executed by U.

Modified Blake-Wilson–Menezes Example: The problem with the previous protocol is that the key is derived does not depend on whether the sessions involved have matching conversations. We therefore modify the Blake-Wilson–Menezes protocol in the following way, and we are then able to prove the protocol is secure. Again we assume public/private key pairs are given by $\mathfrak{pk}_A = g^a$ for Alice and $\mathfrak{pk}_B = g^b$ for Bob, and that the message flows are still defined by

$$A \longrightarrow B : \mathfrak{ek}_A = g^x,$$
$$B \longrightarrow A : \mathfrak{ek}_B = g^y.$$

The key difference is in how the shared secret key k is derived; we also include the transcript of the protocol within the key derivation. Alice derives it via the equation

$$k \leftarrow H({\mathfrak{pk}_B}^x, {\mathfrak{ek}_B}^a, \mathfrak{pk}_A, \mathfrak{pk}_B, \mathfrak{ek}_A, \mathfrak{ek}_B) = H(g^{b \cdot x}, g^{y \cdot a}, g^a, g^b, g^x, g^y).$$

While Bob derives the same key using the equation

$$k \leftarrow H({\mathfrak{ek}_A}^b, {\mathfrak{pk}_A}^y, \mathfrak{pk}_A, \mathfrak{pk}_B, \mathfrak{ek}_A, \mathfrak{ek}_B) = H(g^{x \cdot b}, g^{a \cdot y}, g^a, g^b, g^x, g^y).$$

In terms of "code" for our $\mathcal{O}(U, U', i, m)$ oracle we have that the protocol is defined by

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $r \leftarrow g^{s_{U,U',i}}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r\}$, and return r to the adversary.
- If $m \in G$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $r \leftarrow g^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{m, r\}$,
$$\mathfrak{ek}_{U,U',i} \leftarrow H(m^{\mathfrak{sk}_U}, {\mathfrak{pk}_{U'}}^{s_{U,U',i}}, \mathfrak{pk}_{U'}, \mathfrak{pk}_U, m, r),$$
and return r to the adversary.
- If $m \in G$ and $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1\}$ then $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1, m, \perp\}$,
$$\mathfrak{ek}_{U,U',i} \leftarrow H({\mathfrak{pk}_{U'}}^{s_{U,U',i}}, m^{s_U}, \mathfrak{pk}_U, \mathfrak{pk}_{U'}, r_1, m),$$
and return $\perp$ to the adversary.
- Otherwise abort.

We can now prove that this protocol is secure assuming the Gap Diffie–Hellman problem is hard.

Theorem 18.2. *Let A denote an adversary against the* AKA *security of the modified Blake-Wilson–Menezes protocol, operating with n_P legitimate users each of whom may have up to n_S sessions, where H is modelled as a random oracle. Then there is an adversary B against the Gap Diffie–Hellman problem in the group G such that*

$$\mathrm{Adv}_{\Pi}^{\mathsf{AKA}}(A; n_P, n_S) \le n_P^2 \cdot n_S \cdot \mathrm{Adv}_G^{\mathsf{Gap\text{-}DHP}}(B).$$

From the theorem it is easy to deduce that the modifed Blake-Wilson–Menezes protocol is secure, assuming the Gap Diffie–Hellman problem is hard.

Proof. We assume we are given an algorithm A against the AKA security of the protocol and we want to create the algorithm B against the Gap Diffie–Hellman problem. Algorithm B will maintain a hash list H-List consisting of elements in $G^6 \times \{0,1\}^k$, representing the calls to H on sextuples of elements in G and the corresponding outputs.

Algorithm B has as input g^a and g^y and is asked to compute $g^{a \cdot y}$, given access to an oracle $\mathcal{O}_{\mathsf{DDH}}$ which checks tuples for being Diffie–Hellman tuples. Algorithm B sets up n_P public/private keys for algorithm A as in the legitimate game, but picks one user $U^* \in \mathcal{U}$ and sets its public key to be g^a. Algorithm B also picks a session identifier $i^* \in \{1, \ldots, n_S\}$, and another identity $W^* \in \mathcal{U}$. We let b denote the private key of entity W^*; notice that this is known to algorithm B, and hence is marked in blue.

Algorithm B then calls algorithm A and responds to its oracle queries as in the real game except for the following cases:

- If $\mathcal{O}_{U^*}(\mathsf{corrupt})$ is called then abort.
- If $\mathcal{O}(U^*, W^*, i^*, \mathsf{reveal})$ is called then abort.
- If $\mathcal{O}(W^*, U^*, i^*, m)$ is called then respond with g^x.
- If $\mathcal{O}_{\mathsf{Test}}(U, W, i)$ is called with $U \neq U^*$, $W \neq W^*$ or $i \neq i^*$ then abort. Otherwise respond with a random value in $\{0, 1\}^k$.

