You Sank My Battleship! *

A Case Study in Secure Programming

Alley Stoughton
Andrew Johnson
MIT Lincoln Laboratory
{alley.stoughton,ajohnson}
@ll.mit.edu

Samuel Beller
Brandeis University
sammbeller@gmail.com

Karishma Chadha
Wellesley College
kchadha@wellesley.edu

Dennis Chen
Tufts University
dchen741@gmail.com

Kenneth Foner
Brandeis University
kenneth.foner@gmail.com

Michael Zhivich
MIT Lincoln Laboratory
mzhivich@ll.mit.edu

Abstract

We report on a case study in secure programming, focusing on the design, implementation and auditing of programs for playing the board game Battleship. We begin by precisely defining the security of Battleship programs, borrowing ideas from theoretical cryptography. We then consider three implementations of Battleship: one in Concurrent ML featuring a trusted referee; one in Haskell/LIO using information flow control to avoid needing a trusted referee; and one in Concurrent ML using access control to avoid needing such a referee. All three implementations employ data abstraction in key ways.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; Modules, packages; D.4.6 [*Operating Systems*]: Security and Protection—Access controls; Information flow controls

Keywords security, auditing, access control, information flow control, data abstraction, concurrent functional programming, real/ideal paradigm

* This work was sponsored by DARPA under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the Department of Defense or the United States Government.

PLAS'14, July 29, Uppsala, Sweden.

http://dx.doi.org/10.1145/2637113.2637115

1. Introduction

Programming languages for information flow control (IFC) (Sabelfeld and Myers 2006) promise to make secure programming easier. IFC languages include: Jif (Myers 1999; Arden et al. 2013), a security-typed programming language extending Java with support for IFC, enforced at compile and run time; Fabric (Liu et al. 2009), a distributed programming language based on Jif; LIO (Stefan et al. 2011), a library for Safe Haskell (Terei et al. 2012) providing dynamically enforced IFC; and Breeze (Hriţcu et al. 2013), a dynamically typed functional language with dynamic IFC. IFC languages restrict the flow of data, preventing more-classified data from influencing less-classified results, and preventing lower-integrity data from influencing higher-integrity results.

IFC facilitates partitioning programs into *trusted* and *untrusted* components in such a way that a program's security is independent of the behavior of its untrusted parts. This makes auditing easier, as the auditor can focus on the trusted components. One can also consider mutually distrustful components, perhaps implemented by different software development teams. A given team may want to assure itself that its component is *secure against* the other components, i.e., that even if the other components were replaced by sloppy or malicious code, the security of their component wouldn't be compromised.

Although recent research on secure programming has focused on IFC, other approaches are available, including the use of data abstraction and access control (AC) (Lampson 1971). Data abstraction can be used to maintain data invariants and provide limited views and access to data. It is often useful to supplement other approaches with data abstraction. AC can be used to restrict data access to program parts hold-

ing the right privileges, without limiting how data may be used once accessed.

There is a rich technical literature on language-based security. E.g., (Sabelfeld and Myers 2006) includes 147 references to papers on information flow control. Given all this work, it is surprising how little attention has been paid to the problem of specifying program security policies. With complex programs involving declassification and endorsement, the state-of-the-art involves attempting to capture an informal policy with dozens or hundreds of Jif security annotations. To quote Steve Zdancewic (Zdancewic 2004),

> . . . we do not yet have the tools to easily describe desired security policies. We do not understand the right high-level abstractions for specifying information-flow policies.

These words were written ten years ago, but we believe they could have been written today.

Case studies can help us understand the security-related benefits provided by IFC, AC and data abstraction, as well as the effort required to achieve these benefits in a practical setting. To date, relatively few case studies have been carried out using language-based security. These include: Jif implementations of Battleship (Zheng et al. 2003) and online poker (Askarov and Sabelfeld 2005); the implementation of λChair, an API for implementing secure conference reviewing (Stefan et al. 2011); and GitStar, a code-hosting website enforcing privacy in the presence of untrusted apps, which is built using Hails, an IFC web framework based on LIO (Giffin et al. 2012).

For our case study, we wanted an application that was a good test bed for developing a more formal approach to the specification of whole-program security. Our plan was to implement the application using different language-level enforcement techniques, providing a common ground on which to compare these techniques. For realism, we wanted an application requiring concurrent programming. The application we selected was the two-player board game Battleship, whose informal security policy has just the right level of complexity.

In Battleship, each player begins by placing its ships on a private board. Subsequently, the players take turns shooting cells of each other's boards. After the opponent makes a shot on a player's board, the player announces whether it was a "miss" (the shot cell was vacant) or a "hit" (the shot cell was part of an unspecified ship). When the last cell of a ship is hit, the player announces that this ship has been "sunk" (e.g., "you sank my battleship").[1] The first player to sink all its opponent's ships is declared the winner.

A simplified version of Battleship was implemented by the Jif team (Zheng et al. 2003; Arden et al. 2013).[2] This program isn't interactive; the players' strategies are hard-coded. Furthermore, a player isn't informed when it has sunk a ship, but only whether a shot is a hit or a miss. The first player to hit all its opponent's cells is declared the winner.

In Jif, Java types are augmented with information flow labels from the decentralized label model (DLM) (Myers and Liskov 1997). The DLM label of a value has two parts: a *secrecy* part, saying who may read the value; and an *integrity* part, saying who might have written the value. Jif provides mechanisms for declassifying data—affecting secrecy—and endorsing data—affecting integrity. In the Jif Battleship program, a player's board has the secrecy of the player, but is endorsed by both the player and its opponent. Most of the program's methods have the integrity of both players, meaning they must be trusted by both players. E.g., this is true of the method used by a player to shoot (declassify) one of its cells on behalf of the opponent; consequently, the method must be trusted (audited) by the opponent. Because the program has a single `Player` class, parameterized by the player and opponent, both players must trust the methods of this class. Unsurprisingly, it is easy to create a well-typed `BadPlayer` class that lies about the result of shooting a cell.

The reader may now be wondering what—exactly—it means for a Battleship implementation to be "secure". Informally, it seems our security policy should say that:

(1) a player must properly place its ships before game play begins, and may not move those ships during the game; and

(2) at each stage of the game, a player's knowledge about its opponent's board must correspond exactly to what was revealed through faithfully executed game play.

But this policy description isn't formal enough to be the basis for program auditing, much less for a proof of program security. Questions include: What threads or modules are the players? What data structures are the boards? What constitutes game play? We will answer these questions by giving precise definitions of program security, borrowing ideas from theoretical cryptography (Canetti 2000).

Our case study in secure programming features three Battleship implementations:

- One in Concurrent ML (CML) (Reppy 1999) using a trusted referee. (About 480 trusted lines of code, 500 untrusted LOC.)
- One in Haskell/LIO using IFC to avoid the need for a trusted referee. (About 850 trusted LOC, 580 untrusted LOC.)

[1] The official board game rules allow players to identify a ship for each hit. Disallowing this results in a more complex and interesting security policy.

