The Real/Ideal Paradigm
Lecture Notes 03

Private Count Retrieval

- Server, Client, and an untrusted 3rd party. Server has a database, and the client can make queries.
- Database (DB) is one-dimensional, and a query just says how many times an item appears in the DB.
- Goals:
  - Client should not be able to learn anything about the structure of the DB.
  - Server shouldn't learn anything about the queries made by the client, other than the number of queries.
  - Trusted party learns nothing about the queries/client/server but the pattern of queries, if there is one.
- Assume, most of the time, this works properly and we don't have hash collisions. Even with a truly random oracle, there is a possibility of hash collisions.

New Material

- How to formalize this in EasyCrypt.
- Real/Ideal games for the 3 parties.
- Sketch of a proof of security against the three parties.
  - For a given protocol party, Adversary (Adv) gets to see what happens inside the execution of that party, not change how they behave, but maybe learn "more than they should."

Elements, Secrets, and Hashing

- See slides for precise definitions of Elements, Secrets, Hash tags, and hashing.
- Adversary can access the random oracle, can hash using the same oracle as the parties, but it can't look at the hash map. It only gets to query the oracle.

PCR Protocol

- DB is a list of elements, hdb is a list of hash tags.
- "Views" encode everything that happens in the game.
- Environment (Env) has 4 procedures asking for a DB, but is also allowed to end early by rejecting a query(?).
- Protocol is parameterized by an Env, and the random oracle. Main initializes the random oracle, then runs the protocol properly. Again, see slides to see the exact way it runs the game.
  - Runs the `client_loop()` function(?)
- Boolean to represent the Adversary's ending judgement.

The Adversarial Model

- Remember, we are modeling semi-honest or honest-but-curious security.
- Adversary gets to have the view of one of the parties, but can't change their behavior.
- Adversary gets access to the random oracle, but cannot view inside it.
- Games are parameterized by the Adversary, where the Adversary is trying to return a Boolean judgement to distinguish whether it is looking at the Real vs. Ideal world.

Real Games

- The Protocol was defined (partially) on prior slides.
  - Takes Adversary and gets an Environment made out of the Adversary.
- Defines 4 procedures: `init_and_get_db`, `get_qry`, `qry_done` (so Adversary gets the view), and `final` (Adversary's boolean judgement).
  - All can call the random oracle.
  - Adversary can do hashing when deciding which DB and query to use.
  - Adversary is trying to increase its knowledge through queries, though again, it cannot change the behavior of any parties involved.

Ideal Game

- Parameterized by a Simulator, in addition to the other inputs.
  - It is trying to convince the Adversary that it is looking at the Real game, instead.
  - When examining leakage to the Adversary via the simulator, we don't need to worry about computation power, as exponential work won't actually tell the Adversary anything via the leakage.

2D Sequence of Games

- When doing the sequence of games, we reduce a sequence of games to one game (top level to bottom).

Real Game for Server

- What can the Server learn about the queries and their counts when the protocol happens?
  - Knows the DB.
  - Queries are paired with secrets and being hashed.
  - Nothing goes directly to the server, so the only thing it can learn is that the random oracle's state is changing at every step. Since the Adversary can do hashing, they might notice that something changed about the random oracle between calls.
    - So, the Adversary can learn the number of calls by examining the state of the random oracle.
- So we need to formalize the Ideal Game for the server.

Ideal Game for Server

- See slides for a cool diagram!
- Basically, the simulator makes up the middle parts between the DB and the client. The client does no real hashing, and the simulator just pretends that there is a client, third party, etc., making calls and hashing things.
  - Adversary cannot learn anything about the queries because the queries never interact with the simulator. Similarly, they don't interact with the random oracle.
- How do we get back to the real world?
  - Since we are actually hashing query/secret pairs, how can we relate this step to get rid of hashing to the real world?

    * Every time the client loop runs, the Adversary is "woken up," and so it can at least learn the number of queries no matter what. But the view from the simulator isn't giving them any more information.

Proof of Security Against the Server

- Real/Ideal games should be equally likely to return True.
- Redundant hashing info is on the slides.
  - Basically, `OptHashing` lets you not hash things even when asked to if the hash isn't actually needed.
  - Can the Adversary tell the difference between the optimized vs. non-optimized versions? No, apparently!
    * You can put off the redundant hashing until the protocol is over, or until the thing is actually being hashed. So the Adversary can't learn anything depending on whether you hash it or not.

