Computer Science 2426F — Fall, 2020
St. George Campus — University of Toronto

Notes #7.5

Session Key Agreement
Shared Long-Term Key Setting

We have already seen how two parties can have a secure session over a totally insecure channel, given that they initially share a random key. Here, we view that key as being used to generate a key for A to talk to B, and a key for B to talk to A.

There are a number of ways they might share that key.

- They can physically get together and flip some coins to choose the shared key.
- The other ways involve a pre-existing infrastructure. One example is that the two parties have a long-term shared key, that they then use whenever they want to establish a new session key. This setting is the topic of this set of notes.
- Another infrastructure is a public key infrastructure (PKI), where each party has a public key that everyone knows, and a private key that only he knows; these master keys are then used to do a *session key agreement* (or exchange) protocol. One thing that is needed for this is a *public key encryption primitive.* (Session key agreement is in fact the main use of public key encryption, at the moment.) This kind of key exchange is one mode of the internet standard TLS, and it will be discussed in later notes.
- Another TLS mode of key exchange is where one party has public key credentials and the other party has no credentials. This is typically used when an individual, such as you, wants to communicate with an institution, such as a bank or online merchant. The secure session that is thus established typically begins with the individual sending his login name and password to the institution. Although this is obviously incredibly common, I know of no good definition of security for it in the literature, and I will not discuss it further here.

For now, we will consider the simplest non-trivial key agreement setting. In this setting, two people share a *long-term* master key K. From time to time, they wish to engage in a secure session using a new, random-looking session key. They may even engage in a number of sessions at the same time, agreeing on a new key for each session.

In case this sounds abstract, please notice that many of us engage in such a key agreement protocol quite often. I do, every time I enter my house! This is because the wireless router for my home network has a long-term master key K (that I chose and entered at an earlier date), and my phone has that same key K (that I entered at an earlier date) associated with that network . When I enter my house my phone detects my wireless network and performs a session-key agreement protocol with my router, agreeing on a key S that is then used for a session with my router. I may also then use my tablet to access the internet, as well a my wireless laptop. So in one afternoon I've engaged in three session-key agreement protocols, or at least some devices have done that on my behalf. The most common protocol used is WPA2.

Who is the adversary? The adversary may be that guy sitting in a car outside my house with lots of expensive gear on the roof. He can interfere however he likes with the session key agreement protocols and with session protocols, and he can slow them down or stop them, but he shouldn't be able to learn anything about the session content (other than the rate at which bits are flowing) or change the session content.

What if the adversary *is* able to listen to all the content between me and my router? He will learn what web sites I am talking to; if those web sites are not "secure" (https), he will hear the content of my communications with them; he will also hear the private communication going on within my wireless network – file sharing, for example.

Why not use the long-term key K as every session key? After all, it's randomly chosen and unknown to the adversary. Recall that our definition of secure session protocol assumed that the session key was chosen randomly, independently of everything else. If the same session key were used elsewhere, the adversary might learn something about it (since he might know something about the message), and so it might not look random any more. So we need each session key to look random independently of all the other session keys. (As an exercise, the reader can consider one of our secure session protocols, and what would go wrong if the same key were reused.)

This leads to our definition of security for a key agreement protocol. The adversary will choose one of the session keys that are output, and try to distinguish it from a random string, even given all the other session keys that are output. To understand this more precisely, it would be a good idea to explain exactly what a session key agreement protocol is, and to see an example.

One thing to keep in mind is the issue of how two parties use a session key S to talk to each other. They must derive two random-looking keys from S, S_0 and S_1, that they then use to talk to each other; but who should use S_0 to talk, and who should use S_1 to talk? It would be very bad if they both used S_0 talk. This could be negotiated securely as part of the key exchange, but it is cleaner to assume that each process participating in the exchange knows what role – 0 or 1 – it is supposed to play. This can be done by first engaging in a simple insecure protocol of some kind; for example, the "initiator" can be role 0. Or if the protocol is, say, router/client, we can let the router be role 0 and the client be role 1.

It would be very bad if the adversary could trick two role 0 processes, or two role 1 processes, into sharing a key!

A key exchange protocol is then specified by specifying how a 0-process behaves and how a 1-process behaves. Rather than thinking of these processes as "people", think of them as being spawned by people, devices, etc. Each process will be given the security parameter n given as, say, 1^n. Each process will also be given the long term n-bit key K. If a process terminates successfully, it creates, or outputs, an n-bit session key.