[2] The Battleship program of (Zheng et al. 2003) was written for the Jif/split compiler, which implemented a different language from today's Jif (Arden et al. 2013). The Jif/split Battleship program differs in some respects from the current implementation. Most importantly, in the Jif/split program, the unshot portion of each player's board is declassified at the game's end, to verify that both players placed the same number of ship cells at the game's beginning. It is unclear why this verification isn't part of the current implementation. Mainly, though, the current implementation is simply an updating of the Jif/split program to the current Jif language.

- One in CML using AC to avoid needing a trusted referee. (About 610 trusted LOC, 550 untrusted LOC.)

All three implementations employ data abstraction. In particular, the AC mechanism used by the second CML program is realized using data abstraction.

Haskell and CML are concurrent functional programming languages. Concurrent functional programming is an increasingly popular paradigm, retaining much of the elegance and simplicity of functional programming, while avoiding much of the complexity of shared variable concurrency. We made an early decision to use Haskell/LIO for one of the implementations. It was thus natural for the alternative programs to employ concurrent functional programming, and we chose CML for those programs because of its superior support for modularity, data abstraction and concurrency.

The contributions of our case study are twofold:

- **We gave rigorous, high-level specifications of security based on a non-trivial informal security policy.** Defining whole program security for Battleship was straightforward, once we settled on the right abstraction, but defining security against a malicious opponent was challenging.
- **We compared IFC, AC and data abstraction as ways of implementing a secure program.** Obtaining whole program security for Battleship was straightforward, even without using IFC or AC. But obtaining security against a malicious opponent was very challenging, and seemed to require IFC or AC.

The paper is structured as follows. Section 2 describes the rules of Battleship and the client/server architecture of our programs, and gives the definitions of program security. Section 3 describes the implementation in CML using a trusted referee. Section 4 describes the implementation in Haskell/LIO using IFC. And Section 5 describes the CML implementation using AC. Finally, Section 6 draws conclusions from the case study, and suggests directions for future research.

The source code of our case study is available on the web at:

`www.ll.mit.edu/mission/cybersec/CST/CSTcorpora/Cybersystemscorpora.html`

2. Battleship Rules, Program Architecture and Security Definitions

2.1 Battleship Rules

Battleship is a two-player, two-phase board game. In the *placing phase* of a game, each player places its five ships on a private *board*—a 10×10 grid. Each ship is one cell wide, but the ships have varying lengths: a carrier (abbreviated "c") of length 5; a battleship ("b") of length 4; a submarine ("s") of length 3; a destroyer ("d") of length 3; and a patrol boat ("p") of length 2. Ships may be placed horizontally or vertically, but not diagonally; they may not overlap.

	a	b	c	d	e	f	g	h	i	j
a										
b						b				
c	c	c	c	c	c	b				
d						b				
e						b				
f										
g			p		s	s	s			
h			p				d			
i							d			
j							d			

Figure 1. Player's Board with Ships Properly Placed

	a	b	c	d	e	f	g	h	i	j
a										
b						b				
c	c	C	C	C	C	b	$*$			
d		$*$		$*$		b				
e						B	$*$			
f										
g		$*$	p		S	S	s			
h			p				D			
i				$*$			D			
j				$*$	$*$	$*$	d			

Figure 2. Player's Board During Shooting Phase

Figure 1 contains an example of a board on which all ships have been properly placed. Board cells are indexed by *positions*, pairs (r, c), where $r, c \in \{\mathrm{a}, \ldots, \mathrm{j}\}$: r selects a row, and c selects a column within that row. We say that a ship is *positioned at* position p iff p is the position of the ship's leftmost cell—if the ship is placed horizontally—or topmost cell—if the ship is placed vertically. E.g., the battleship and carrier in Figure 1 are positioned at (b, f) and (c, a), respectively.

In the *shooting phase* of a game, the players take turns shooting cells of their opponents' boards. When the opponent shoots a cell of the player's board, the player updates its board to indicate that this cell has been shot. To show that a vacant cell has been shot, we'll put a "$*$" in the cell; to show that a cell of a ship has been shot, we'll capitalize the ship's letter. E.g., in Figure 2, the carrier's last four cells, the

battleship's last cell, and the first two cells of the submarine and destroyer have been shot, and vacant cells (c, g), (d, b), (d, d), (e, g), (g, b), (i, d) and (j, d)–(j, f) have been shot. If the opponent shoots a cell of the player's board that's already been shot, it is told that repeated shooting is illegal, and to shoot again; this process continues until a legal shot is made. When a vacant cell is shot, the opponent is told the shot was a "miss". When a cell of a ship is shot—but some of the ship's other cells remain unshot—the opponent is told it has "hit" a ship, without being told which ship was hit. But when the last unhit cell of a ship is hit, the opponent is told which ship it has sunk (e.g., "you sank my battleship"). E.g., if cell (c,a) of the board in Figure 2 were shot, the opponent would be told "you sank my carrier". When the opponent sinks the player's last ship, the opponent is told it has won the game.

2.2 On Battleship Security

In a non-computer-mediated game of Battleship, *cheating* is obviously possible. E.g., a player can

- fail to properly place all five ships during the placing phase of the game, or move a ship during the shooting phase of the game;
- lie about whether a shot was a hit or a miss, say a ship was sunk when it wasn't, or pretend a ship wasn't sunk when it was; or
- peek at its opponent's board before the game is over.

Some—but not all—cheating can be detected, by record keeping during the game and examining an opponent's board at the game's conclusion.

A player may inadvertently reveal more about its board than the opponent is entitled to know at a given stage of the game. E.g., when one's carrier is hit but not sunk, saying "you hit my carrier" can reveal more than saying "you hit a ship". We will refer to such over-revealing of information as *leaking*. Also, it might be considered wrong for a player to make use of external aids, ranging from bookkeeping to asking others for advice. We'll call this *consultation*, below.

A computer-mediated implementation of Battleship can prevent cheating and leaking—as long as all player interactions are via the computer. And, it can force any consultation to happen outside of the program. But since players may be physically separated and unsupervised, controlling consultation may not be a worthwhile goal. With computer mediation, though, the notion of the *current state of a player's board* is no longer clear-cut. In fact, it is perfectly possible for a user's understanding of the current state of its board to be inconsistent with what its opponent has learned through shooting. Such an inconsistency could be due to user interface bugs, or could have deeper roots. Unfortunately, this means that, were we to define program security from the users' viewpoints, we would be asking for something approaching functional correctness.

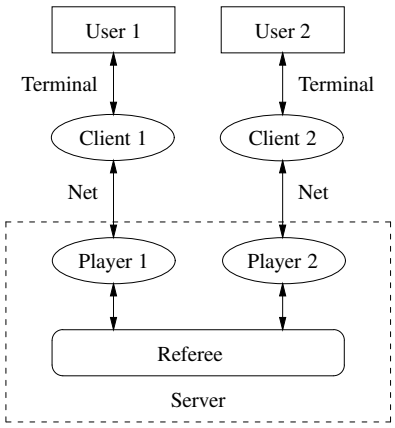

Figure 3. Program Architecture

It is useful for program security to be a weaker property than functional correctness. Furthermore, there are advantages to allowing flexibility in user interfaces, as well as in core logic. E.g., assuming we give up on trying to prevent consultation, we could accommodate core logic with a range of helpfulness and automation, ranging from not doing any record keeping, to doing (some of) the shooting automatically. This is the approach we will take.