Ideal Game for Third Party

- No secret used, they are just hashing things using their own private random oracles.
  - They're using private true random functions that the Adversary has no access to.
- Adversary can see patterns, but it has no way to take a hash tag and learn anything about the original input, assuming no hash collisions.
- Adversary is invoked with the Third Party's view when the DB and queries are requested by the game.
- Ideally, Adversary learns only patterns, but nothing about the contents of the DB or the queries.
  - Adversary learns these things via the view that the third party is simulating, with no access to the client/server private true random oracle.
  - Adversary learns nothing about the order of the DB, either.

Security Against Third Party

- To tell the games apart, Adversary has to guess the secret, e.g., call hash function and hash it with the secret.
- Adversary potential strategies can be viewed on the slides.

  - Basically, it takes a candidate secret and tries to hash everything it has seen from client/server and see if it matches.
    * If in the Real world, this would let it find the secret if it has unbounded computational power. In the Ideal world, Adversary would not be able to find the secret since the client/server are using a different, private oracle.
      · So, we need to bound the computational power of the Adversary in order to get a meaningful security theorem.

Third Party Proof

- We can introduce a new abstraction, "secrecy random oracles."
  - Offers limited hashing to element/secret pairs, and unlimited hashing of elements.
  - Puts limits on the brute force attack described prior (where the Adversary just guesses all possible secrets).
- Adversary should only be able to tell the games apart if it has the secret.
  - Proven, again, with "up to bad" reasoning. As long as the Adversary doesn't randomly guess the secret very early in the computation, this is safe. So it's bounded by the probability that the Adversary guesses the right secret (usually extremely unlikely).
- "Secret Guessing Oracle" is a new abstraction that gives Adversary a limited number of guesses at the secret. The probability of the Adversary winning is bounded by the number of possible secrets (which is exponential).
  - Secrecy Random and Secret Guessing Oracles are reusable.

Client Proof

- See slides for diagrams for the Real and Ideal games for the Client.
- (In the Ideal game) The DB coming from the Adversary in the Environment is being given "the simulator's interface to the game" (SIG).
  - Lets us see the map of things + number of times those things appear in the DB.
  - SIG can be limited to a fixed # of responses. Adversary just learns the number of queries.
- Adversary can try to force a hash collision. If there's a pair of elements paired with the secret (which the Adversary knows because the Client knows the secret, so it's in the view) hash to the same tag, then it can win.

  - Creating a hash collision in the Real game gives 1, and in the Ideal game gives 0, so it can use this to distinguish between the Real/Ideal worlds.

- Adversary can also propose a DB of distinct elements with more elements than hashtags. This would guarantee a hash collision, and so the Adversary could always win with a sufficiently large number of queries.
  - Therefore, we need to limit the size of the DB that the Adversary can provide.
    * Info about how to enforce a hashing budget is in the slides. It also uses up-to-bad reasoning, in case of a random collision with a very low probability or non-termination.

- Concept of a Budgeted Random Oracle lets us limit the number of hashes to avoid brute forcing a collision.
  - Can have a collision-possible or a collision-free-within-budget version.
    * Collision-possible can have random collisions. Collision-free-within-budget guarantees that you will never have a collision as long as you are within your 'budget'.
      · Transferring between the two oracles comes with a slight probability increase of a collision.

- I'm not sure when or why you switch between the oracles in this proof. :(
  - The slides contain just a sketch of a proof, sadly.
  - Apparently, we paid the penalty for switching oracles twice in this proof.

Summary/Lessons Learned

- The size of the EasyCrypt formalization is relatively short.
- The proof is long, but if we trust EasyCrypt to check that, then this is fine. :)
- The budgeted random oracles let us sidestep the issue of hash collisions.
- Some of the slides went by too fast! But you can view them to see some more of the things we have learned. :)

Prep for the Next Lecture

- Battleship!
  - We can construct a Real/Ideal paradigm using it, and audit our model of Battleship using proofs of security.
  - Explanation of Battleship rules and basic strategy are on the slides.

Program Architecture and Behavior

- See the slides here for a good diagram as well!
- Used Concurrent ML to implement referee and board.
  - Changed from the referee model to 2 mutually distrustful player interfaces instead. Both trust the same infrastructure, but hopefully requires less trust than an actual referee.
    - We rely on PL security for this enforcement, data abstraction.
- What about error behavior of the trusted model?
  - We'll see how to handle that next time.