Here is an example of a key agreement protocol. The protocol will use a pseudo-random function generator F. Informally, it works as follows.
Each party will send a random n-bit string to the other;
the session key will be the exclusive-or of the results of applying F_K to the two random strings.
More formally, we describe the protocol as follows.

Protocol 1

Process $\langle 0 \rangle$ works as follows:

- Choose random $r \in \{0,1\}^n$;
  send out r.
- Receive n-bit message t.
- Output $F_K(r) \oplus F_K(t)$ as the session key.

Process $\langle 1 \rangle$ works as follows:

- Receive a n-bit message s.
- Choose a random $u \in \{0,1\}^n$;
  send out u.
- Output $F_K(s) \oplus F_K(u)$ as the session key.

Note that each process has an output channel that it writes to, and an input channel that it reads a desired number of bits from (if enough bits aren't there, it waits for them to come). We also assume, for simplicity, that for each process, the number of rounds, and the number of bits sent or received at each round, is independent of the security parameter n. We insist on *correctness*, that is, in the absence of an adversary, where the processes read and write to each other as intended, the two processes output the same session key; clearly Protocol 1 satisfies this.

Should this protocol be considered secure? Clearly a *passive* adversary that only listens to the communication cannot learn anything about the session keys (we're assuming F is pseudo-random). What about an *active* adversary that can read from and write to every process? Here is something an active adversary can do. After reading a string r from a $\langle 0 \rangle$ process, send the same string r back to the process; the process will then create the session key $\bar{0}$, an obvious insecurity.

What if we changed the protocols slightly so that the $\langle 0 \rangle$ process will not accept if $t = r$, but will *fail* instead (we will talk about failure later)? This would still be insecure, as follows: the adversary would read r_1 from one $\langle 0 \rangle$ process and read r_2 from another $\langle 0 \rangle$ process, and then would send r_2 to the first process and r_1 to the second process; the two $\langle 0 \rangle$ processes would then create the same session key! This is very bad, since the session keys of two clients should be independently random. In the more precise security definition below, the adversary will be able to distinguish one of them from random, after seeing the other one.

What if we changed the protocols in a different way? Let us say that pseudo-random function generator F is such that F_K maps $2n$-bit strings to n-bit strings We then change the $\langle 0 \rangle$ process to create the session key $F_K(r\,t)$, and similarly change the $\langle 1 \rangle$ process to create the session key $F_K(s\,u)$.

Protocol 2

Process $\langle 0 \rangle$ works as follows:

- Choose random $r \in \{0,1\}^n$;
  send out r.
- Receive n-bit message t.
- Output $F_K(r\,t)$ as the session key.

Process $\langle 1 \rangle$ works as follows:

- Receive n-bit message s.
- Choose a random $u \in \{0,1\}^n$;
  send out u.
- Output $F_K(s\,u)$ as the session key.

Is this secure? Before answering this question, we need a more precise definition of security.

Definition of Security

What is the probabilistic, polynomial-time adversary (call him ADV) allowed to do? He can read from and write to processes, but where do those processes come from? Of course, ADV creates them. He creates as many $\langle 0\rangle$ processes and $\langle 1\rangle$ processes as he likes (in polynomial time of course). These processes will all use the same master key K, which is chosen randomly and which ADV can't see. ADV reads whatever is sent from the processes he creates, and when a process wants to read something, ADV sends it whatever he likes. Because of the fixed form of the processes, ADV knows when a process terminates. Sometimes a process may discover that something is wrong. For example, a signature may fail to verify, or a check such as $t \neq r$ fails, as we saw above. In this case, instead of outputting a session key, the process will output the symbol FAIL. ADV will be told if the process has successfully terminated and output a session key, or if it FAILed.

At any time after a process successfully terminates, ADV may choose to see the outputted session key. He may, instead, at any time, choose to *challenge* that key. *Challeng*ing a key means he is given either a random n-bit string or the created session key (each with probability $1/2$), and he will try to say which – say 1 if he is given the correct string, and 0 otherwise. Before answering, he may continue to interact with other processes, and may open (choose to see) other processes' outputted session keys. We say the protocol is *secure* if for every such adversary and every c, for sufficiently large n, the probability he succeeds at guessing the right bit is $\leq 1/2 + 1/n^c$.