2.3 Program Architecture

Our Battleship programs use a client/server architecture in which the interesting computation happens in a multithreaded server. The core of the server is a referee, which takes in network sockets for communicating with the players' client sides, spawns a thread[3] for each player, and then interacts with the players, running the game. The clients simply mediate between the users and the server's player threads, communicating with the users via standard input and output, and with the server via network connections. This architecture is illustrated in Figure 3.

The Battleship executable is invoked from the Unix/Linux shell, and uses command-line arguments to decide whether to start a server or client. One first starts the server running on a given host/port. Then each of the users starts a client, specifying the host/port to connect to. When game play begins, the first player to have connected will be the first player to shoot.

Our implementations don't encrypt the data flowing between players' client and server sides, nor do they perform any kind of authentication. It would be easy to write a standalone client program in which data could flow directly from client to client, not via the server, but that would also interoperate with our server. But our standard client implementation doesn't allow this, despite featuring untrusted code.

[3] In general, threads may spawn additional, supporting threads.

	a	b	c	d	e	f	g	h	i	j
a										
b										
c		+	+	+	+		*			
d		*		*						
e						+	*			
f										
g		*			+	+				
h							+			
i				*			+			
j				*	*	*				

Figure 4. Opponent's Shooting Record

	a	b	c	d	e	f	g	h	i	j
a										
b										
c	C	C	C	C	C		*			
d		*		*						
e						+	*			
f										
g		*			S	S	S			
h							D			
i				*			D			
j				*	*	*	D			

Figure 5. Subsequent Opponent's Shooting Record

2.4 Inferring Locations of Sunk Ships

In our Battleship implementations, users are kept informed by their player threads of the states of their own, private boards. Furthermore, they are reminded where they have shot on their opponents' boards. E.g., Figure 4 shows the opponent's shooting record corresponding to the player's board of Figure 2. Misses are represented by "$*$", and hits by "$+$". If the opponent were then to shoot cell (c, a), it would be told it had sunk the carrier, and cell (c, a) of its shooting record would be set to "C". The opponent can then infer that the carrier is horizontally positioned at (c, a). We have implemented an algorithm for inferring the cells of sunk ships from a shooting record. In this case, the user interface would change cells (c, b)–(c, e) from $+$ to "C".

If the opponent were then to sink the submarine by shooting cell (g, g), it would still not know whether it was positioned horizontally at (g, e) or vertically at (g, g). This means a secure implementation of Battleship must not reveal the *position* of a ship when it is sunk. Once the destroyer is sunk by hitting (j, g), a smart user (or our inference algorithm) could determine that the submarine was horizontally positioned at (g, e), with the destroyer vertically positioned at (h, g). This shooting record is shown in Figure 5, and would be displayed by our user interface.

2.5 Defining Program Security

Instead of defining program security from the users' viewpoints, we'll take a more abstract approach. A *player* will be a program abstraction, a server component with the following interface:

- A way to start the player, giving it a network socket for communicating with its client side as well as its *identity* (Player 1 or Player 2).
- A way of asking the player to choose a *complete placing board* (*CPB*), i.e., a board on which its five ships have been properly placed, but no shooting has taken place.
- A way to ask the player what position it wants to shoot next.
- A way to inform the player of the result of such a shot (illegal repetition, miss, hit of an unspecified ship, sinking of a specified ship).
- A way to tell the player where its opponent has shot.
- A way to tell the player it has won or lost the game.

In the shooting phase of the game, we work with (*shooting phase*) *boards* in which cells are annotated with whether they have been shot or not. Initially, a CPB is converted to a shooting phase board in which no cells have been shot. And there are functions for: (a) shooting a cell of a board, returning the shooting result (illegal repetition, miss, hit, sinking of a specified ship) plus the resulting board; and (b) checking whether all five ships have been sunk on a board.

Players will be required to communicate only via their interfaces (including via the network sockets they are passed for communicating with their client sides). Similarly, the client sides of players will be required to communicate only via their interfaces (with their users, via standard input and output, and with their server sides, via network sockets). In languages with low-level libraries allowing control flow or data abstraction to be compromised, we won't allow the server and client sides of players to use such libraries. For brevity in what follows, when we stipulate that a module or program component may "communicate only via its interface", this should be understood as also prohibiting it from using such low-level libraries.

Definition 2.1 (Referee Security) We say that a server's referee is *secure* iff it communicates only via its interface, doesn't directly use the network sockets for communicating with the players' client sides, and *behaves* as if it were executing the following model algorithm—as measured from the vantage points of the players:

- The referee starts up the players, giving them the network sockets for communicating with their client sides, and telling them their identities (1 and 2).
- The referee obtains CPBs from the players. These CPBs are then converted to shooting phase boards.
- The referee then enters its main loop, in which it alternates letting the players take shots at each other's boards (Player 1 goes first, then Player 2, etc.). When all ships of a player's board have been sunk, the player is told it has lost, and its opponent is declared the winner. Otherwise, the body of the loop works as follows:
  - The next player to shoot is asked where it wants to shoot on its opponent's board.
  - The shot is carried out using the shooting function, yielding a shooting result and the resulting board.
  - If the shooting result is an illegal repetition, the player is asked to choose a position again, and the opponent's board isn't changed.
  - Otherwise: the player is told the result of its shot; the opponent player is told where on its board the player shot; and the opponent's board is replaced by the board resulting from the shooting.

Because a player can't tell when the referee interacts with its opponent, this definition *doesn't* impose a total ordering on the referee's interactions with the players. E.g., after calculating the result of a shot, the actions of (a), telling the player the result of its shot, and (b), telling the player's opponent where on its board the player shot, could be carried out in either order. On the other hand, the referee can't, e.g., delay action (b) until after it asks the player's opponent where it wants to shoot next, since the player's opponent would notice such a delay.

Definition 2.2 (Program Security) We say that a program is *secure* iff its referee is secure, and the client and server sides of its players communicate only via their interfaces.

According to this definition, the client and server sides of players are untrusted—need not be audited—except for the requirement of communicating only via their interfaces. This allows for players that mislead their users, as well as for players with varying degrees of automation and helpfulness.

The reader should compare our definition of program security with the informal security policy of the introduction. Because the referee obtains a complete placing board from each player and subsequently manages both players' boards, part (1) of the informal policy is ensured by our definition. And part (2) is a consequence of the algorithm followed by the referee. But note our definition's requirement that the referee keep a player informed of the shots made on its board. This requirement was implicit in the informal policy.

