Lecture Notes 04

Review

- We are implementing secure Battleship in Haskell/LIO.
- We will rely on secure architecture. A trusted referee has the 2 players' boards, takes the initial boards, and runs the game of Battleship according to its rules.
  - Referee implemented in ConcurrentML.
  - Can we split the referee into mutually distrustful player interfaces?
    * Using a protocol to run the game instead, the players should see the same thing.
    * If a player deviates from the protocol, how can we define from the other interface that they are still secure? **(See slides for a nice diagram!)**

Real/Ideal Security Framing

- The real/ideal security framing gives us a means of reasoning about and enforcing a fair game of Battleship.
- How do we define the ideal world?
  - Using the model referee is a good start. If it is enforcing the rules of the game as a trusted party, it coincides with the best case scenario for the two-player interface version. If the adversary can't tell the difference between that and a simulator running the game and pretending to be a referee, that should give us our proof of security (i.e., if you can't distinguish between the real/ideal versions of the game, then you can treat the real version as the ideal version).
    * We want the outcomes of execution to be the same on the Real/Ideal side.
    * **... Is this what we want?**
    * If M violates the protocol, then the simulator terminates, but the model referee may have learned enough to output to the adversary. So the adversary could learn which is the simulation,

potentially, by trying to violate the protocol and determining whether the protocol communicates an error, or a result and *then* an error.

  - **See the paper from PLAS 2014 (Uppsala!)**

Battleship Game Overview

- See slides for a brief overview of the game Battleship, plus the finer details of strategy, and what information the players get when landing shots.

LIO

- Thanks, "Steve Security!"
- A library for Concurrent Haskell with dynamic enforcement of information flow control.
- Gives mutable variables shareable between predicates and used for communication.

LIO Battleship

- Player Interfaces exchange, using trusted code, labeled boards using labeled cells.
- **See slides for the code which implements this.**
- See slides for an LIO example! The diagram is very helpful. :)
- The "1 AND 2" bit on the right side is saying that both 1 and 2 trust the codebase used to generate the initial state of the game.
- 1 on the left refers to secrecy. So, "1: ..." says that Player Interface 1 is allowed to declassify the cell.
  - While Player Interface 2 could choose not to label it, the PC label would be raised and have 1 with it. The channels it communicates on won't allow that. So if 2 declassifies one of these cells, it won't be allowed to communicate further.
- Each ship has a mutable variable associated with it. When pb is run for either, they can reference, read, and write to this mutable variable.
- If we didn't do that, it would be possible for player 2 to collect the unclassified cells, run the DC actions in a different order. It could then ask player 1 to declassify HC, then GC, etc. It could then send them to player 1, but run the actions in a different order. This would give player 2 more

information than they are entitled to, as they could learn that they sunk the patrol boat before they finished hitting all of it.

- Player 1, when getting the cell, has permission to declassify it, and trust is because it was generated using the mutually trusted codebase.

Concurrent ML

- Library for Standard ML
- No special security features (neither does concurrent Haskell)
- From abstract types and mutable references, you can program access control and such.

Concurrent ML + AC Battleship

- Player Interfaces implemented using a different mechanism in ML.
  - Exchange using trusted code, boards are immutable and abstract and locked, and can be unlocked using unforgeable keys.
  - **... But there's a catch!** We have a type of keys, a type of counted keys, but there's no way of deconstructing a counted key. If the key is unforgeable and an int, you can put them together to make a counted key, but you can't get just the key back from that point. Type of key and counted key are abstract.
    - (Secret monad!? The counted key sounds a little monadic...)
- **See slides for an example game using this construction!**
- Basically, since you can't destruct the components of the protocol, you can't do the same attack as the LIO example.
  - The counters are what keep you from performing the prior exploit. It enforces the order of calls in the protocol.
- **See slides for a diagram about M's possible knowledge when run against the Simulator.**
- At the start, the model referee hasn't said anything about the opponent's board's state. It knows that information, but there is no leakage of information. The board is locked and immutable, though, so we have to have it point to a mutable variable (remember that from before) in order to update its state.

- Meanwhile, the model referee wants to shoot, and M chooses to shoot at a location. It sends it to "the other player interface," and the model interface updates the board corresponding to M's query, and says that M got a hit. The supervisor patches "hit" into that mutable variable from before, simulating an updated board. The Supervisor sends a counted key that unlocks the proper position, then M can run the "shoot" procedure again, and consults the mutable variable to choose what to return ("hit").

What Happens Without the Counters?

- Without the counters, or if M could destruct them, it could give GC a counter "1" and shoot lb1. This would let the attack from earlier, where you learn where the patrol boat is a bit early, happen again in another form. This would fail to replicate the things that happen in the real world.

M Commits to a Board

- What if there is a bug in the protocol, and M moves its ships around to avoid G's shots?
  - How do we know that this can't happen?
  - Ideal world: The referee gets both players to send their boards.
  - So, in the Real world, the Supervisor is being asked what the boards are at the start of the game.
  - Supervisor has an API call and can check which board was used to create the current board. So, M has to give a real board to the 'trusted infrastructure,' and the Supervisor constructs the corresponding prior board to check it... I think.
- If there were a bug like this in the protocol, we could distinguish between the boards.
- What might frustrate the Supervisor in this case?
  - If S has given M a locked empty board first, then M sends back the same board (a replay attack). Then, there would be no way of doing an extraction to check how the board was derived! (There is no valid prior board state, and so the check can't compute a valid prior board.)
  - The protocol has to protect against this attack, or else risk getting stuck in a locked state (which would distinguish between the Real/Ideal world, **possibly** as the Real world could get stuck and the Ideal world would simply reject the board).

Summary

- Separating the definition of security from its enforcement, she claims, is huge!
  - "How do we know which of M's actions are okay?" EasyCrypt can help with that!
- Lots of bugs were found, of course... but it shows why formalization is useful! It helped to find them.
- Sandboxing?
  - Safe Haskell used in LIO stops bad behaviors in M. But M might be able to stop the process by introducing an error. So, we would want to check that M isn't blowing up and is only interacting via accepted channels for the system.
  - For Concurrent ML, this had to be done by just reading the code. But you could make a type system do this! It would probably just require some type system engineering.

Discussion

- How do we know that a Real/Ideal paradigm says what we want?
  - It's VERY easy to get it wrong, as Cryptography definitions are NOTORIOUSLY precarious. Small changes in protocol can eliminate ALL of our security guarantees. **See the slides for a diagram outlining one such case...**
- What alternatives to Real/Ideal do we have, in this setting or others?
  - It's hard to get right... it would be awesome to describe what we wanted with, like, a Hoare logic or something. Can PL people help with this?
- When is splitting a trusted component into two mutually distrustful components helpful?
- For Battleship, can we rely on smaller trusted computing bases?
  - The referee was simple, and we substituted a really hard thing to simulate it. Is there something simpler we could do instead, to get the same benefit?
- Why didn't we use information flow control?
  - Originally were trying to evaluate LIO, and thought Battleship was a cool classic thing to implement! But it turned out to not require information flow control. In the end, everything is declassified.

  - Could we reason about when information flow control is or isn't needed?

Future Work

- We want to prove security in a proof assistant!
  - People are working on this right now! They aren't there yet, though.
  - The proofs are very, very hard to do, and hard to mechanize. Need something like Iris, but with types...
  - Friendly competition here is welcome!