The Real/Ideal Paradigm

Lecture 4

Alley Stoughton

Boston University

Oregon Programming Languages Summer School
June 3–13, 2024
Boston University
We’ll start this last lecture with a review of:

- the program architecture of our secure battleship implementations in Haskell/LIO and Concurrent ML
- our Real/Ideal Paradigm definition of security against a malicious player interface
- Then we’ll survey the two implementations and consider how we used our security definition to audit them
Referee holds and updates both Players' boards

First Client to connect to Server gets to shoot first
Trusted Referee

- We implemented in Concurrent ML a trusted referee that holds and updates both player’s boards, enforcing the rules of the game.
- But we were also interested in reducing the trusted computing base (TCB), by splitting the referee into mutually distrustful player interfaces.
Splitting Referee into Mutually Distrustful Player Interfaces (PIs)
Splitting Referee into Mutually Distrustful Player Interfaces (PIs)

PIs will rely on some trusted infrastructure.

How do we define security against a malicious opponent PI?
Security Against Malicious PI (Tentative)

boolean judgment

security: if there is an execution on one side resulting in $b$, then there is an execution on the other side also resulting in $b$
Security Against Malicious PI (Tentative)

security: unfortunately, if M doesn’t follow the protocol, the error behavior (termination) in the two worlds can be different.
Security Against Malicious PI

security: instead, we propagate errors, and model referee only yields a non-erroneous result if simulator player says OK
On GitHub
https://github.com/alleystoughton/battleship
you can find a link to our PLAS 2014 paper *You Sank My Battleship!: A Case Study in Secure Programming* plus the Haskell/LIO and Concurrent ML code

- Note that the error propagation presented above is *not* followed by this code or described in the paper
## Ambiguity Example: Patrol Boat

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>p</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>p</td>
<td></td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- GD
- HC
- GC
**Ambiguity Example: Patrol Boat**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>p</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>s</td>
<td></td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>s</td>
<td></td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>s</td>
<td></td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LIO

• LIO is a library for Concurrent Haskell with dynamic enforcement of information flow control
• Information flow labels have both secrecy and integrity components
• Provides mutable variables, which can be shared between threads, and used for communication
LIO Battleship

- PIs exchange — using trusted code — labeled boards, made of labeled cells:

```plaintext
data LSR = -- labeled shot result
  Miss -- a miss
  | Hit -- hit an unspecified ship
  | Sank Ship -- sank a specified ship

data LC = -- labeled cell
  LC
  (DCLabeled
   (Principal, -- originating player interface
    Principal, -- receiving player interface
    Pos, -- position of cell
    DC LSR -- DC action for shooting cell
  ))
```
LIO Example

PI 1

Patrol Boat
MVar

PI 2

MVar
LIO Example

PI 1

Patrol Boat MVar

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2
LIO Example

PI 1

1 : (1, 2, HC, pb) : 1 ∧ 2

Patrol Boat
MVar

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2
LIO Example

PI 1

: (1, 2, HC, pb) : 1 \land 2

PI 2

1 : (1, 2, GC, pb) : 1 \land 2

1 : (1, 2, HC, pb) : 1 \land 2

Patrol Boat MVar
LIO Example

**PI 1**

: (1, 2, HC, pb) : 1 \land 2

**PI 2**

1 : (1, 2, GC, pb) : 1 \land 2

1 : (1, 2, HC, pb) : 1 \land 2

Patrol Boat MVar

HC
LIO Example

PI 1

: (1, 2, HC, pb) : 1 \land 2

PI 2

1 : (1, 2, GC, pb) : 1 \land 2

1 : (1, 2, HC, pb) : 1 \land 2

: (1, 2, HC, pb) : 1 \land 2

Patrol Boat MVar

HC
LIO Example

PI 1

: (1, 2, HC, pb) : 1 \land 2

PI 2

1 : (1, 2, GC, pb) : 1 \land 2

1 : (1, 2, HC, pb) : 1 \land 2

Yields Hit

: (1, 2, HC, pb) : 1 \land 2

Patrol Boat MVar

HC
LIO Example

\( \begin{align*}
\text{PI 1} & : (1, 2, \text{HC, pb}) : 1 \land 2 \\
1 : (1, 2, \text{GC, pb}) : 1 \land 2
\end{align*} \)