The reader might be wondering whether the requirement that a referee behave *exactly* like the model referee of Definition 2.1 is overly restrictive. For example, it might seem safe

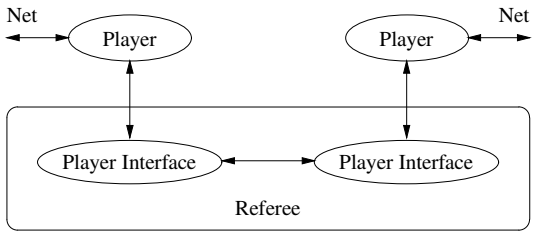

Figure 6. Server with Referee Made of Player Interfaces

to allow a referee to repeat messages to players. But message repetition could be used by a referee to convey information to one of the players. E.g., suppose the referee wants to tell Player 2 that Player 1 has an unshot ship cell at the jth cell of the ith row of its board, for $i, j \in \{0, \ldots, 9\}$. It could do this by repeating a message to Player 2 $10i + j + 1$ times. We'll leave to future work the question of whether Definition 2.1 may safely be weakened.

The most straightforward way of implementing a referee is to directly implement the model referee algorithm. This, in fact, is what we've done in the first of our CML implementations—see Section 3. However, if we don't want to make use of a trusted third party, we can split the referee into mutually distrustful components, which we'll call *player interfaces* (*PIs*), as in Figure 6. With this architecture, the referee spawns two player interfaces, passing them the network sockets for their players' client sides, their identities (1 and 2) as well as a means of communicating with each other. The player interfaces then spawn their players, passing them their network sockets and identities.

This splitting of the referee into mutually distrustful player interfaces allows each PI to be written and maintained by a different software development team. This may be sensible when the teams are able to agree on a protocol for PI interaction, but when each team wants some freedom to choose how to implement that protocol. Our standard definition of program security applies to this architecture: we must convince ourselves that the referee consisting of the composition of the two player interfaces behaves as it should. In particular, given a "good" player interface, G, we may convince ourselves that the referee consisting of the composition of G with itself behaves correctly. But we also want to say what it means for G to be secure against an arbitrary possibly malicious opponent. We give this definition by borrowing ideas from the real/ideal paradigm of theoretical cryptography (Canetti 2000).

Definition 2.3 (Security Against Malicious Opponent) Suppose G is a player interface such that the referee consisting of the composition of G with itself is secure. Then G is *secure against any malicious opponent* iff, for all player interfaces M that communicate only via their interfaces, there exists a player $S(M)$—a "simulator" based on M—such that the referees of Figure 7 *behave* identically from

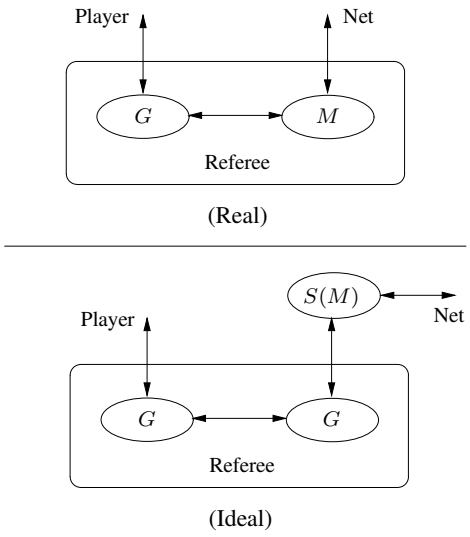

Figure 7. Simulating Malicious Player Interface M by Player $S(M)$

the vantage points of the player (on the left) and the network socket (on the right). (M isn't depicted as creating a player, since it may or may not do so.)

Since the program in which the referee is G composed with itself—the "ideal" program—is assumed to be secure, this tells us that M doesn't compromise G's security in the "real" program. M is sandboxed in $S(M)$, which must function without having more information or control than any other player. In the real program, M can't modify its player's board during the shooting phase, because the player $S(M)$, in the ideal program, must supply a CPB before the shooting phase begins. In the real program, M can't exfiltrate to the network socket information about G's player's board to which it isn't entitled, since then $S(M)$ would be able to achieve the same result in the ideal program. And M can't cause G to mislead its player in the real program, because then $S(M)$ would be able to do this in the ideal program.

M may behave in such a way that G detects an error of the PI interaction protocol. In such a case, G will cause the program to abort, and part of the proof that the simulator player $S(M)$ works as it should consists of showing that $S(M)$ will also cause a program abort in such a circumstance.

In Sections 4 and 5, we'll consider two—quite different—ways of implementing a player interface that is secure against any malicious opponent: one in LIO using IFC, and one in CML using AC. In both programs, the PIs begin by exchanging "protected" boards—b_1 and b_2, in Figure 8. Through this exchange, each PI commits to its complete placing board (CPB). Subsequently, the PIs take turns sending shot messages (s_1, s_2, ...) to their opponents, who then send back shot response messages (r_1, r_2, ...).

It is easy to define a player interface that—when composed with itself—is secure. But defining a player interface

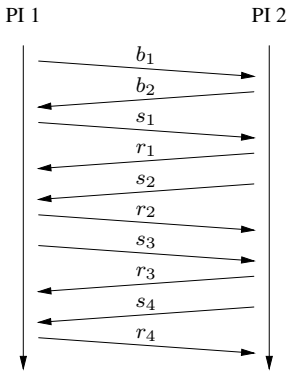

Figure 8. Player Interface Message Exchange

that's secure against a malicious opponent is challenging. In fact, it took multiple iterations for our designs to stabilize. During this process, we found our definition of security against a malicious opponent to be invaluable. Armed with this definition, we were better able to find security flaws.

3. Implementation in CML with a Trusted Referee

We now consider our implementation of Battleship in CML using a trusted referee. CML (Reppy 1999) is a library for the Standard ML (SML) (Milner et al. 1997) programming language. SML is a mostly functional language with static scoping, strong static typing, eager (as opposed to lazy) evaluation, and a sophisticated module language supporting data abstraction. CML provides lightweight threads and synchronous communication over channels.

This implementation of Battleship is straightforward. Its main modules are:

- A trusted `Ship` module providing a datatype `ship`.
- A trusted `Board` module implementing abstract datatypes of placing boards (on which some ships are properly placed), complete placing boards (CPBs, on which all ships are properly placed) and shooting phase boards.
- An untrusted `Player` module implementing the players of Section 2.
- An untrusted `Client` module implementing the client side of a player.
- A trusted `Referee` module implementing the referee of Section 2.
- A trusted `Main` module defining the program's entry point.

Auditing this program is also straightforward. We must check that the implementations of `Player` and `Client` don't communicate except via their interfaces. Most importantly, we must check that the referee implemented by `Referee` is secure (see Definition 2.1). This is obvious, as the referee is a direct implementation of the model referee algorithm.

4. Implementation in LIO with IFC

We now consider our implementation of Battleship in LIO using IFC to avoid the need for a trusted referee.

