Notes #9

The goal is for two parties to be able to spawn processes, and pairs of processes will invoke a protocol in order to share a session key. The goal of the adversary will be to learn something about the session key created by a process spawned by an honest party A who is trying to talk to another honest party B. We say he "challenges" this process: this means he is given either the actual session key or a random string, and he has to guess which. The adversary is allowed to look at any other output session key that he wishes to, except that he is not allowed to "cheat" by looking at a session key identical to that of the challenged process, that comes from a "matching" process (see below). The idea is that only "matching" processes should be able to succeed at sharing a session key, and session keys shared by good guys should look like random, independent strings, even to an adversary who has total control of the network.

Definition of Key Exchange in a PKI (public key infrastructure)

Before we define a rigorous notion of security, we have to state exactly what a key exchange protocol is.

Every party will have access to the security parameter n through the string 1^n.
A "person" will be an n-bit string corresponding to a unique "name" in a public key infrastructure table. The world will have good-guy (i.e. honest) persons that the adversary will be trying to break; good-guy people will be properly following their protocol, whereas bad-guy people can do whatever they like, and are really just pseudonyms for the adversary.

We have a key-generating probabilistic algorithm *GEN* which, on input 1^n, runs in polynomial time and outputs a long-term public key and a long-term private key: *pub* and *pri*.
Public and private keys will be generated for good-guy people using *GEN*. We assume that the length of these keys depend only on n. We assume the lengths of *pub* and *pri* depend only on n. Bad-guy people can choose their public keys however they like, and they can even choose keys that are similar or identical to other keys in the table. However, bad-guys must choose *unique* names.

Note that although names in the PKI table are unique, there is no real sense in which they actually identify anybody. Therefore, each table entry will not only have a name and a public key, but also a field that describes the individual in question.

Every role of a session key exchange protocol is a probabilistic, polynomial time algorithm.
A process spawned by a good-guy person, say A, has the following inputs:

- The name, A, of the spawning person. For convenience, we assume $A \in \{0,1\}^n$.
- The private key of A, pri_A.
- The name, say M, of the person the process is trying to share a key with, $M \in \{0,1\}^n$.
- The public key of M, pub_M. (This is public information, and we do not concern ourselves here with how A learns it.)
- The *role* of the process, a bit b. (See below for an explanation.)

We call such a process an A process of type $\langle A, M, b\rangle$. If M is also a good-guy, we call such a process a *good-guy process.*

Of course, A will have no idea whether M is a good-guy or a bad-guy.
We also allow the possibility that M is actually A, that is, A has spawned two processes that want to talk to each other.

In addition to its inputs, a process has a channel on which it writes strings, and a channel on which it reads strings. We will assume that the number of "flows" or rounds is fixed. We will also assume that whether a process reads or writes on a given flow depends only on the role bit. We will also assume that the number of bits read or written on each flow depends only on the security parameter 1^n and the role bit.

When a process is done interacting it will output either a special symbol FAIL – indicating that something has gone wrong, or a session key K. The length of K should be n. For convenience in describing certain protocols, we will allow it to be a fixed function of n, say $\ell(n)$; for example, $\lfloor n/2 \rfloor$.

We insist that, in the absence of an adversary, the protocol be correct, in the obvious sense. That is, say that people A and B have generated their keys correctly using GEN. Then if the processes $\langle A, B, 0\rangle$ and $\langle B, A, 1\rangle$ interact correctly using the protocol, then they both output session keys, and the keys they output are identical.

End of Definition of Key Exchange in a PKI (public key infrastructure)

The reader may ask how it came to be that A spawned a process that wants to talk to M, using role b. The answer is that it is not our concern. Somehow, perhaps using some (likely insecure) network communication, perhaps as the result of a phone call, or on a whim, A decided to do this. The reader may also ask how, once A and B decide to talk to each other, the two processes that they launch actually manage to communicate. In the absence of an adversary, this happens due to something called "the magic of the internet" and it is none of our concern.

We assume our adversary has complete control of the internet. He decides which processes to launch, and he reads whatever bit-strings they send, and he sends to each process whatever string he wishes. He even chooses everybody's name, subject to the condition that the names are unique. A good-guy chooses his keys according to GEN, after his name is chosen. A bad-guy's public key is chosen however the adversary likes. The adversary can only launch processes of the form $\langle A, B, b\rangle$ where A is a good-guy name. (It wouldn't help for the adversary to launch processes that begin with a bad-guy name, because the adversary can always mentally simulate those on his own.)

