6.897: Selected Topics in Cryptography
Lectures 13 and 14

Lecturer: Ran Canetti

Highlights of last week's lectures

- Showed how to realize F_{zk} in the F_{com}-hybrid model.
- Showed how to realize any "standard" functionality:
  - In the F_{auth}-hybrid model, for semi-honest adversaries.
  - in the (F_{auth},F_{crs})-model, for Byzantine adversaries.
  - In the F_{auth}-hybrid model, for Byzantine adversaries with honest majority.

This week:

- Universal composition with joint state: motivation, formulation, proof, uses.
- UC formulation of signature schemes:
  - The signature functionality, F_{sig}.
  - Equivalence with CMA-security
- Achieving authenticated communication: Realizing F_{auth} given F_{sig} and certification authorities.

Yoav (after last lecture): "Gosh, we would need a really long reference string to realize functionalities this way…"

Indeed, a naïve use of the CRS would mean:

- A copy of F_{crs} per copy of F_{com}.
- O(k) copies of F_{com} per copy of F_{zk}.
- O(r) copies of F_{zk} per copy of F_{cp} to compile a protocol with r rounds.
- O(n) copies of F_{cp} in a protocol for n parties…

Is it really necessary to use so many copies of the CRS?

First answer: We can realize all the commitments using a single copy of F_{mcom}, which in turn uses a single copy of F_{crs}.

But, if we do that then we need to analyze the entire multiparty protocol (including all copies of F_{zk}, F_{cp}, etc.) as a single unit. This does away with much of the benefits of the UC theorem...

In fact, this is not specific to the UC framework: whenever we analyze multiple protocols that have a common subroutine, we analyze them in one piece. (Examples: multiple copies of NIZK over a single CRS, or multiple key exchange sessions using a single long-term authentication module [BR93])

Can we do better?

Can we continue writing and analyzing single-instance functionalities, and still have them use some joint state and randomness?

A more abstract view

We have:

- A protocol Q in the F-hybrid model for some F, that uses multiple independent copies of F.
  (e.g., F is F_{com}, and Q is the Blum ZK protocol, or multiple copies of it.)
- A protocol P that realizes (in one instance) multiple independent copies of F.
  (e.g., P is the protocol that realizes F_{mcom}.)

Can we compose them while maintaining security?

A formalization

- Multi-instance extensions of ideal functionalities:

 Let F be an ideal functionality. Then the multi-session extension of F, denoted FF, proceeds as follows:

- FF runs multiple copies of F. Each copy has its own identifier, denoted *ssid*.
- FF expects all its inputs to be of the form (sid,ssid,…), where sid is the session id of FF, and ssid is an identifier for the copy of F run within FF. An incoming message with ssid s is then forwarded to the copy of F with ssid s. If no such copy exist, then a new one is invoked and is given ssid s.
- Whenever a copy ssid of F within FF wishes to generate output to some party, or send a message to the adversary, then FF prepends its sid to the message and forwards it to the said recipient.

Example: F_{mcom}

1. Upon receiving (sid,cid,C,V,“commit”,x) from (sid,C), do:
   1. Record (cid,x)
   2. Output (sid,cid,C,V, “receipt”) to (sid,V)
   3. Send (sid,cid,C,V, “receipt”) to S
2. Upon receiving (sid,cid“open”) from (sid,C), do:
   1. Output (sid,cid,x) to (sid,V)
   2. Send (sid,cid,x) to S

F_{mcom} is the multi-session extension of F_{com} (ie, F_{mcom}=FF_{com}).

Example: F_{crs} (with distribution D)

1. Upon receiving (sid,ssid,pid,“crs”) from (sid,pid), do:
    1. If there is a recorded pair (ssid,v) then output v to (sid,pid) and send (ssid,pid,v) to the adversary.
    2. Else, choose a value v from D, record (ssid,v), and continue as in Step 1.1.

The composition operation:
Universal Composition with Joint State (JUC)

Start with:

- A protocol Q in the F-hybrid model (that may run multiple copies of F).
- A protocol P that securely realizes FF.

Construct the composed protocol $Q^{[P]}$:

- At the first activation of $Q^{[P]}$, each party invokes a copy of P with some fixed sid (say, sid=0).
- Whenever protocol Q calls a copy of F with input (sid=s,x), $Q^{[P]}$ calls P with input (sid=0,ssid=s,x).
- Each output (0,s,y) of P is treated as an output (s,y) coming from the copy of F with sid=s.

Theorem [JUC: UC with joint state]:

Let Q be a protocol in the F-hybrid model, and let P be a protocol that securely realizes FF. Then protocol $Q^{[P]}$ emulates protocol Q.