4.1 LIO

LIO (Stefan et al. 2011) is an IFC library for Safe Haskell (Terei et al. 2012), an extension of Haskell that facilitates safe execution of untrusted code. (Safe) Haskell is a mostly functional language with static scoping, strong static typing, lazy evaluation, and a simple module language. It supplements a purely functional base language with monads, one of which (`IO`) implements input/output, mutable variables (mvars) and multi-threading.

Data in LIO may be explicitly labeled with a security *label*, and is always implicitly labeled. Labels form a lattice, ordered by a *can flow to* relation, $\sqsubseteq$, describing when data with one label may flow to data with another label. The evaluation context contains two labels, a *current label* (*CL*)[4] and a *clearance*. LIO ensures the CL is always an upper bound of the labels of all observed values. The clearance is an upper bound on how high the CL may be raised. When a value is explicitly labeled with label l, l must be greater than or equal to the CL, and less than or equal to the clearance. Unlabeling a labeled value raises the CL to the least upper bound of the CL and the labeled value's label. LIO uses a labeled version of the IO monad—called `LIO`—to keep track of the CL and clearance. A value of type `LIO` a b, where a is a label type and b is a type, is a computation that, if run with a starting CL and clearance, will produce a value of type b, possibly changing the CL and clearance in the process.

The standard LIO label type is `DCLabel`, whose values are disjunction-category (DC) labels (Stefan et al. 2012). A DC label has the form $\langle s, i\rangle$, where s and i are propositional formulas whose atoms are principals, with s corresponding to *secrecy* and i to *integrity*. The secrecy part describes the combination of principals needed to *declassify* a value, and the integrity part refers to the combination of principals that have *endorsed* a value. The *can flow to* relation for DC labels is defined by: $\langle s, i\rangle \sqsubseteq \langle s', i'\rangle$ iff s' implies s, and i implies i'. `DCLabeled` a is the type of DC-labeled values of type a. Public data is labeled $\langle$`True`, `True`$\rangle$, `dcPublic`. The type constructor `DC` is an abbreviation for `LIO DCLabel`; a value of type `DC` a is a `DC` *action*, a computation using DC labels that produces a value of type a.

LIO has unforgeable *privileges* corresponding to formulas. Possession of privileges allows the label-related rules to be relaxed. If the privilege pr corresponds to the formula ϕ, then $\langle s, i\rangle \sqsubseteq_{pr} \langle s', i'\rangle$, i.e., $\langle s, i\rangle$ *can flow to* $\langle s', i'\rangle$ *when possessing* pr, iff $\phi \wedge s'$ implies s and $\phi \wedge i$ implies i'. E.g., when unlabeling a labeled value, one can use a privilege to reduce the raising of the CL.

Privileges may also be used to relabel labeled values. Suppose, e.g., the privilege pr corresponds to the principal q, and the labeled value x consists of a value v and label $\langle q, r\rangle$—classified by q and endorsed by r. Then pr can be used to *declassify* x, producing a labeled value consisting of v with label $\langle$`True`$, r\rangle$; it can also be used to *endorse* x, producing a labeled value with label $\langle q, q \wedge r\rangle$—classified by q, and endorsed by both q and r.[5]

4.2 Program Structure

The main modules of the LIO Battleship program are:

- A trusted `Ship` module providing a datatype `Ship`.
- A trusted `Board` module implementing abstract datatypes of placing boards and complete placing boards (CPBs), but also implementing *labeled boards*.
- An untrusted `Player` module implementing the server side of a player.
- An untrusted `Client` module implementing the client side of a player.
- A trusted `PlayerInterfaceMsg` module implementing a type of player interface messages, along with a way of creating a pair of functions for communicating messages via an mvar.
- A trusted `PlayerInterface` module implementing player interfaces.
- A trusted `Main` module, defining the program's entry point.

`Main` has the Safe Haskell Mode "Unsafe", allowing it to use unsafe operations for managing network connections, creating IO handles, and creating privileges. IO handles and mvars are labeled objects. To use them, the CL and the label of the handle or mvar must be the same, modulo the possession and use of applicable privileges.[6] When the program begins, the CL and clearance will be `dcPublic` and $\langle$`False`, `True`$\rangle$, respectively. This allows the CL's secrecy to rise, and means that privileges must be possessed and used to endorse data. However, during normal operation, the CL will remain fixed at `dcPublic`.

In server mode, `Main` forks threads for the two player interfaces, giving them network handles for communicating with their players' client sides, functions for communicating with each other, and their identities. Each player interface is also given the privilege corresponding to its principal—`"player1"` or `"player2"`. The network handle given to the PI—and then to the player it starts—is labeled `dcPublic`, and the mvars underlying its functions for communicating with the other PI are also labeled `dcPublic`. The PI is started with a CL of `dcPublic`.

Because the `Player` and `PlayerInterface` modules don't have access to unsafe operations, player interfaces and the players they create are automatically sandboxed. A PI

[4] CL is often called the program counter (PC) label in the literature.

[5] Subject to the new labels being less than or equal to the clearance.

[6] The CL will automatically be raised, if necessary.

```
data LSR = MissLSR | HitLSR | SankLSR Ship.Ship
data LC =
  LC (DCLabeled
      (Principal,  -- originating principal
       Principal,  -- receiving principal
       Pos,        -- position of cell
       DC LSR))    -- DC action for shooting cell
data LB  -- abstract type
sub :: LB -> Pos -> LC
update :: LB -> Pos -> LC -> LB
data LBC  -- abstract type
completeToLBC ::
  Complete -> Principal -> DCPriv -> DC(Maybe LBC)
lbcToLB ::
  LBC -> Principal -> DCPriv -> DC(Maybe LB)
```

Figure 9. Labeled Boards Part of `Board` Module

can only communicate via the functions it is given for communicating with its opponent PI, and via the network handle it is given for communicating with its player's client side. This guarantee is specific to LIO. In another IFC language, we might have needed to use IFC to stop PIs and players from communicating inappropriately.

In client mode, `Main` uses the untrusted `Client` module to run a player's client side, passing the client IO handles for the standard input and output (for communicating with the user) and for the network connection to the client's server side. Because `Client` doesn't have access to unsafe operations, clients are automatically sandboxed.

4.3 Labeled Boards

The trusted `Board` module implements labeled boards. More precisely the datatypes, abstract types and functions of Figure 9 are provided by `Board`. The type `LB` of *labeled boards* is an abstract type, and labeled boards are made up of *labeled cells*, which are elements of the concrete type `LC`. The functions `sub` and `update`—`Pos` is the abstract type of board positions—are used to look up and update cells of labeled boards, respectively.