BUT there is a big flaw in the above definition! ADV can trivially break *every* protocol. All he has to do is create a $\langle 0\rangle$ process and a $\langle 1\rangle$ process, and simulate them sending messages back and forth as in the protocol; that is, whatever the $\langle 0\rangle$ process writes is sent to the $\langle 1\rangle$ process, and vice versa. Then the two processes will output the same session key, and all ADV has to do is challenge one of them and open the other one, and he will learn exactly what the challenged key is.

So we want to modify the power of ADV in the above definition in as minor a way as possible so as to disallow the above attack. If ADV has challenged a process of type $b \in \{0, 1\}$, he is not allowed to open the session key of a type $1 - b$ process that has created *exactly same key*; if ADV has opened the session key of a type b process, he is not allowed to challenge a process of type $1 - b$ that has created *exactly same key*. This, however, raises the question of how does ADV know when a particular $\langle 0\rangle$ process has created the same session key as a particular $\langle 1\rangle$ process, so that he can follow this constraint? We will therefore give ADV some additional power: for every $\langle 0\rangle$ process and $\langle 1\rangle$ process that have both successfully terminated, ADV is *told* whether or not they have created the same session key; say he is told this by the *same-key* oracle. This is our final definition. Since we can prove some protocols secure according to this definition, the only possible problem with giving ADV this additional power is that there may be a protocol that is insecure according to this definition, but that we wish to consider secure. However, after a great deal of thought, this seems to me unlikely.

It turns out that according to this definition, Protocol 2 is secure. The proof is not difficult, and is left to the reader.

Instead of proving this, we shall do a more difficult proof of a more complicated protocol. Protocol 3 is closer to the one actually used in the WPA2 standard for wireless networking, and has more in common with the TLS protocol that is used in the PKI setting discussed in later notes. We assume that F and G are pseudo-random function generators such that for an n-bit key K, F_K maps n-bit strings to n-bit strings and G_K maps $2n$-bit strings to n bit strings. We view the long-term shared key as a pair K_1, K_2 of n-bit keys.

Informally, the protocol works as follows.

Process $\langle 1 \rangle$ receives a random string from $\langle 0 \rangle$;
it then chooses a random string β and lets the session key be $F_{K_1}(\beta)$;
it then sends to $\langle 0 \rangle$: β and a signature (or MAC) using G_{K_2} of β, together with the random string sent by $\langle 0 \rangle$.

Protocol 3

Process $\langle 0 \rangle$ works as follows:

- Choose a random $r \in \{0,1\}^n$;
  send out r.
- Receive n-bit messages α, γ;
  check that $\gamma = G_{K_2}(\alpha\, r)$,
  otherwise FAIL.
- Output $F_{K_1}(\alpha)$ as the session key.

Process $\langle 1 \rangle$ works as follows:

- Receive a n-bit message s.
- Choose random $\beta \in \{0,1\}^n$;
  let $\delta \leftarrow G_{k_2}(\beta\, s)$;
  send out β, δ;
- Output $F_{K_1}(\beta)$ as the session key.

Proof that if F and G are pseudo-random, then Protocol 3 is secure.

Imagine the protocol was run where, instead of using F_{K_1} and G_{K_2} for randomly chosen K_1 and K_2, the parties used instead randomly chosen functions $f : \{0,1\}^n \to \{0,1\}^n$ and $g : \{0,1\}^{2n} \to \{0,1\}^n$. If there were an adversary that breaks the original protocol but not the revised one, then it can be used to break the pseudo-randomness of either F or G; the proof of this is left as an exercise. So it is sufficient to show that the revised protocol, Protocol 3′, is secure.

Protocol $3'$

Process $\langle 0 \rangle$ works as follows:

- Choose a random $r \in \{0,1\}^n$;
  send out r.
- Receive n-bit messages α, γ;
  check that $\gamma = g(\alpha\, r)$,
  otherwise FAIL.
- Output $f(\alpha)$ as the session key.

Process $\langle 1 \rangle$ works as follows:

- Receive a n-bit message s.
- Choose random $\beta \in \{0,1\}^n$;
  let $\delta \leftarrow g(\beta\, s)$;
  send out β, δ;
- Output $f(\beta)$ as the session key.

