The Real/Ideal Paradigm

How to Enforce Security

- **Cryptography**
  - Randomness
  - Intractable math problems (Quantum may unwind this...)
  - Unpredictable complexity (e.g., Hashing)
    - For hashing, if you can find a collision, it's all over!
    - Not hard in a complexity theory sense, but we can always innovate when one breaks.
- **PL Security**
  - Unforgeable references to heap objects
  - Data abstraction
  - Dynamic control flow
- **Resource Managers**
  - An OS or something that holds/manages resources
  - Referred per-client resource descriptors

How to Define Security

- Employ the Real/Ideal paradigm from Cryptography
- Security means that the Adversary can't tell Real and Ideal systems apart based on inputs/outputs.
  - In the Ideal side, M is embedded in a simulator and can't tell that the system is not actually doing the thing happening in the Real side, where nothing is being simulated.
  - What damage could M do by relating info from the honest parties to the actual functionality?
  - Can you run these side by side and compare? No.

        * Instead, the adversary runs a test on the system, and then returns a judgement. We agree it is secure if the probability that the Adversary returns some probability close to the actual probability.
        * Adversary can experiment individually with one of the worlds, does not get to run them in parallel. Adversary returns T if Real and F if Ideal. Our goal is to make that very hard for the Adversary.
        * If the Adversary cannot tell the worlds apart, then we have security in the Real world because the Ideal world has it by construction. If the Adversary can't tell Real apart from Ideal, then the same security from Ideal must apply to Real.

The EasyCrypt Proof Assistant

- For reasoning about probabilistic imperative programs, including ones involving black-box code.
- Four program logics:
    - Hoare (partial correctness), Prob Hoare (pHL) (Prob procedures terminate with events holding), Prob relational Hoare (pRHL) (Relational reasoning), Ambient Logic (classical higher-order, not dependently typed).
- Proofs with Lemmas using tactics similar to Coq (especially SSReflect).

Cryptographic Security

- Protocols can be specified in EasyCrypt
    - We have to implement the randomness they use. We won't do that in these lectures.
    - There's work on formally connecting EasyCrypt code to low-level implementations of these protocols.
- Example: Symmetric Encryption
    - We will...
        * Define symmetric encryption.
        * Specify security for the scheme using security of PRF.
        * Prove security via reduction to PRF.
    - Do a form of R/I without simulator, but the top-level security theorem will appear to use an indistinguishability game. But R/I underpins it!

- Treatment is parameterized by 3 types:
  - type key. (encryption keys)
  - type text. (plaintexts)
  - type cypher. (cyphertext, scheme specific)
- Scheme is a stateless implementation of a module with a standard interface.
  - proc key_gen() : key
  - proc enc(k: key, x: text)
  - proc dec(k: key, c: cipher)
- Scheme is correct iff running main with arbitrary x returns x with $Pr[1]$. Example on the slides.

An Encryption Oracle Game (IND-CPA Security)

- 4 Procedures, generates a key it stores inside itself (init), then 3 ways of encrypting a plaintext to get a ciphertext. One for the Choose part of the game (encpre), one for the game (enc, generates the ciphertext to decode), and one for the adversary to use (encpost, used in Guess).
- Adversary gets access to the oracle. It can Choose and Guess:
  - Choose returns a pair of plaintexts, can call the encpre procedure, and can call it as much as it likes. But, encpre, after some parameter of calls, starts doing nothing interesting. So the security is dependent on the limit to the number of calls that return something meaningful.
  - Guess = gets a ciphertext, and must choose which of the plaintexts the ciphertext encrypted. Can call encpost as much as it likes, but the defined limit makes it stop returning useful info after some number of calls. Uses the actual key that the game used for its encryption.
- Goal is to prove that the adversary doesn't win more than half the time, or only a tiny bit more than half the time (negligibly more). This is only possible when encryption is probabilistic. Decryption has to be deterministic, of course.
- Can also represent it as the probability that the Adversary loses. Can bound distance between probability that the adversary wins to 1/2. If framed in terms of losses instead, you can just reverse the protocol and get the winning percentage anyway.

Pseudorandom Functions

- Our PRF is an operator F with type key $\rightarrow$ text $\rightarrow$ text.
- Details will be on slides.
- Random function module is a module with init and f (transforms text to text).
- It uses our PRF F to make a random function...? I'm not really sure how that works!
- Game here is for the Adversary to determine whether they are interacting with a PRF vs. a truly Random function.
- We need to limit the Adversary (RFA = Random Function Adversary) somehow, because otherwise it could build a map of inputs to outputs in exponential time. This would distinguish between Random and PRF for sure.

Correctness

- Can prove that the enc'd value always comes out to the right dec'd value value.
- One on the slides uses the XOR thing.

Next Time

- We will pick up from here with some review.
- If we have an Adversarial strategy of our scheme, what does that say about our security theorem?
- If I were an Adversary, how would I make progress in breaking this scheme?
- Give a high-level sketch of our security theorem.

Misc

- <$ is syntax for variable assignment w/ flip.
  - y <$(distribution over type of variable y)
  - Pick a value from this distribution and assign it to y.
- $ is used because of a parsing issue. They had to disambiguate certain things. So they use <- for regular assignment, and <$ for probabilistic assignment. <@ is for assigning the result of a procedure call to a variable.