To elucidate the design of labeled cells, we'll start by considering how labeled cells could have been implemented if, upon shooting a cell, a player was only to be told whether the shot was a miss or a hit. In that case, we could try these definitions of `LSR` (*labeled shot result*) and `LC`:

```
data LSR = MissLSR | HitLSR
data LC = LC (DCLabeled LSR)
```

Suppose that a player interface (the "originating PI", or "OPI") wants to safely share its board with its opponent (the "receiving PI", or "RPI"), thus committing to that board. It can turn each cell of its board into a labeled cell with label $\langle oprin, oprin\rangle$, where $oprin$ is the OPI's principal. The resulting labeled board can then be sent to the RPI. When the RPI wants to shoot a cell of the OPI's board, it can't declassify the labeled cell itself, since it doesn't have the OPI's privilege. And were it to unlabel the labeled cell, this would add $oprin$ to the secrecy part of its CL, making it unable to communicate with either its player's client side or the OPI. This is since the IO handle and mvars via which it communicates are labeled `dcPublic`. Consequently, the RPI must send the labeled cell to the OPI for declassification. To ensure the labeled cell it gets back from the OPI may be trusted, the RPI endorses the labeled cell first, giving it label $\langle oprin, oprin \wedge rprin\rangle$, where $rprin$ is the RPI's principal. When the labeled cell is returned by the OPI, the RPI can check that its label is $\langle \texttt{True}, oprin \wedge rprin\rangle$—declassified, but still carrying its endorsement. It can then unlabel the labeled cell, without raising the CL, getting the labeled shot result.

There is a problem with this version of the protocol, however: nothing prevents the OPI from carrying out a "replay attack", sending a previously declassified labeled cell back to the RPI. To prevent such attacks, we can add a cell's position to its data:

```
data LC = LC (DCLabeled(Pos, LSR))
```

The goal is for the RPI to be able to verify that the labeled cell it sent to the OPI for declassification was returned to it. But the cell position can only be read by the RPI once the cell is declassified, and thus there is nothing stopping the OPI from providing a labeled board with incorrect positional information. Our solution to this problem is to do the construction of the labeled board from a CPB in trusted code.[7] This allows a second flaw in the protocol to be remedied, stopping the OPI from handing over a labeled board that doesn't correspond to any CPB. Such cheating might eventually be detected, but our definition of security against a malicious opponent requires that it be impossible to attempt. The inclusion of a cell's position in its data is also needed so the OPI can send that position to its player when the cell is shot.

One final problem is that a declassified labeled cell's label $\langle \texttt{True}, prin_1 \wedge prin_2\rangle = \langle \texttt{True}, prin_2 \wedge prin_1\rangle$ has an ambiguous origin—it might have originated from either PI. This would allow a PI to send a declassified labeled cell it had received from its opponent back to the opponent as a shot result. Consequently, we add the principals of the originating and receiving PIs to a labeled cell:

```
data LC =
  LC (DCLabeled
      (Principal, Principal, Pos, LSR))
```

(the originating principal first, followed by the receiving principal).

Now, let's return to the actual game, where a player learns it has sunk a ship upon shooting its last unshot cell. Here the datatype of labeled shot results is:

```
data LSR = MissLSR | HitLSR | SankLSR Ship.Ship
```

[7] Another solution is for the RPI to endorse a labeled cell at position (r, c) with formula $rprin \vee x$, where x is the conversion of (r, c) into a principal, making it unnecessary to include the position in the cell's data.

But it is no longer sufficient to include a labeled shot result in a labeled cell, since shooting a cell of ship *ship* may yield `HitLSR` or `SankLSR` *ship*, depending upon whether all other cells of the ship were previously hit. Instead, we switch to this definition of `LC`:

```
data LC =
  LC (DCLabeled
      (Principal, Principal, Pos, DC LSR))
```

With this design, a labeled cell contains a `DC` action that, when run, returns a labeled shot result. For this to work, the `DC` actions of the cells of a ship must share a private mvar[8] recording the positions of the cells of the ship that have been hit so far. Running the `DC` action of a cell of ship *ship*, causes the position of the cell to be added, if necessary,[9] to the contents of *ship*'s mvar. When the updated list of positions contains all the ship's cell's positions, the `DC` action returns `SankLSR` *ship*; otherwise, it returns `HitLSR`.

This design seems plausible, but suffers from a serious flaw: once a ship, *ship*, has been sunk, an RPI can determine the other cells of the ship by re-running the `DC` actions of all previously hit cells that *could* be part of the ship, looking for result `SankLSR` *ship*. Going back to Figure 4 of Section 2, after the RPI sank the submarine by shooting cell (g, g), it could re-run the `DC` actions of cells (g, e) and (g, f), yielding `SankLSR Submarine` in both cases. The fix is for the `DC` action associated with a cell of ship *ship* to return `SankLSR` *ship only* when the updated contents of the ship's mvar contains the positions of all of the ship's cells, *and* the cell's position was the *last* position to be added to the mvar. This design has the property that running the `DC` action of a labeled cell more than once has no effect, and returns the labeled shot result that was returned the first time the `DC` action was run.

One final issue remains: the RPI is capable of running the `DC` actions of the declassified labeled cells it receives in a different order from the one in which the OPI sent the cells to it. Returning to Figure 4, if the RPI sent the labeled cells at positions (g, e), (g, f) and (g, g) to the OPI for declassification, in that order, it's not entitled to learn that the cell at (g, e) is part of the submarine. But by running the `DC` actions of these cells in reverse order, that's what it would learn. The fix is for the OPI to run a labeled cell's `DC` action, before returning the declassification of the cell to the RPI.

As we explained above, the construction of a labeled board from an OPI's CPB is carried out by trusted code, so as to ensure the labeled board corresponds to a CPB. This construction is initiated by the OPI, but must also involve the RPI, because otherwise the OPI could modify the constructed labeled board before sending it to the RPI. Consequently, the construction of the labeled board is carried out in two steps.

First, the OPI sends the RPI a value of the *labeled board closure* abstract type, `LBC`. This value is produced using the function `completeToLBC`, which takes in the OPI's CPB $compl$, principal $oprin$, and corresponding privilege, and returns a `DC` action that optionally delivers a value of type `LBC`. The `DC` action only returns `Nothing` when the principal and privilege are inconsistent, or when the CL or clearance would prevent the action from succeeding. Otherwise, it returns `Just` of a labeled board closure lbc.

The RPI uses the function `lbcToLB` to turn lbc into a labeled board. This function also takes in the RPI's principal $rprin$, and corresponding privilege. The resulting `DC` action returns `Nothing` if the supplied principal and privilege don't agree, or the principal is the same as $oprin$, or the CL or clearance won't allow the labeled board to be constructed, or lbc had already been converted to a labeled board. Otherwise, it returns `Just` of the labeled board that is consistent with $compl$, whose cells are all labeled $\langle oprin, oprin \wedge rprin \rangle$ and have $oprin$ and $rprin$ as their originating and receiving principals, respectively, and whose private mvars are empty.

4.4 Player Interface Protocol

The PIs begin by exchanging the labeled board closures constructed from their CPBs. Each PI converts the labeled board closure it receives into a labeled board. Should this conversion fail, it will be because it was sent its own labeled board closure, or since the other PI converted the labeled board closure to a labeled board. This is a protocol violation, and results in the program being terminated.[10] Each PI knows that at most one labeled board was constructed from its CPB. The shooting phase of the game then begins.