Imagine polynomial-time ADV runs, and reads from, and writes to, a bunch of $\langle 0 \rangle$ processes and $\langle 1 \rangle$ processes, and answers the correct bit with probability p. We will create a polytime, probabilistic algorithm ADV′ that can do the following given oracles for random functions $f : \{0,1\}^n \to \{0,1\}^n$ and $g : \{0,1\}^{2n} \to \{0,1\}^n$: ADV′ will query f and g on various points, eventually choosing a point u that f has not been queried at; ADV′ will then be given (with probability 50/50) either $f(u)$ or a random string, and will have to tell which (after further querying which does not include querying f at u). ADV′ will succeed with probability very close to p; Since ADV′ can only succeed with probability 1/2, this will show that p is exponentially close to 1/2.

At first it seems very easy to do this: we just have ADV′ simulate ADV and the processes that ADV creates. To do this, we have to be able to simulate the information that ADV receives.

- Whenever ADV runs a process that needs a g value, ADV′ queries g. Note that this allows ADV′ to tell when a $\langle 0 \rangle$ process FAILS.
- Given a $\langle 0 \rangle$ process and a $\langle 1 \rangle$ process that have terminated successfully, we have to simulate the same-key oracle and tell ADV whether or not they have created identical session keys. Say that the $\langle 0 \rangle$ process writes r and receives α,γ; say that the $\langle 1 \rangle$ process receives s and writes β,δ; Since ADV will not be able to find two inputs on which f yields the same output (except for exponentially small probability), these two processes will only output the same session key if $\alpha = \beta$.
- Whenever ADV wants to open a process (that is, see the session key output), ADV′ merely queries f at the appropriate point. The problem is that if ADV challenges a process that creates session key $f(u)$, in order for ADV′ to do what we want, we have to make sure that no process that ADV opens (which is the only way ADV′ sees f values) either before or after the challenge, outputs as the session key $f(u)$. We will deal with this issue after seeing more details of the simulation.

Details of simulation to show Protocol 3' is secure.
ADV$'$ has access to random f and g, and for some u, ADV$'$ wants to distinguish between $f(u)$ and a random string (without querying $f(u)$).

ADV$'$ will use ADV, f, g, and maintain descriptions of processes, as follows:

When ADV creates a 0 process, say $\langle 0\rangle^i$,
ADV$'$ will create a 0 process $\langle 0\rangle^i$.

When ADV creates a 1 process, say $\langle 1\rangle^j$,
ADV$'$ will create a 1 process $\langle 1\rangle^j$.

When ADV wants to read an n-bit string from a 0 process,
ADV$'$ randomly chooses an n-bit string and gives it to ADV.

When ADV sends two n-bit strings to a 0 process,
ADV$'$ simulates that same process receiving those two strings, and then does the check; if the check fails, ADV$'$ gives to ADV the fact that that process has FAILed.

If and when ADV "opens" a 0 process,
ADV$'$ queries f on the appropriate α and sends the result to ADV.

When ADV sends an n-bit string to a 1 process,
ADV$'$ simulates that same process receiving that string.

When ADV wants to read two n-bit strings from a 1 process that has previously received a string, say s,
ADV$'$ will randomly choose an n-bit β, query g on βs, and give β and $g(\beta s)$ to ADV.

If and when ADV "opens" a 1 process,
ADV$'$ queries f on the appropriate β and sends the result to ADV.

If ADV wants to know for a particular 0 process and a particular 1 process that have successfully completed and have not been opened, whether they have created the same session key,
ADV$'$ will see if they have applied f to the same string; if so, ADV$'$ will give "yes" to ADV, else ADV$'$ will give "no" to ADV.

If and when ADV "challenges" a process (that has not been opened) with session key $f(u)$
ADV$'$ announces that he wants to distinguish between $f(u)$ and a random string; ADV$'$ is then given an n-bit string x, which he gives to ADV as the challenge for ADV.

When ADV responds, finally, to his challenge with bit b,
ADV$'$ responds to HIS challenge with the same bit b.

Recall that we have to deal with the following problem:
Whenever ADV wants to open a process (that is, see the session key output), ADV$'$ merely queries f at the appropriate point. The problem is that if ADV challenges a process that creates session key $f(u)$, in order for ADV$'$ to do what we want, we have to make sure that no process that ADV opens (which is the only way ADV$'$ sees f values) either before or after the challenge, outputs as the session key $f(u)$.

Say that ADV challenges a $\langle 1\rangle$ process that receives s and writes β,δ, and outputs $f(\beta)$. ADV is not allowed to open (or to have opened) any $\langle 0\rangle$ process that outputs the same key. Can ADV open another $\langle 1\rangle$ process that chooses the same value for β on the second step? No (except for exponentially small probability), because every $\langle 1\rangle$ process chooses an independently random β.