\( \begin{align*}
\text{PI 2} & : (1, 2, \text{HC, pb}) : 1 \land 2 \\
1 : (1, 2, \text{GC, pb}) : 1 \land 2
\end{align*} \)

Yields Hit

Patrol Boat

MVar

HC
LIO Example

PI 1

: (1, 2, HC, pb) : 1 \land 2

: (1, 2, GC, pb) : 1 \land 2

Yields Hit

PI 2

1 : (1, 2, GC, pb) : 1 \land 2

1 : (1, 2, HC, pb) : 1 \land 2

1 : (1, 2, HC, pb) : 1 \land 2

Patrol Boat

MVar

HC
LIO Example

PI 1

: (1, 2, HC, pb) : 1 \land 2

: (1, 2, GC, pb) : 1 \land 2

Yields Hit

PI 2

1 : (1, 2, GC, pb) : 1 \land 2

1 : (1, 2, HC, pb) : 1 \land 2

Yields Hit

: (1, 2, HC, pb) : 1 \land 2

Patrol Boat

MVar

GC, HC
LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, GC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

Yields Hit

: (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, GC, pb) : 1 ∧ 2

Patrol Boat MVvar

GC, HC
LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∨ 2

: (1, 2, GC, pb) : 1 ∨ 2

Yields Hit

Yields Sank PatrolBoat

PI 2

1 : (1, 2, GC, pb) : 1 ∨ 2

1 : (1, 2, HC, pb) : 1 ∨ 2

: (1, 2, HC, pb) : 1 ∨ 2

Patrol Boat

MVar

GC, HC

Yields Hit

Yields Sank PatrolBoat
LIO Example

PI 1

: (1, 2, HC, pb) : 1 \land 2

: (1, 2, GC, pb) : 1 \land 2

Still Yields Hit

Yields Sank PatrolBoat

Patrol Boat MVar

PI 2

1 : (1, 2, GC, pb) : 1 \land 2

1 : (1, 2, HC, pb) : 1 \land 2

: (1, 2, HC, pb) : 1 \land 2

: (1, 2, GC, pb) : 1 \land 2

GC, HC
Concurrent ML

• Concurrent ML is a library for Standard ML (we use the Standard ML of New Jersey implementation)
• It has no special security features
• But the combination of its abstract types (provided by its rich module system) and mutable references can be used to program access control
CML + AC Battleship

- PIs exchange — using **trusted** code — **immutable, abstract locked boards**, whose cells can be unlocked using **unforgeable keys** held by originating player:

```ml
type key (* key *)
type ck (* counted key *)
val labelKey : key * int -> ck
type lb (* locked board *)
datatype lsr =
    | Invalid (* invalid counted key *)
    | Repeat (* illegal repetition *)
    | Miss (* missed a ship *)
    | Hit (* hit an unspecified ship *)
    | Sank of ship (* sank the given ship *)
val lockedShoot : lb * pos * ck -> lb * lsr
```
CML + AC Example

PI 1

PI 2

lb₁
CML + AC Example

PI 1

PI 2

$Ib_1$

HC
CML + AC Example

PI 1

HC

PI 2

Ib₁

HC
CML + AC Example

PI 1

HC

PI 2

Ib₁

HC

(key₁,
₁)
CML + AC Example

PI 1

HC

Hit

PI 2

lb₁

HC

(key_{HC}, 1)

lb₂
CML + AC Example

PI 1

HC

Hit

PI 2

Ib₁

HC

(key_{HC, 1})