We have to explain the role of the "role" bit b. It may be useful for the key exchange itself, if the exchange is not symmetric. But there is another reason. When two processes share a session key, they intend to use it in some session. For this intended use to be convenient, they have to have some asymmetrical information. This information may be used to determine who starts the session, or to determine who is the server and who is the client, etc. This means that the role bit involves important security issues (see below). Again, the reader may ask how a process came to have a particular role bit, and again, it is none of our business.

Discussion, and definition of *Matching* processes

If A and B are good-guys, we say that $\langle A, B, b\rangle$ and $\langle B, A, \overline{b}\rangle$ are *matching* processes, where $\overline{b} = 1 - b$. Note that two matching processes must both be good-guy processes. An adversary should not be able to make two good-guy, non-matching processes share a key. For example, imagine how it could be bad if processes $\langle A, B, 0\rangle$ and $\langle B, A, 0\rangle$ shared a session key. Each one might begin

the session that follows by exclusive or'ing some message with the session key, or with a string pseudo-randomly generated from the session key. The adversary would then learn the exclusive-or of the two messages.

Example

Before defining security, it will be useful to give an example of a session key exchange protocol. It will turn out that this protocol is insecure.

As in the previous notes, let us, for convenience assume the following setting and the following assumption.

Setting:
For each n we have a fixed n-bit prime p_n and a generator g_n for $\mathbb{Z}^*_{p_n}$; everybody has access to these numbers. When appropriate, we consider a member of $\mathbb{Z}^*_{p_n}$ to be not only a number, but also a string of length (exactly) n.
We also have a polynomial time computable function $h : \{0,1\}^n \to \{0,1\}^{\lfloor n/2 \rfloor}$, that is known by everyone.

Assumption:
Let $\{D_n\}$ be a polynomial size family of circuits where D_n has $2n + \lfloor n/2 \rfloor$ input bits and 1 output bit.
Let $prob_D(n)$ be the probability that if x and y are randomly chosen from $\{0, 1, \ldots, p-2\}$, and u_0 is assigned $h(g_n^{xy} \bmod p_n)$, and u_1 is randomly chosen from $\{0,1\}^{\lfloor n/2 \rfloor}$, and bit is randomly chosen from $\{0,1\}$ and D_n is given $g_n^x \bmod p_n$, and $g_n^y \bmod p_n$, and the challenge $CH = u_{bit}$, then D_n outputs bit.
Then $prob_D(n) \leq \frac{1}{2} + \frac{1}{n^c}$ for each c and sufficiently large n.

We now give an example of a key exchange protocol. Say that the security parameter is 1^n and let $p = p_n$ and $g = g_n$.
Gen will create a public and private key exactly as in a secure signature scheme, where every signature has length n.

Here is an informal description of how A and B work if they wish to share a session key:
A computes $g^x \bmod p$ and signs it to B, and B computes $g^y \bmod p$ and signs it to A; then both of them output the $\lfloor n/2 \rfloor$ bit session key: $h(g^{xy} \bmod p)$.

Here is a more careful description of the protocol. When we write $SIGN_A(m)$, we mean the signature of m using A's private key, that is, $SIGN_{pri_A}(m)$. We will also use VER_A to denote VER_{pub_A}.

Protocol 1: Process $\langle A, B, b \rangle$ works as follows:

- Choose a random $x \in \{0, 1, \ldots, p-2\}$, compute $\alpha = g^x \bmod p$, and send out the $2n$-bit message: $[\alpha, SIGN_A(\alpha\, b)]$. (Note that A's private key is used to sign the $n+1$ bit string.)
- Receive a $2n$-bit message $[\beta, \sigma]$ where $|\beta| = |\sigma| = n$.
  Check that $\beta \in \mathbb{Z}^*_p$, and use B's public key to check that σ is a valid signature of $[\beta\, \overline{b}]$ (that is, check that $VER_B([\beta\, \overline{b}], \sigma) = 1$); if not, halt and output FAIL.
- Output $h(\beta^x \bmod p)$ as the session key.