The only problem is in maintaining consistency between any reveal queries made by A and any calls made by A to the hash function H. In other words we need to maintain consistency of the H-List in both the reveal and hash function queries made by A. However, such consistency can be maintained in the same manner as in Theorem 16.9 using the $\mathcal{O}_{\mathsf{DDH}}$ oracle to which algorithm B has access[2].

Note that the probability that B aborts is $1/(n_P^2 \cdot n_S)$. If the algorithm A is able to win the AKA game with non-negligible probability then A must make a *hash* query on the critical sextuple. As we are using the modified Blake-Wilson–Menezes protocol, no reveal query made by A will result in the same input to the hash function as the critical query. Since no other reveal query will have the same transcript etc. Thus to have any advantage A must make the critical call to the hash function.

We now examine what this sextuple will be; we let m denote the ephemeral public key passed to W^* in the test session and let $x = \mathrm{dlog}_g(m)$; note that B does not know x so we mark this value in red. If U^* was an initiator oracle then the sextuple is

$$(\mathfrak{pk}_{W^*}{}^x, \mathfrak{ek}_{W^*}{}^a, \mathfrak{pk}_{U^*}, \mathfrak{pk}_{W^*}, \mathfrak{ek}_{U^*}, \mathfrak{ek}_{W^*}) = (\mathfrak{pk}_{W^*}{}^x, g^{y \cdot a}, g^a, \mathfrak{pk}_{W^*}, m, g^y).$$

Thus in this case the algorithm B simply looks for the critical query on the H-List using its $\mathcal{O}_{\mathsf{DDH}}$ oracle and outputs the second value of the tuple as the answer to the Gap Diffie–Hellman problem. If U^* was the responder oracle, then a similar method can be applied using the first value of the tuple, as the critical sextuple is given by

$$(\mathfrak{ek}_{W^*}{}^a, \mathfrak{pk}_{W^*}{}^x, \mathfrak{pk}_{W^*}, \mathfrak{pk}_{U^*}, \mathfrak{ek}_{W^*}, \mathfrak{ek}_{U^*}) = (g^{y \cdot a}, \mathfrak{pk}_{W^*}{}^x, \mathfrak{pk}_{W^*}, g^a, g^y, m).$$

$\square$

A similar proof can be made for the MQV protocol, which due to space we do not cover here.

Chapter Summary

- Digital certificates allow us to bind a public key to some other information, such as an identity. This binding of key with identity allows us to solve the problem of how to distribute authentic public keys.
- Various PKI systems have been proposed, all of which have problems and benefits associated with them.
- Implicit certificates aim to reduce the bandwidth requirements of standard certificates, however they come with a number of drawbacks.

[2]The details are tedious and are left to the reader.

- A number of key agreement protocols exist based on a trusted third party and symmetric encryption algorithms. These protocols require long-term keys to have been already established with the TTP; they may also require some form of clock synchronization.
- Diffie–Hellman key exchange can be used by two parties to agree on a secret key over an insecure channel. However, Diffie–Hellman is susceptible to a man-in-the-middle attack and so requires some form of authentication of the communicating parties.
- To obtain authentication in the Diffie–Hellman protocol, various different options exist, of which we discussed the STS protocol, the Blake-Wilson–Menezes protocol and the MQV protocol.
- Various formal logics exist to analyse such protocols. The most influential of these has been the BAN logic. These logics help to identify explicit assumptions and problems associated with each protocol; they can identify attacks but usually do not provide security proofs.
- For a computational proof of security one needs to define an elaborate security model. Such models capture a multitude of attacks; resulting in the most simple protocols being deemed insecure.

Further Reading

The paper by Burrows, Abadi and Needham is a very readable introduction to the BAN logic and a number of key agreement protocols based on static symmetric keys; much of our treatment of this subject is based on this paper. Our treatment of security models for key agreement is based on work started in the paper by Bellare and Rogaway. Our proof of security of the modified Blake-Wilson–Menezes protocol is based on the paper by Kudla and Paterson.

M. Bellare and P. Rogaway. Entity authentication and key distribution. Advances in Cryptology – Crypto 1993, LNCS 773, 232–249, Springer, 1994.

M. Burrows, M. Abadi and R. Needham. *A Logic of Authentication.* Digital Equipment Corporation, SRC Research Report 39, 1990.

C. Kudla and K.G. Paterson. *Modular security proofs for key agreement protocols.* Advances in Cryptology – Asiacrypt 2005, LNCS 3788, 549–565, Springer, 2005.