Ib₂

GC
CML + AC Example

PI 1

HC
GC

PI 2

Ib₁
HC
(key_{HC, 1})

Hit
Ib₂
GC
CML + AC Example

PI 1

HC

GC

PI 2

$\text{lb}_1$

$\text{HC}$

$(\text{key}_{\text{HC}}, 1)$

Hit

$\text{lb}_2$

$\text{GC}$

$(\text{key}_{\text{GC}}, 2)$
A counted key is only applicable to a single locked board, and can’t be deconstructed.
Construction of Simulator Player for CML + AC

Adversary

boolean judgment

simulator player

Model Referee

G
M
Referee

S(M)

Adversary
Construction of Simulator Player for CML + AC

Referee

supervisor interacts with M using *reimplementation* of locked board abstract type

Model Referee

supervisor
CML + AC: M Doesn’t Learn More Than it Should

Referee

Model Referee

G \leftrightarrow M

M

S
CML + AC Simulator Example

Model
Referee

S(M)
Supervisor

M

lb₁

GC
HC
CML + AC Simulator Example

\[ S(M) \]

Supervisor

Model
Referee

\[ M \]

\[ lb_1 \]

GC

HC
CML + AC Simulator Example

Supervisor

\[ S(M) \]

Referee

\[ M \]

HC

ib_1

HC

GC

HC
CML + AC Simulator Example

Model
Referee

HC

HC

GC

HC

S(M)

Supervisor

M

lb₁

HC
CML + AC Simulator Example

Supervisor

$S(M)$

Model
Referee

$M$

$\text{HC}$

$\text{Hit}$

$\text{GC}$

$\text{HC}$

$\text{HC}$

$\text{HC}$
CML + AC Simulator Example

Supervisor

$S(M)$

$M$

Model Referee

$HC$

$Hit$

$GC$

$HC$

$Hit$

$lb_1$

$HC$
CML + AC Simulator Example

Supervisor

$S(M)$

$M$

Model
Referee

HC

Hit

GC

HC

Hit

(HC, 1)

lb$_1$

lb$_2$
CML + AC Simulator Example

Supervisor

\( S(M) \)

\begin{align*}
\text{Model} & \\
\text{Referee} & \\
\text{HC} & \\
\text{Hit} & \\
\text{HC} & \\
\text{Hit} & \\
\text{HC} & \\
\text{Hit} & \\
\text{GC} & \\
\text{HC} & \\
\text{Hit} & \\
\text{GC} & \\
\text{HC} & \\
\text{Hit} & \\
\end{align*}

\( M \)

\( (\text{key}_{\text{HC}}, 1) \)
CML + AC Simulator Example

\[ S(M) \]

**Supervisor**

- HC
- Hit
- GC

**M**

- \( \text{lb}_1 \)
- HC
- \( \text{key}_{\text{HC}, 1} \)
- \( \text{lb}_2 \)
- GC

**Model**

- HC

**Referee**

- GC
- Hit
CML + AC Simulator Example

Model Referee

Supervisor

$S(M)$

$M$

Hit

$S(M)$

$M$

Hit

$S(M)$

$M$

Hit

Sank PatrolBoat

Model Referee

Supervisor

Hit

$S(M)$

$M$

Hit

Model Referee

Supervisor

Hit

$S(M)$

$M$

Hit

Model Referee

Supervisor

Hit

$S(M)$

$M$

Hit

Model Referee

Supervisor

Hit

$S(M)$

$M$

Hit

Model Referee

Supervisor

Hit

$S(M)$

$M$

Hit

Model Referee

Supervisor

Hit

$S(M)$

$M$

Hit

Model Referee

Supervisor

Hit

$S(M)$

$M$

Hit

Model Referee

Supervisor

Hit

$S(M)$

$M$

Hit
CML + AC Simulator Example

Model Referee

Model
Referee

Supervisor

$S(M)$

$M$

HC

Hit

GC

HC

Hit

GC

HC

Hit

GC

Sank PatrolBoat

Sank PatrolBoat

Sank PatrolBoat

(key$_{HC}$, 1)
CML + AC Simulator Example

Model Referee

Supervisor

\( S(M) \)