That is: for any adversary A there exists an adversary H such that for any environment Z we have: $EXEC^{F}_{Q,H,Z} \sim EXEC_{Q[P],A,Z}$.

Corollary:

If Q securely realizes some ideal functionality G then so does protocol $Q^{[P]}$.

Application of the JUC theorem to the construction of [CLOS]

Here F is F_{com} and FF is F_{mcom}:

- Can write and realize each functionality (ZK,C&P,general compiler) for a single instance.
- Can use the UC theorem to obtain a composed protocol Q in the F_{com}–hybrid model. Protocol Q uses many copies of F_{com}.
- Can then use the JUC theorem to compose Q with a single copy of the protocol that realizes F_{mcom}, thus using only a single copy of the CRS.

Proof of the JUC theorem:

Plan:

Define a protocol Q' in the FF-hybrid model, and show:

- Protocol $Q^{[P]}$ is identical to protocol Q'^{P}.
- Protocol Q'^{P} emulates protocol Q'.
- protocol Q' emulates protocol Q.

Protocol Q' (in FF-hybrid model):

Identical to protocol Q, except:

- Q' uses a single copy of FF, with sid 0.
- Any input x of Q to copy s of F is replaced by a call (0,s,m) to FF.
- Any output (0,s,y) from FF is treated as an output y coming from copy s of F.

We have:

- Protocol Q'^{P} emulates protocol Q' (from the UC thm).
- However, protocol Q'^{P} is only a different way of describing protocol $Q^{[P]}$.
- Thus, protocol $Q^{[P]}$ emulates protocol Q'.

Remains to show: Q' emulates Q.

Let A' be an adversary interacting with Q' in the FF-hybrid model. Construct an adversary A that interacts with Q in the F-hybrid model, and show that $EXEC^{FF}_{Q',A',Z} \sim EXEC^{F}_{Q,A,Z}$ for all Z.

Adv. A runs A':

- Messages sent by A' to parties running Q' are forwarded to the actual parties running Q.
- Messages from the parties running Q are forwarded to A'.
- For each message (0,s,m) sent by A' to FF, A sends the message (s,m) to copy s of F.
- Whenever A gets a message m from a copy of F with sid s, it forwards a message (0,s,m) from FF to A'.
- Whenever A' corrupts a party, A corrupts the same party and reports the obtained information to A'.

Validity of the simulation is straightforward… ■

How about general protocols in the CRS model?

Motivation: Assume we had a protocol that realizes FF_{crs} in the F_{crs}–hybrid model, using only a single copy of F_{crs}. Then it would suffice to construct only single-instance protocols, even in the CRS model. (For instance, realizing F_{com} would be enough, and we wouldn't need F_{mcom} …)

Results [CR03]:

- Any protocol that realizes FF_{crs} in the F_{crs}–hybrid model, using only a single copy of F_{crs} , must be interactive (ie, each party should send at leasy one message).
- Using the Blum 3-move coin-tossing protocol, can realize FF_{crs} in the F_{mcom}–hybrid model, using only a single copy of F_{crs}. Using protocol UCC, we get the desired result. (But we didn't get rid of protocol UCC…)

Application of the JUC theorem to signature-based protocols

Another case where multiple protocol instances use the same subroutine is the case of protocols based on signature schemes:

- Signature-based message authentication
- Signature-based key-exchange
- Signature-based Byzantine Agreement

In all these cases, protocols use long-term signature keys for multiple protocol sessions.

Goal: Define and analyze such protocols for a single session (ie, a single session-key) and then use JUC for deduce that the multi-session interaction (using a single long-term signature module) is secure.

To do that, need to be able to formalize the signature mechanism as an ideal functionality.

Digital signatures as an ideal functionality

Digital signatures are typically thought of as a tool within protocols, rather than a "protocol" by itself. But it's useful and instructive to treat digital signature as a protocol, with a specified ideal functionality. Potential benefits:

- Modularity of analysis (e.g., applying the JUC theorem).
- Re-asserting the adequacy of existing notions of security.
- Provide a bridge to formal analysis of protocols.

But, how to formalize?

There are two main approaches:

- Define signatures as a stand-alone primitive [C01,C-Krawczyk02,C-Rabin03,Backes-Hofheinz03,C03]
- Define signatures as part of a more complex functionality that provides also other services [Backes-Pfitzmann-Waidner03]

We'll focus on the stand-alone approach (it is more modular).