When an RPI wants to shoot a labeled cell on behalf of its player, it first sees if the labeled cell is already declassified, asking its player to choose another position to shoot, if that's the case. Otherwise, it sends the labeled cell to the OPI for declassification. When given a labeled cell to declassify, the OPI checks that the labeled cell really did originate from it, and that the labeled cell's position isn't in its list of shot positions. Otherwise, it signals a protocol violation. It then runs the `DC` action of the labeled cell, and tells its player which of its cells has been shot. If the resulting labeled shot result says that one of its ships was sunk, it makes a note of that ship; when all of its ships have been sunk, it tells its player it has lost the game. The OPI is able to trust the `DC` action's result since it knows the labeled cell is part of the unique labeled board that originated from it. The OPI then sends the declassification of the labeled cell back to the RPI.

Upon receipt of a declassified labeled cell, the RPI checks that the labeled cell still has its endorsement. It then unlabels it, checking that the cell's position is correct and the cell's receiving principal is its own principal. If either of these checks fails, this is a protocol violation. Otherwise, it runs

[8] We'll give them label `dcPublic`.

[9] The `DC` action might already have been run.

[10] This is always the response to a protocol violation.

the cell's DC action, telling its player the shot's result, and updating its record of which of its opponent's ships are yet to be sunk. (If the OPI hadn't already run the cell's action, this will make no difference.) When no such ships remain to be sunk, it tells its player it has won the game.

4.5 Auditing

The auditing process for this Battleship implementation has two parts. First, we must convince ourselves that our program is secure, i.e., that the referee consisting of the composition of our standard player interface, G, with itself is secure (see Definition 2.1). This is straightforward.

Second, we must show that G is secure against any malicious opponent (Definition 2.3). Here, we must show that any possibly malicious player interface M that communicates only via its interface may be transformed into a simulator player $S(M)$, in such a way that the referees of Figure 7 behave identically from the vantage points of the player (on the left) and network socket (on the right). $S(M)$ consists of supervisory code, which runs M in a sub-thread. The goal in constructing $S(M)$ is for M in the real program to remain in sync with the simulated version of M in the ideal program; we also need that the messages exchanged between player interfaces in the real and ideal programs remain in sync, except when M in the real program has violated the PI protocol, in which case this violation must be detected by $S(M)$.

$S(M)$ will know the principal of M (`"player1"` or `"player2"`), but it won't have access to the corresponding privilege. Thus the version of M that runs in a sub-thread of $S(M)$ will have to run in a variant of the `LIO` monad. Since M can't exfiltrate its privilege to a thread it didn't create, the simulation of M's use of its privilege is feasible.

The key to this construction is the realization that the simulated version of M will have two versions of labeled board closures, labeled boards and labeled cells, *coexisting* in the same three types:

- Normal ones, but where the supervisory code of $S(M)$ has access to special inspection functions. E.g., when the simulated version of M produces a labeled board closure lbc, the supervisory code needs to be able to look up the CPB, $compl$, that it represents, so this can be returned as the CPB chosen by $S(M)$.
- Partially defined ones, which the supervisory code of $S(M)$ will gradually make more and more defined. E.g., the labeled board closure supplied by $S(M)$'s supervisory code to the simulated version of M is turned into a labeled board in which the positions of ships are undefined, in contrast to the real labeled board derived by M from the labeled board closure received from G. As G gives $S(M)$ the results of the shots it makes on behalf of the simulated version of M, the supervisory code of $S(M)$ uses special functions to "patch in" this information to the labeled cells of the board, via side-effects. In this way, when M in the real program learns something about G's board, the simulated version of M in the ideal program is also able to learn exactly the same thing.

Space limitations don't allow us to give the details of the construction of $S(M)$. We have sketched the proof that this construction works, but haven't written the full proof yet.

```
type ck and kb and lb and tlb
val labelKey : Key.key * int -> ck
val idOfTLB : tlb -> bool
val lbOfTLB : tlb -> lb
val completeToBoardPair :
      complete -> kb * (bool -> tlb)
val keyedAllSunk : kb -> bool
val lockedAllSunk : lb -> bool
val keyedShoot : kb * pos -> kb * Key.key option
val lockedAlreadyShot : lb * pos -> bool
datatype lsr = InvalidLSR | RepeatLSR | MissLSR
             | HitLSR | SankLSR of Ship.ship
val lockedShoot : lb * pos * ck -> lb * lsr
```

Figure 10. Locked Boards Part of Board Module

5. Implementation in CML with AC

The LIO Battleship implementation of Section 4 uses IFC for implementing labeled cells. However, that use of IFC amounts to an application of AC: if a receiving PI (RPI) unlabels a labeled cell that's still classified by the originating PI (OPI), this will stop the RPI from subsequently communicating with its player's client side or with the OPI.

In this section, we'll consider an implementation of Battleship in CML that explicitly uses AC. This implementation could be directly translated into LIO; we used CML to emphasize the point that it didn't require use of IFC. This program is an amalgam of our first CML solution (Section 3) and the LIO solution. In this solution, a PI shares a locked board with its opponent PI, and then provides keys to unlock the board's cells, in response to shot requests.

5.1 Keys, Counted Keys and Keyed/Locked Boards

There is a trusted `Key` module implementing an abstract type `key` of unforgeable *keys*:

```
type key
val newKey : unit -> key
val sameKey : key * key -> bool
```

Each call of `newKey` returns a distinct key, and `sameKey` tests keys for equality.

The trusted `Board` module now implements keyed and locked boards, in addition to placing and complete placing boards. More precisely, the abstract types, datatype and functions of Figure 10 are provided by `Board`. There is an abstract type `ck` of *counted keys*—keys labeled by counters. Counted keys are created using `labelKey`; crucially, no way of destructing keys is exported from `Board`.

Keyed (`kb`) and *locked* (`lb`) *boards* are immutable data structures. Each of their cells contains a *membership*, saying

whether it is part of a ship, and, if so, which ship. Keyed and locked boards come in pairs: a cell of the keyed board contains a key for unlocking the corresponding cell of the locked board, unless the keyed board's cell was already shot (using `keyedShoot`). Locked boards contain counters: to shoot a cell of a locked board, one needs a counted key whose counter is equal to the board's counter and whose key will unlock the cell.

The process of shooting/unlocking a cell of a locked board (using `lockedShoot`) isn't local to that cell, if the cell is part of a ship: it must be determined whether the ship's other cells are all unlocked, so as to know whether to return a locked shot result of `HitLSR` or `SankLSR` of that ship. This is why the type `lb` of locked boards is abstract, and its locked cells aren't exposed.

A value of abstract type `tlb` is a *totally locked board*, consisting of a boolean identity (`true` = Player 1, `false` = Player 2), plus a locked board whose counter is 0 and in which all cells are locked. The functions `idOfTLB and lbOfTLB` select the components of a totally locked board.