\( M \)

\( HC \)

\( Hit \)

\( GC \)

\( Sank PatrolBoat \)

\( lb_1 \)

\( HC \)

\( key_{HC, 1} \)

\( lb_2 \)

\( GC \)

\( key_{GC, 2} \)

\( Hit \)

\( Sank PatrolBoat \)

\( HC \)

\( Hit \)
CML + AC Simulator Example

\[
S(M) = \begin{cases} 
HC & \text{Hit} \\
GC & \text{Sank PatrolBoat} \\
\end{cases}
\]

\[
M = \begin{cases} 
lb_1 & \text{Hit} \\
lb_2 & \text{GC} \\
lb_3 & \text{Hit} \\
\end{cases}
\]

Supervisor

Model

Referee

\begin{align*}
\text{Hit} & \text{(key}_{HC}, 1) \\
\text{Hit} & \text{(key}_{GC}, 2) \\
\end{align*}

HC

GC

HC

GC

HC

GC

HC

GC

HC

GC

HC

GC

HC

GC

HC

GC

HC

GC

HC

GC

HC
abstract type has two kinds of locked boards: one for **shooting** and one for **extraction**; **S** extracts board from locked board **M** initially provides

Q: What is the potential pitfall with this approach?
CML + AC: M Commits to a Board

abstract type has two kinds of locked boards: one for shooting and one for extraction; S extracts from the locked board M provides its source board, to give to G

A: A replay attack in which M gives G back its own locked board must be prevented
Summary

• We used theoretical cryptography’s real/ideal paradigm to define when one program interface is secure against a possibly malicious program interface
  • This separates the definition of security from its enforcement
  • We gave two secure implementations, using our definition to guide our design and informally audit it
  • Using LIO and information flow control
  • Using Concurrent ML + access control
  • We found numerous security bugs during our audits
Summary

- Safe Haskell mostly automates the check that the malicious player interface only communicates via its channels.
- But we also want to check that it doesn’t do an `exit` (terminating the whole program) — and this may have to be checked manually.
- In Concurrent ML, it must be manually checked that the malicious PI only communicates via its channels.
Research Questions

• How do we know that a real/ideal paradigm definition says what we want?
• Designing ideal functionalities is something of an art, and tools for making their design easier would be useful
• Tools for helping the designer know they got the correct definition would also be helpful
How Do We Know This Is What We Want?

Suppose we forgot to include opponent’s shots

G | M
---|---
Referee

M could learn more than it should in real world, and S(M) could simulate this by making different shots

Simulator Player

S(M)
Research Questions

• What are alternatives to the real/ideal paradigm for defining the security of one component against another?

• When is it useful to split a trusted component into two mutually distrustful ones?

• For Battleship, are there solutions relying on smaller trusted computing bases?

• When is information flow control necessary to achieve security?

• Why did Battleship not require information flow control?
Future Work

• We want to prove security using a proof assistant
• It must be possible to formalize and reason about a programming language with
  • A rich module system, supporting abstract types
  • Concurrency
  • Mutable references
• We need to be able to reason about thread scheduling
• We are currently investigating whether the Coq development of the concurrent separation logic Iris would be a good vehicle for this work
• Joint work with Jared Pincus, Arthur Azevedo de Amorim and Marco Gaboardi
Questions about Example 3?
Real/Ideal Paradigm Summary and Discussion

• Let’s end these lectures with an open discussion about the real/ideal paradigm
• Possible discussion points:
  • Difficulty defining ideal functionalities capturing correct security notions
  • Approaches to proving security in the real/ideal paradigm
  • Applicability to non-cryptographic security
  • Possible alternative approaches