Say instead that ADV challenges a $\langle 0\rangle$ process, say $\langle 0\rangle^1$, that writes r and receives α,γ, where $\gamma = g(\alpha, r)$, and outputs $f(\alpha)$. Since g is randomly chosen, in order for ADV to know the value of $g(\alpha, r)$, ADV must have previously sent r to a $\langle 1\rangle$ process, say $\langle 1\rangle^1$, that then chooses random string α.

ADV is not allowed to open (or to have opened) any $\langle 1\rangle$ process that outputs the same key. Can ADV open another $\langle 0\rangle$ process that outputs the same session key, $f(\alpha)$? Consider a different $\langle 0\rangle$ process, say $\langle 0\rangle^2$, that outputs $f(\alpha)$; this process will (almost certainly) choose a string $r' \neq r$ in the first step, and then receive α, γ' such that $\gamma' = g(\alpha, r')$. Since g is randomly chosen, in order for ADV to know the value of $g(\alpha, r')$, ADV must have seen a *different* $\langle 1\rangle$ process, say $\langle 1\rangle^2$, that received r' and then (randomly) chose α, the same string chosen in $\langle 1\rangle^1$. This can't happen except with exponentially small probability.

It is not hard to see with a more careful analysis, that for some constant c, the probability p with which ADV succeeds in breaking Protocol 3$'$ is within $n^c/2^n$ of 1/2, the probability with which ADV$'$ succeeds (where c depends on the running time of ADV). Using the pseudo-randomness of F and G, the probability an adversary can break Protocol 3 is within $1/n^d$ of 1/2 for every d and sufficiently large n.

The reader may note that although Protocol 2 and Protocol 3 are both secure, Protocol 3 has an additional *Property* that some may find desirable. In Protocol 3, whenever a $\langle 0\rangle$ process creates a session key, there must (except with negligible probability) be a $\langle 1\rangle$ process that outputs the same session key. It's not clear why this *Property* is important. Of course, we could have rearranged the protocol to have *Property$'$*: whenever a $\langle 1\rangle$ process creates a session key, there must (except with negligible probability) be a $\langle 0\rangle$ process that outputs the same session key. However, it is not possible to achieve *both* of these properties, since a process that sends last can not be sure that the message was received.

Possible problems with Protocol 3

- If a $\langle 0\rangle$ process terminates without FAILing, it knows that a $\langle 1\rangle$ process has terminated with the same session key. However, when a $\langle 1\rangle$ process terminates, it doesn't even know if it has been receiving information from a $\langle 0\rangle$ process that has been active since the $\langle 1\rangle$ process started.

- In practice, the K_1, K_2 are generated from a "pass phrase" that is chosen by the owner of the router. What if the pass phrase is chosen poorly, for example, as a random 9-digit string? There are only a billion such strings. But it is not necessary that an adversary try to connect to the network a billion or so times, which would be inconvenient and which might be detected. Instead after passively listening in to one legitimate connection and learning the value, for some α and r, of $G_{K_2}(\alpha r)$, an adversary can try all billion such strings **OFF-LINE**: for each 9-digit x, compute K_2 and see if $G_{K_2}(\alpha r)$ has the right value.

- Imagine the previous hack isn't applicable. An adversary can do the following:
  Record a key exchange and the subsequent session; at some later point, break into a house or steal a phone, and learn the pass phrase; then, from the recorded key exchange, learn the session key exchanged, and then use that key to learn everything said in the recorded session.

The first problem can be ameliorated by adding a third flow to the system where the $\langle 0 \rangle$ process sends an acknowledgment to the $\langle 1 \rangle$ process, say $F_{K_3}(\alpha)$. Now when a $\langle 1 \rangle$ process terminates without FAILing, it knows that a $\langle 0 \rangle$ process has terminated with the same session key. Also, when a $\langle 0 \rangle$ process terminates, it knows that it has been receiving information from a $\langle 1 \rangle$ process that has been active since the $\langle 0 \rangle$ process started, although it cannot know if its last message has been successfully delivered, and so it cannot know if a $\langle 1 \rangle$ process has terminated with the same session key.

By using public-key encryption, WPA3 (coming soon to a router near you!) claims to avoid the last two problems.