Note that we have decided that both role-0 and role-1 parties begin by sending out a message, and then continue by receiving one. That is, neither party "goes first". We chose to do it this way

because it is easier to describe, and perhaps slightly more efficient. It also illustrates the important point that one should use the style given above to describe how a process behaves, namely, a description of when to send and when to receive and what to send and how many bits to receive and what to do with those bits one has received. The more typical way of describing a protocol is to *interleave* the description of what each of the two parties should do. This is easier to read and more intuitive, and for this reason is a very bad way to present a protocol: the intuitiveness implies constraints that may not be enforceable (such as supposed turn-taking), and masks the range of deceptions that an adversary might be performing.

Definition of Security of Key Exchange in a PKI

We now start to define security. We first describe the adversary ADV. He will be either uniform and probabilistic or nonuniform and deterministic, and (if uniform) will have as input the security parameter 1^n.

ADV will completely control the choice of names and the setting up and running of processes, in any order he likes. He will get information about which processes FAIL, and he will be able to open (i.e. see) any session key he likes. He will *challenge* exactly one good-guy process (i.e. both names are good-guys). that has completed without FAILing; for that process, he will be given either the session key or a random string, and he has to guess which. He is not allowed to challenge a process that has the same session key as some opened matching process, or open a process that matches and has the same session key as the challenged process.

We therefore give ADV access to an *Oracle*: For every two matching processes that have completed successfully (that is, not FAILed), the Oracle tells him whether or not they have computed the same session key. This seems like a lot of power to give to the adversary. However, since it makes our definition simpler, and since we can still prove that some protocols are secure, we assume ADV has this Oracle.

All this can happen in any order that makes sense, and ADV only makes his final guess about the challenge at the very end. In particular, he is allowed to use information he gets by running the processes in order to choose names of new people. He is allowed to use the challenge string in order to run processes in a particular way. (This last point is very important, and reflects the fact that in the real world, a session key may be used by a process as soon as it is selected, and this use may give our adversary useful information.)

In particular, ADV can do all of the following many times in any order, subject to the stated constraints:

- Adv chooses an n-bit name for a good-guy; this name must be different from all previously chosen names. For good-guy A, *GEN* is used to create a private key pri_A and a public key pub_A, and ADV is given pub_A.

- ADV chooses an n-bit name for a bad-guy; this name must be different from all previously chosen names. For bad-guy name U, ADV creates a public key pub_U. As part of the creation of this public key, ADV may choose to remember some additional private information.

  Of course, ADV may choose to create a public and private key using *Gen*, but he can do anything else he wishes. For example, he can choose pub_U to be equal to pub_A for some good-guy A, although of course he will not know pri_A.

  (If he wants to be really weird, ADV can choose a pub_U to resemble someone's name, or he can choose someone's name to resemble someone's public key!)

- ADV creates a process of type $\langle A, M, b\rangle^i$ where A is a good-guy name, $b \in \{0, 1\}$, and M is either a good-guy name or a bad-guy name, and i indicates that this is the i-th process of type $\langle A, M, b\rangle$ that ADV has established.

- If process $\langle A, M, b\rangle^i$ wants to write a message, then ADV can read that message.
  If process $\langle A, M, b\rangle^i$ wants to read a message, then ADV can send that process any message (of the appropriate length) that he wishes.

- Say that process $\langle A, M, b\rangle^i$ has read or written its last message and has therefore completed and decided on its output of either FAIL or a session key. ADV will see whether or not it has FAILed.

- For every pair of matching processes that have both completed without FAILing, Adv is told (by the Oracle) whether or not they have output the same session key.

- If process $\langle A, M, b\rangle^i$ has completed but not FAILed, ADV can choose to see its output session key; we say that ADV *opens* that session key. Note that ADV doesn't have to open this session key right away; he can do it any time he likes, or not at all.

  HOWEVER, ADV cannot open a process that matches and (according to the Oracle) has output the same session key as a previously challenged (see below) process.

- Exactly **once**, ADV will *challenge* a process. This must be a process $\langle A, B, b\rangle^i$ where B (as well as A) is a good-guy name, and the process must have completed without FAILing.

  When this process is challenged, the following will happen.

  A random bit *bit* is chosen (ADV doesn't see it);
  u_0 is assigned the session key K output by the process, and u_1 is assigned a random string (of the right length $\ell(n)$ for a session key); ADV is given the challenge string $CH = u_{bit}$. ADV's goal will be to eventually guess *bit*.

  HOWEVER, ADV cannot challenge a process that matches and (according to the Oracle) has output the same session key as a previously opened process.