5.2 Player Interface Protocol

An OPI calls `completeToBoardPair` with its complete placing board (CPB) $compl$. This creates a matching keyed/locked board pair (kb, lb), where the memberships of the cells of kb and lb correspond to the memberships of the cells of $compl$, and the counter of lb is 0, and then returns $(kb, tlbFun)$, where $tlbFun$ takes an identity id and returns the `tlb` with identity id and locked board lb. The OPI calls $tlbFun$ with its identity, and sends the resulting `tlb` to the RPI. The RPI checks that the identity of the totally locked board it has received isn't its own identity. If this isn't true, a protocol error has occurred—the OPI may have been trying to send the RPI's `tlb` back to it. Otherwise, it uses `lbOfTLB` to retrieve the OPI's locked board, which is guaranteed to have no unlocked cells.

Once each PI has turned the totally locked board it received into its opponent's locked board, we enter the shooting phase of the game. Each PI maintains a *key counter*, which is initially 0.

When an RPI wants to shoot a given cell of the OPI's locked board, it first uses `lockedAlreadyShot` to ensure it hasn't already shot that cell. It then sends the cell's position to the OPI. The OPI uses `keyedShoot` to shoot the specified cell of its keyed board, yielding an updated board and an optional key. If this optional key is `NONE`, that cell has already been shot—a protocol error. Otherwise, the OPI labels the key with its key counter, sends the resulting counted key back to the RPI, and increments the key counter by 1.

The RPI then uses the counted key to shoot the cell of the locked board, using the function `lockedShoot`, which returns an updated locked board, plus the result of the shooting. A result of `InvalidLSR` means that either the counted key's counter wasn't equal to the locked board's counter, or the counted key's key wouldn't unlock the locked board's cell; this is a protocol error. It isn't possible that `RepeatLSR` will be the result, as the RPI already verified it wasn't repeating a shot. When the shot succeeds, the counter of the updated locked board is one more than the original locked board's counter; otherwise, the updated locked board is the same as the original one.

The functions `keyedAllSunk` and `lockedAllSunk` are used by the PIs for determining whether all ships of a keyed or locked board are sunk; when this happens, the PIs inform their players of the game's result.

5.3 Assessment and Auditing

To understand the role of counters in our design, consider the consequence of dispensing with them. Locked boards are immutable, and the RPI could keep a copy of the initial locked board, lb, plus a record of the keys it has been sent, associating them with their positions. After having sunk a ship, $ship$, it could learn the ship's position, as follows. If the cell at position pos *could* be part of $ship$, the RPI would apply the keys with positions other than pos to lb, yielding lb', and then use pos's key to shoot position pos of lb', looking for a result of `SankLSR` $ship$. The use of counters makes it useless to re-shoot an earlier locked board, which can only be shot with the unique counted key provided by the OPI for that purpose. Of course, it is crucial that `ck` be an abstract type: otherwise, an RPI could extract the key of a counted key, and construct new counted keys from that key.

This implementation is simpler than the LIO implementation for two reasons:

- Keyed and locked boards don't involve mutable state, unlike the labeled boards of the LIO implementation.
- The messages exchanged by player interfaces during the shooting phase are much simpler: positions and counted keys are simpler to reason about than labeled cells.

The auditing process for this Battleship implementation is similar to, but simpler than, that of the LIO implementation. First, we must convince ourselves that our program is secure, i.e., that the referee consisting of the composition of our standard PI, G, with itself behaves identically to the model referee algorithm. This is straightforward. Second, we must show that G is secure against any malicious opponent. As in the LIO case, the proof that a simulator $S(M)$ can be correctly defined has only been sketched to date.

6. Conclusions

We defined whole program security by specifying that a referee must behave identically to a model implementation from the players' vantage points, and defined security of a player interface against a malicious opponent using the real/ideal paradigm. Although these definitions are *sufficient* for ensuring security, they are certainly not *necessary*. E.g., just because a program doesn't implement the player abstraction of Section 2.5, it doesn't follow that we should con-

sider it insecure. Further research is needed on definitional formalisms for whole program security.

We found that whole program security was easily achievable, even without using IFC or AC. On the other hand, achieving security against a malicious opponent was challenging, and seems to require using IFC or AC. Our AC solution in CML is considerably simpler than the IFC solution in LIO.

We found data abstraction to be a powerful supporting technique for achieving program security. In particular, the AC mechanism used by the second CML program is realized using data abstraction.

LIO allows modules to be sandboxed, preventing them from using low-level libraries or communicating except via their interfaces, and we found this facility extremely useful. There is no obvious reason why sandboxing couldn't be added to Standard ML/CML, but to our knowledge, this hasn't yet been done.

The paradigm of splitting trusted code—in our case, the referee—into mutually distrustful modules—in our case, player interfaces—is a natural one, but it is far from clear when it makes sense to do this. In both our LIO and CML + AC implementations, we needed some additional trusted code to facilitate the splitting of the referee. When the trusted code needed to carry out splitting grows too large, it will call into question the utility of the splitting.

There is also the question of what it was about Battleship that made it possible to implement mutually distrustful player interfaces without using IFC. Part of the answer is that, in Battleship, once information is declassified, it is completely public. Applications in which untrusted components are asked to carry out sensitive computations, the results of which will remain classified, might need IFC. But sometimes effective sandboxing (as in LIO) is sufficient to control such information flow.

We believe there is a great need for more case studies in language-based security. The balance between theory (e.g., non-interference results) and practice seems heavily on the theory side. Actual case studies—warts and all—are a crucial way of grounding the field.

Acknowledgments

This work benefited from discussions with Robert Cunningham, Cătălin Hriţcu, Amit Levy, David Mazières, Benjamin Pierce, Emily Shen, Gregory Sullivan, Mayank Varia and Mitch Wand. We are indebted to Deian Stefan for his explanations of LIO's intricacies. We wish to thank the anonymous referees for their detailed feedback on our submission.

References

O. Arden, S. Chong, A. Myers, K. Vikram, and D. Zhang. Jif Distribution, Version 3.4.1, April 2013. `www.cs.cornell.edu/jif`.

A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. of the 10th European Conference on Research in Computer Security*, ESORICS'05, pages 197–221. Springer-Verlag, 2005.

R. Canetti. Security and composition of multi-party cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 47–60, Berkeley, CA, USA, 2012. USENIX Association.

C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *Proc. of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 3–17. IEEE Computer Society, 2013.

B. W. Lampson. Protection. In *Proc. of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443. Princeton University, 1971. Reprinted in *Operating Systems Review*, 8, 1, January 1974, pages 18–24.

J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. of the ACM Symposium on Operating Systems Principles*, pages 321–334. ACM, 2009.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML—Revised 1997*. MIT Press, 1997.

A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM, 1999.

A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 129–142. ACM, 1997.

J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Sept. 2006.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. *SIGPLAN Notices*, 46(12):95–106, Sept. 2011.

D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Proc. of the 16th Nordic Conference on Information Security Technology for Applications*, NordSec'11, pages 223–239. Springer-Verlag, 2012.

D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières. Safe Haskell. In *Proc. of the 2012 Haskell Symposium*, pages 137–148. ACM, 2012.

S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, 2004.

L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. of the 2003 IEEE Symposium on Security and Privacy*, pages 236–250. IEEE, 2003.