The deal signature functionality: Attempt 1

1. On input (sid,“KeyGen”) from party (sid,S), register party (sid,S) as the signer.
2. On input (sid,“sign”,m) from (sid,S), record m.
3. On input (sid,“verify”,m) from any party, return (sid,yes/no) according to whether m is recorded.

Too ideal… a realizing protocol would have to deal with communicating the public key and the signatures.

The ideal signature functionality: Attempt 2

1. On input (sid,“KeyGen”) from party (sid,S), register party (sid,S) as the signer, and return to S a “public key” v (chosen at random).
2. On input (sid,“sign”,m) from (sid,S), return a random “signature” s to S, and record (m,s,v).
3. On input (sid,“verify”,m,s,v’) from any party, return (sid,yes/no) according to whether (m,s,v’) is recorded.

Too ideal:

- Public keys and signatures do not have to be random.
- What if m is signed (ie, recorded), but with a different signature than s?
- What if m was never signed but the signer is corrupted?

The ideal signature functionality: Attempt 3

1. On input (sid,“KeyGen”) from party (sid,S), register party (sid,S) as the signer. Forward (sid,S) to A, obtain a “public key” v from A, and output v to (sid,S).
2. On input (sid,“sign”,m) from (sid,S), forward (sid,m) to A, obtain a “signature” s from A, output s to (sid,S), and record (m,s,v).
3. On input (sid,“verify”,m,s,v’) from any party, return (sid,f) where:
   - If (m,s,v’) is recorded then f=1.
   - If S is uncorrupted and (m,s*,v’) is not recorded for any s*, then f=0.
   - Else, forward (m,s,v’) to A, and obtain f from A.

Too weak: Allows a corrupted signer to repudiate signatures, by not recording a signature, and later answering verification queries inconsistently.

The ideal signature functionality: Attempt 4

1. On input (sid,"KeyGen") from party (sid,S), register party (sid,S) as the signer. Forward (sid,S) to A, obtain a "public key" v from A, and output v to (sid,S).
2. On input (sid,"sign",m) from (sid,S), forward (sid,m) to A, obtain a "signature" s from A, output s to (sid,S), and record (m,s,v,1). Verify that no prior record (m,s,0) exists.
3. On input (sid,"verify",m,s,v') from any party, return (sid,f) where:
   - If (m,s,v',b) is recorded then f=b.
   - If S is uncorrupted and (m,s*,v',1) is not recorded for any s*, then f=0.
   - Else, forward (m,s,v') to A, obtain f from A, and record (m,s,v',f).

What if the verifier has the wrong verification key?

The ideal signature functionality: F_{sig}

1. On input (sid,"KeyGen") from party (sid,S), verify that sid=(S,sid'). If not, ignore the input. Else, forward (sid,S) to A, obtain a "public key" v from A, and output v to (sid,S).
2. On input (sid,"sign",m) from (sid,S), where sid=(S,sid'), forward (sid,m) to A, obtain a "signature" s from A, output s to (sid,S), and record (m,s,v,1). Verify that no prior record (m,s,v,0) exists.
3. On input (sid,"verify",m,s,v') from any party, return (sid,f) where:
   - If (m,s,v',b) is recorded then f=b.
   - If S is uncorrupted and (m,s*,v',1) is not recorded for any s*, then f=0.
   - Else, forward (m,s,v') to A, obtain f from A, and record (m,s,v',f).

Note: F_{sig} generates outputs without consulting the adversary. (Indeed, it models Local computation.
Still, the adversary knows each signed message and each signature. This is problematic if we want secret/anonimized signatures.

The privacy-preserving ideal signature functionality: $F_{priv\text{-}sig}$

1. On input (sid,“KeyGen”) from party (sid,S), verify that sid=(S,sid’). If not, ignore the input. Else, forward (sid,S) to A, obtain a “public key” v from A, and output v to (sid,S).
   In addition, obtain from the adversary two programs: a signature generation program SIG and a verification program VER.
2. On input (sid,“sign”,m) from (sid,S), where sid=(S,sid’), let s=SIG(m), output s to (sid,S), and record (m,s,v,1). Verify that no prior record (m,s,v,0) exists.
3. On input (sid,“verify”,m,s,v’) from any party, return (sid,f) where:
   - If (m,s,v’,b) is recorded then f=b.
   - If S is uncorrupted and (m,s*,v’,1) is not recorded for any s*, then f=0.
   - Else, let f=VER(m,s,v’), and record (m,s,v’,f).

Q: Can we realize F_{sig} ?