When ADV has finished doing the above, he will output a guess bit' at *bit*, and we consider it to be a success if $bit' = bit$. We insist that for each such ADV and every d, the probability of success is $\leq 1/2 + 1/n^d$ for sufficiently large n.

End of Definition of Security of Key Exchange in a PKI

This definition is very complicated and subtle, and hard to absorb all at once. Frankly, I'm not completely sure it's correct. It is, however, the simplest definition I have seen for key exchange in this setting. This definition is a modification, due to Wesley George, of a more complicated definition from an earlier version of these notes.

It is useful to see why Protocol 1 described above is insecure according to this definition. Our ADV will work as follows:

- ADV first chooses two good-guy names A and B. (In fact, we can let $A = B$.)
  ADV also chooses a bad-guy name U, and chooses a public key pub_U and a corresponding private key pri_U using *GEN*.

- ADV creates processes $\langle A, B, 0\rangle$ and $\langle B, U, 1\rangle$. ADV will make these two non-matching processes output the same key. The trick is that whatever A signs to B, ADV can re-sign as U to C.

- ADV reads $[\alpha, SIGN_A(\alpha\, 0)]$ from $\langle A, B, 0\rangle$, and
  ADV reads $[\beta, SIGN_B(\beta\, 1)]$ from $\langle B, U, 1\rangle$.

- ADV sends $[\beta, SIGN_B(\beta\, 1)]$ to $\langle A, B, 0\rangle$.
  ADV sends $[\alpha, SIGN_U(\alpha\, 0)]$ to $\langle B, U, 1\rangle$.
  (Note that ADV can create $SIGN_U(\alpha\, 0)$ since ADV has seen α and ADV knows the private signing key pri_U corresponding to the public key pub_U that will be used by B to check the signature.)

- Both $\langle A, B, 0\rangle$ and $\langle B, U, 1\rangle$ output session keys. ADV knows they will be the same key.
  ADV chooses to open the session key K of $\langle B, U, 1\rangle$, and
  ADV chooses to challenge the session key of $\langle A, B, 0\rangle$, receiving challenge CH.

- ADV outputs 0 if $CH = K$, and 1 otherwise.

Note that $\langle A, B, 0\rangle$ and $\langle B, U, 1\rangle$ are *not* matching processes, so ADV is allowed to challenge one and open the other, and so he succeeds with probability (nearly) 1.

Note that we can make this adversary even simpler, by having him choose U's public key to be the same as A's. Then he can send exactly $[\alpha, SIGN_A(\alpha\, 0)]$ to $\langle B, U, 1\rangle$, without re-signing anything.

We now wish to fix this protocol. The problem, of course, was that the messages being signed did not contain their intended destinations. We therefore create the following improved protocol.

Protocol 2: Process $\langle A, B, b\rangle$ works as follows:

- Choose a random $x \in \{0, 1, \ldots, p-2\}$, compute $\alpha = g^x \bmod p$, and send out the $2n$-bit message: $[\alpha, SIGN_A(\alpha\, b\, B)]$. (Note that A's private key is used to sign the $2n+1$ bit string.)

- Receive a $2n$-bit message $[\beta, \sigma]$ where $|\beta| = |\sigma| = n$.
  Check that $\beta \in \mathbb{Z}_p^*$, and use B's public key to check that σ is a valid signature of $[\beta\, \overline{b}\, A]$; if not, halt and output FAIL.

- Output $h(\beta^x \bmod p)$ as the session key.

Theorem: Protocol 2 is secure.

Proof: We will outline a proof of this theorem.
Consider an adversary ADV that supposedly breaks the protocol.
Let 1^n be a security parameter for which he "succeeds" with probability $> 1/2 + \epsilon$ where ϵ is a non-negligible function of n.

The first question we ask is how likely it is that ADV forges a signature of some good-guy? That is, how likely is the event that ADV manages to create a string that verifies as the signature of a message m by a good-guy – say C – , when ADV has not previously seen that m signed by a C process? If this probability is at least ϵ, then we can use ADV to break the signature scheme. (This is left as an exercise for the reader.) Assume otherwise. It is convenient to assume that ADV *never* forges a signature. We can do this by changing ADV so that if he ever sees himself about

to (successfully) forge a good-guy signature, he instead flips a coin to determine his guess at the value of *bit*. This new adversary (we will still call him ADV) never forges, and he succeeds with probability $> 1/2 + \epsilon/2$.