A: Given a signature scheme H=(GEN,SIG,VER), construct the protocol P^H:

- When invoked with (sid,KeyGen) and pid=S, check that sid=(S,sid'). Then, run (p,v)←GEN(k),return v to the caller, *and keep p.*
- When invoked with (sid,Sign,m), run s←SIG(p,m) and return s. (SIG may maintain state between activations.)
- When invoked with (sid,Verify,m,s,v'),retrurn VER(m,s,v').

Theorem:

A scheme H is existentially unforgeable against chosen message attacks if and only if the protocol P^H securely realizes F_{sig}.

Reminder: Existential unforgeability against CMA

A scheme H=(GEN,SIG,VER) is EU-CMA-secure if:

- Completeness: For all Adversary F, Prob[(p,v)$\leftarrow$GEN(),m$\leftarrow$F(v),VER(m,SIG(p,m),v)=1]~1
- Consistency: For all m,s,v, Var(VER(m,s,v))~0 (This property holds trivially when VER is deterministic.)
- Unforgeability: Prob[(p,v)$\leftarrow$GEN(),(m*,s*)$\leftarrow F^{SIG(p,*)}$(v) s.t. F never asked to sign m*, and VER(m*,s*,v)=1]~0

Proof of equivalence:

P^H realizes F_{sig} → H is EU-CMA-secure:

Completeness: Assume H is not complete, then construct an environment Z and adversary A that distinguish a run of P^H from the ideal process for F_{sig}: Z invokes a simple KeyGen→Sign→Verify sequence for an uncorrupted signer.

Consistency: Assume H is not consistent. Z invokes a KeyGen→Sign sequence for a corrupted signer and verifies the signature several times.

Unforgeability: Assume there exists a forger G for H. Z runs G:

- Z Invokes an uncorrupted S with KeyGen, obtains v, gives to G.
- When G asks to sign m, Z asks S to sign m, obtains s, gives G.
- When G generates (m*,s*), Z asks S to verify (m*,s*,v). Outputs the accept/reject answer.

Analysis: In a run of P^H, Z outputs 1 with non-neglig. probability. In the ideal process, Z never outputs 1.

Proof of equivalence:

H is EU-CMA-secure → P^H realizes F_{sig}:

Let Z be an environment that distinguishes a run of P^H from ideal interaction with F_{sig}, for any ideal-process adversary S. In particular, Z works for the following “generic S”:

- When asked by F_{sig} to generate a key, S runs (p,v)←GEN() and returns v.
- When asked by F_{sig} to generate a signature on message m, S runs s←SIG(p,m) and returns s.
- When asked by F_{sig} to verify (m,s,v’), S runs f←VER(m,s,v’) and returns f.

Claim: Let B be the event that in a run of Z and S in the ideal model, the signer never signed m, and still an (sid,Verify,m,s,v) activation is answered with 1, and. Then, given that event B does not occur, A’s views of the ideal and real executions are statistically close.

Corollary: Since Z distinguishes REAL from IDEAL with non-negl. probability, event B occurs with non-negl. Probability.

Given Z, contruct a forger G for H. G runs Z:

- When Z activates the signer with KeyGen, G gives Z the v from G's input.
- When Z asks the signer to sign m, G asks its oracle to sign m, gets s, and gives Z.
- When Z asks to verify (m,s,v), G checks whether (m,s,v) is a forgery. If so, then it outputs (m,s,v). Else, it continues to run Z.
- When Z asks to corrupt the signer, G aborts.

We are guaranteed that G succeeds with at least the probability of event B.

■

Note: A corollary from the proof is that P^H is adaptively secure iff it is non-daptively secure.

Authenticated communication using F_{sig}

Plan:

- Define a “registry” functionality, F_{reg}.
- Show how can realize F_{auth} in the $(F_{reg,}F_{sig})$-hybrid model:
  - Define a “certification functionality”, F_{cert}, that provides ideal binding between signatures and parties.
  - Show how to realize F_{cert} in the $(F_{dir,}F_{sig})$-hybrid model.
  - Show how to realize F_{auth} in the F_{cert}-hybrid model.
- Authenticating multiple messages with a single key-pair:
  - Define FF_{cert}
  - Realize FF_{cert} using a single copy of F_{cert}.
  - Use the JUC theorem to combine.

Most of this material appears in eprint.iacr.org/2003/139

The "public registry" functionality, F_{reg}

1. When receiving (sid, "Register",v) from party (sid,S), verify that sid=(S,sid'). Then send (sid,S,v) to the adversary, and record (S,v).
2. Upon receiving (sid, "Retrieve",S) from any party, return (sid,S,v) if there is a record (S,v); else return (sid,S,-).

Notes:

• F_{reg} does not "verify knowledge/ability" of any sort. It also does not prevent copying of registered values. Still, it suffices for authentication.

• Each copy of F_{reg} deals only with a single registrant, whose identity is encoded in the sid.

The certification functionality: F_{cert}

1. On input (sid,“sign”,m) from (sid,S), where sid=(S,sid’), forward (sid,m) to A, obtain a “signature” s from A, output s to (sid,S), and record (m,s,1). Verify that no prior record (m,s,0) exists.
2. On input (sid,“verify”,m,s) from any party, return (sid,f) where:
   – If (m,s,b) is recorded then f=b.
   – If S is uncorrupted and (m,s*,1) is not recorded for any s*, then f=0.
   – Else, forward (m,s) to A, obtain f from A, and record (m,s,f).

F_{cert} is similar to F_{sig} except that the KeyGen interface is deleted. Instead, verification is done directly with respect to the signer’s identity (which appears in the sid).

Realizing F_{cert} in the (F_{reg},F_{sig})-hybrid model

Protocol:

- At first activation, signer (sid,S) [verifies that sid=(S,sid'), and] calls F_{sig} with (sid.0, "KeyGen"), obtains v, and calls F_{sig} with (sid.1, "Register",v).
- Whenever activated with input (sid,"sign",m), (sid,S) [verifies that sid=(S,sid'), and] calls F_{sig} with (sid.0, "Sign",m), obtains s, and outputs s.
- Whenever activated with input (sid, "verify", m,s), where sid=(S,sid'), the activated party calls F_{reg} with (sid.1, "retrieve",S), obtains v, calls F_{sig} with (sid.0, "Verify",m,s,v), and outputs the result.

Note: Security is unconditional and simulation is perfect.

Reminder: The authenticated message transmission functionality, F_{auth}

1. Receive input (sid,S,R,m) from party (sid,S). Then:
    1. Output (sid,S,R,m) to party (sid,R)
    2. Send (sid,S,R,m) to S
    3. Halt.

Realizing F_{auth} in the F_{cert}-hybrid model

Protocol:

- When activated with input (sid,S,R,m), party (sid,S) calls F_{cert} with (S.sid, “Sign”, m.R), obtains signature s, and sends (sid,S,m,s) to (sid,R).
- When receiving message (sid,S,m,s), (sid,R) calls F_{cert} with (S.sid, “Verify”, m.R, s). If returned value is 1 then output (sid,S,R,m).

Note: Security is unconditional and simulation is perfect.

Authenticating multiple messages with a single verification key

- So far, we need a different copy of F_{cert} (and thus a different copy of F_{sig} and F_{reg}) for authenticating each message. This is wasteful…
- How to authenticate multiple messages with a single copy of F_{cert} per party?
  - Option 1: Analyze all copies of F_{auth} within a single instance.
  - Option 2: Use the JUC theorem:
    - Define FF_{cert}, the multi-session extension of F_{cert}.
    - Realize FF_{cert} using a single copy of F_{cert}.
    - The JUC theorem says that the composition of multiple copies of a protocol using F_{cert} with a single copy of a protocol that realizes FF_{cert} is secure.

The multi-session certification functionality: FF_{cert}

1. On input (sid,ssid,“sign”,m) from (sid,S), where sid=(S,sid’), forward (sid,ssid,m) to A, obtain a “signature” s from A, output s to (sid,S), and record (ssid,m,s,1). Verify that no prior record (ssid,m,s,0) exists.
2. On input (sid,ssid,“verify”,m,s) from any party, return (sid,ssid,f) where:
   - If (ssid,m,s,b) is recorded then f=b.
   - If S is uncorrupted and (ssid,m,s*,1) is not recorded for any s*, then f=0.
   - Else forward (ssid,m,s) to A, obtain f from A, and record (ssid,m,s,f).

FF_{cert} is identical to F_{cert}, except that it keeps a different record for all the messages signed with each different ssid.

Note: FF_{cert} handles only a single singer.

Realizing FF_{cert} using a single copy of F_{cert}

Idea: Sign the ssid together with the message.

Protocol:

- When activated with input (sid, ssid, “Sign”,m), party (sid,S) [verifies that sid=(S,sid’) and] calls F_{cert} with input (sid, “Sign”, ssid.m), obtains signature s, and outputs s.
- When activated with input (sid, ssid, “Verify”,m,s), party (sid,pid) calls F_{cert} with input (sid, “Verify”, ssid.m,s), obtains a value f, and outputs f.

Note: Security is unconditional and simulation is perfect.