We now wish to construct an adversary ADV′ that breaks the above **Assumption**.
Say that x and y have been chosen randomly from $\{0, 1, \ldots, p-2\}$ and ADV′ is given $g^x \bmod p$ and $g^y \bmod p$, as well as an appropriately chosen challenge string *CH*. Our goal is to output 0 if $CH = h(g^{xy} \bmod p)$, and to output 1 if *CH* was chosen randomly. Our approach will be to simulate ADV, tricking him into solving our problem

To do this, imagine that we could somehow guess that the process that ADV will challenge is $\langle A, C, b\rangle^r$ where A, C are good-guy names. We will simulate ADV, using *GEN* to choose both the public and private signature keys for the good-guys. We will then use our input $g^x \bmod p$ as the α generated by $\langle A, C, b\rangle^r$.

Consider the message $[\beta, \sigma]$ that $\langle A, C, b\rangle^r$ reads. Since $\langle A, C, b\rangle^r$ doesn't FAIL, it must be the case that σ verifies as a signature by C of $[\beta\, \bar{b}\, A]$. Therefore, since ADV never forges, there must have been a C process that signed $[\beta\, \bar{b}\, A]$, and because of the presence of $\bar{b}$ and A in the signed message, the signature must have come from a process of type $\langle C, A, \bar{b}\rangle$.

Imagine that we could somehow guess that this process was $\langle C, A, \bar{b}\rangle^s$. We will use our input $g^y \bmod p$ as the β that is generated by $\langle C, A, \bar{b}\rangle^s$ and given to ADV. Because we have generated the signature keys for the good-guy names, including A and C, we are able to simulate the signatures in the two processes $\langle A, C, b\rangle^r$ and $\langle C, A, \bar{b}\rangle^s$. Also, for every process other than these two – say a B process – we are able to simulate B by choosing a random $z \in \{0, 1, \ldots, p-2\}$, computing $g^z \bmod p$, and signing the appropriate message, and verifying the response.

When ADV eventually challenges $\langle A, C, b\rangle^r$, we give him our input *CH* as *his* challenge string. We will output as our answer whatever bit ADV outputs as his.

It remains to show how we can simulate the rest of ADV perfectly. The main issue is showing how to simulate ADV getting information from the Oracle and how to simulate ADV opening the session keys of various processes.

First, consider two matching (good-guy) processes $\langle D, E, b\rangle$ and $\langle E, D, \bar{b}\rangle$ that have successfully completed; under what conditions will they output the same session key? Since ADV cannot forge signatures, the $\langle D, E, b\rangle$ process must have received a signed group element chosen by some E process, and the $\langle E, D, \bar{b}\rangle$ process must have received a signed group element chosen by some D process. If they have received each other's group element (we know when this is the case), then they will output the same session key. Otherwise they almost certainly won't, for the following reason. If an adversary is given a small set of randomly chosen values from $\{0, 1, \ldots, p-2\}$, then it is unlikely there will be a_1, a_2, a_3, a_4 in the set such that $a_1 \neq a_2$, $a_3 \neq a_4$, $\{a_1, a_2\} \neq \{a_3, a_4\}$, and $h(g^{a_1 a_2} \bmod p) = h(g^{a_3 a_4} \bmod p)$, otherwise the **Assumption** would be false. (Exercise.)

We can therefore simulate the Oracle answers that ADV receives. How can we simulate ADV receiving the session key of a process? If the process is neither $\langle A, C, b\rangle^r$ nor $\langle C, A, \bar{b}\rangle^s$, then we know the exponent of the group element that the process has generated (since we chose it ourself), and so we can compute the session key output by the process. (Note that we can compute this session key, even if the other name in the process, the non-owner name, is a bad-guy.)

What if ADV wants to open $\langle C, A, \bar{b}\rangle^s$? If this happens, then it must be the case that the session key $\langle C, A, \bar{b}\rangle^s$ outputs is different from that of $\langle A, C, b\rangle^r$, and that therefore the group element $\langle C, A, \bar{b}\rangle^s$ receives must have come from some $\langle A, M, b\rangle^i$-process other than $\langle A, C, b\rangle^r$. Since we chose the exponent of the $\langle A, M, b\rangle^i$ process ourself, we can use it to compute the session key output by $\langle C, A, \bar{b}\rangle^s$.

Therefore we succeed with the close to same probability that ADV does: $> 1/2 + \epsilon/2 - \delta$. where our error δ, due to our imperfect simulation of the same-key oracle, is less than $1/n^d$ for every d and sufficiently large n.

However, all the above depended upon our choosing A, C, b, r, s correctly. So we will try to guess these values. Actually, we cannot really hope to guess a name A and be right with significant probability, since (for example) ADV may choose these names randomly. Instead, let us denote the i-th good-guy name chosen by ADV as N_i. So we can guess i, j, b, r, s such that $\langle N_i, N_j, b\rangle^r$ is the process ADV will challenge and $\langle N_j, N_i, \bar{b}\rangle^s$ is the process that the challenged process will receive a signed message from. This guess will be right with probability $1/n^d$ (for some d). We will be able to tell during the simulation whether or not our guess is correct; if we determine that our guess was not correct, we will flip a coin to determine our answer. The simulation of the adversary is perfectly correct independently of our guess, so the probability that ADV will succeed given that our guess is correct is $> 1/2 + \epsilon/2$.

Therefore we succeed with probability $> (1 - 1/(n^d)) \cdot 1/2 + (1/n^d)(1/2 + \epsilon/2) = 1/2 + \epsilon/(2n^d)$.□

It is interesting to consider what would happen if we omitted the role-bits inside the signed messages in Protocol 2. It is not hard to see that if we did this, then an adversary could cause two good-guy processes $\langle A, B, 0\rangle$ and $\langle B, A, 0\rangle$ to share a key. (Exercise!) These are not matching processes. Therefore, an adversary could correctly answer a challenge on one of the session keys by opening up the session key of the other process.

It is important to appreciate that the insecurities we have demonstrated in variants of Protocol 2 are not merely weird artifacts of a strange definition. Within the cryptographic community they are usually considered to be fatal flaws.

Some people in that community, however, would consider that Protocol 2 fails to have an additional property that they think session key exchange protocols should have.

Note that an adversary can send to an $\langle A, B, 0\rangle$ process an old, signed message from a $\langle B, A, 1\rangle$ process – perhaps one that was signed years ago. So the $\langle A, B, 0\rangle$ process generates a session key that he incorrectly thinks he has shared with a $\langle B, A, 1\rangle$ process. (Similarly, an adversary can send an old signed message to a $\langle B, A, 1\rangle$ process.)

Now to a certain extent, this problem is unavoidable. We would ideally like it to be the case that whenever a process generates a session key, then he is sure that a matching process has generated the same key. This is impossible. *Someone* has to go last in a protocol, and if the adversary cuts off this last message, then only one of the two processes will generate a session key.

One can, however, insist on something like the following conditions, at the cost of having a more complex protocol with three (instead of two) "flows":
If a role-0 process outputs a session key, then there is a matching role-1 process that has output the same key. Also,
if a role-1 process outputs a session key, then there is matching role-0 process who has sent him something at some point since he started the protocol.

Protocol 2 clearly lacks both of these conditions. In my opinion, however, this does not matter because I regard these conditions as extremely arbitrary. For instance, even if both processes really do share a session key, there is nothing to stop an adversary from cutting a line so that in effect one (or both) are no longer present. If you are concerned about the presence of your matching party, then you should use the session key (in a secure way, as described in previous notes) with

the convention that from time to time each person is expected to say something (like "uh-uh" in a telephone conversation) to prove he is still there.

Our definition of security assumes that an adversary has not broken into a good-guy's computer. Often one desires that a key exchange protocol retain *some* security even in the case of such a break-in.

Protocol 2, in fact, happens to possess such a feature called "Forward Security". The point is that in some sense, it is secure against an adversary who at some time in the future breaks into your computer and learns your PKI private key. Of course, from that time on you may have no security, in the sense that the adversary can impersonate you. But if all your session keys from the past have already been used, and those sessions are over, and the session keys and session information have been deleted from your computer, then nothing is learned about those past sessions by breaking into your computer and seeing your PKI private key. This is the case even if the adversary had been listening in on and recording those sessions, or even interfering in them. The reason Protocol 2 has this property is that the PKI private key is used only for signing, and not for encrypting. If the adversary learns your private key later it gives him absolutely no useful information about keys that have been exchanged in the past.

Protocol 2 also happens to be secure against "Key Compromise Impersonation": although an adversary who obtains your PKI private key can now impersonate you to another good-guy, he *cannot* impersonate another good-guy to you.

Of course, we have made no attempt here to give careful definitions of these additional security properties.