

ABSTRACT

Title of Dissertation: Advanced Language-based Techniques
 for Correct, Secure Networked Systems

James Parker
Doctor of Philosophy, 2020

Dissertation Directed by: Professor Michael Hicks
 Department of Computer Science

Developing correct and secure software is an important task that impacts many areas including finance, transportation, health, and defense. In order to develop secure programs, it is critical to understand the factors that influence the introduction of vulnerable code. To investigate, we developed and ran the Build-it, Break-it, Fix-it (BIBIFI) security-oriented programming contest as a quasi-controlled experiment. BIBIFI aims to assess the ability to securely build software, not just break it. In BIBIFI, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teams' submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. We ran three contests involving a total of 156 teams and three different programming problems. Quantitative analysis from these contests found that

the most efficient build-it submissions used C/C++, but submissions coded in a statically-typed language were less likely to have a security flaw. Break-it teams that were also successful build-it teams were significantly better at finding security bugs.

The contest results showed that programmers make mistakes in both the design and implementation of their programs in ways that make it vulnerable. To mitigate these issues, we advanced the state of the art in language-integrated techniques for security enforcement, including formal methods. First we created **LWeb**, a tool for enforcing label-based, information flow policies in database-using web applications. In a nutshell, **LWeb** marries the **LIO** Haskell IFC enforcement library with the **Yesod** web programming framework. The implementation has two parts. First, we extract the core of **LIO** into a monad transformer (**LMonad**) and then apply it to **Yesod**'s core monad. Second, we extend **Yesod**'s table definition DSL and query functionality to permit defining and enforcing label-based policies on tables and enforcing them during query processing. **LWeb**'s policy language is expressive, permitting dynamic per-table and per-row policies. We formalize the essence of **LWeb** in the λ_{LWeb} calculus and mechanize the proof of noninterference in Liquid Haskell, an extension of Haskell that adds *refinement types* to the language. This mechanization constitutes the first metatheoretic proof carried out in Liquid Haskell. We also used **LWeb** to build the web site hosting BIBIFI. The site involves 40 data tables and sophisticated policies. Compared to manually checking security policies, **LWeb** imposes a modest runtime overhead of between 2% to 21%. It reduces the trusted code base from the whole application to just 1% of the application code, and 21%

of the code overall (when counting LWeb too).

Finally, we further advance the capabilities of Liquid Haskell by using it to verify the correctness of distributed applications based on *conflict-free replicated data types* (CRDTs). To do so, we add an extension to Liquid Haskell that facilitates stating and semi-automatically proving properties of typeclasses. Our work allows refinement types to be attached to typeclass method declarations, and ensures that instance implementations respect these types. The engineering of this extension is a modular interaction between GHC, the Glasgow Haskell Compiler, and Liquid Haskell's core proof infrastructure. To verify CRDTs, we define them as a typeclass with refinement types that capture the mathematical properties CRDTs must satisfy, prove that these properties are sufficient to ensure that replicas' states converge despite out-of-order delivery, implement (and prove correct) several instances of our CRDT typeclass, and use them to build two realistic applications, a multi-user calendar event planner and a collaborative text editor. In addition, we demonstrate the utility of our typeclass extension by using Liquid Haskell to modularly verify that 34 instances satisfy the laws of five standard typeclasses.

Advanced Language-based Techniques for
Correct, Secure Networked Systems

by

James Parker

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Advisory Committee:
Professor Michael Hicks, Chair/Advisor
Dr. Niki Vazou
Professor Michelle Mazurek
Professor David Mount
Professor Lawrence Washington

© Copyright by
James Parker
2020

Acknowledgments

There are numerous people in my life who have helped make this dissertation possible. First, I would like to thank my advisor, Michael Hicks who has been an invaluable mentor and supporter of my research. In addition to everything he's taught me academically, he has served as a role model. It is obvious how much he cares for his students, and he always aims to do what is best.

During her time as a postdoctoral researcher at the University of Maryland, Niki Vazou exposed me to the wonderful world of Liquid Haskell. I believe it has the potential to become a tool that is used to formally verify programs on a larger scale outside of academia. With her guidance, we have taken some of the first steps towards this goal. I have enjoyed working on many interesting problems with all of my collaborators including Yiyun Liu, Andrew Ruef, Michelle Mazurek, Daniel Votipka, Kelsey Fulton, Phúc Nguyễn, Piotr Mardziel, Dave Levin, Patrick Redmond, Lindsey Kuper, and Matthew Hou. I hope that we continue to work together in the future.

Finally, I would not be here without the love and support of my family. My parents built a strong foundation for me growing up. My brother and I foster a competitive environment that drives us to constantly improve. My wife has stood by my side and encouraged me throughout my journey through graduate school.

Table of Contents

Acknowledgements	ii
Table of Contents	iii
List of Tables	vi
List of Figures	vii
List of Abbreviations	ix
Chapter 1:Introduction	1
1.1 Background on Liquid Haskell	7
Chapter 2:Build It, Break It, Fix It: Contesting Secure Development	11
2.1 Introduction	11
2.1.1 Acknowledgements	14
2.2 Build-it, Break-it, Fix-it	15
2.2.1 Competition phases	16
2.2.2 Competition scoring	18
2.2.3 Discussion	23
2.2.4 Implementation	26
2.3 Contest Problems	29
2.3.1 Secure Log	29
2.3.2 ATM	33
2.3.3 Multiuser DB	37
2.4 Build-it Submissions: Successes and Failures	42
2.4.1 Failure Stories	43
2.4.2 Success Stories	46
2.5 Quantitative Analysis	47
2.5.1 Data collection	48
2.5.2 Analysis approach	49
2.5.3 Contestants	50
2.5.4 Ship scores	51
2.5.5 Resilience	57
2.5.6 Presence of security bugs	61
2.5.7 Breaking success	65

2.5.8	Model differences	72
2.5.9	Summary	73
2.6	Related work	74
2.7	Conclusion	78

Chapter 3:LWeb: Information Flow Security for Multi-tier Web Applications 80

3.1	Introduction	80
3.1.1	Acknowledgements	84
3.2	Overview	84
3.2.1	Label-Based Information Flow Control with LIO	85
3.2.2	Yesod	87
3.2.3	LWeb: Yesod with LIO	89
3.3	Mechanizing Noninterference of LIO in Liquid Haskell	93
3.3.1	Security Lattice as a Type Class	93
3.3.2	λ_{LIO} : Syntax and Semantics	95
3.3.3	Noninterference	99
3.4	Label-based Security for Database Operations	104
3.4.1	Database Definition	104
3.4.2	Querying the Database	106
3.4.3	Monadic Database Queries	108
3.4.4	Noninterference	113
3.5	Liquid Haskell for Metatheory	115
3.5.1	Advantages	116
3.5.2	Disadvantages	118
3.6	Implementation	119
3.6.1	Extensions	119
3.6.2	Trusted Computing Base	122
3.7	The BIBIFI Case Study	123
3.7.1	BIBIFI Labels	123
3.7.2	Users and Authentication	124
3.7.3	Opening the Contest	125
3.7.4	Teams and Declassification	126
3.7.5	Breaks and Advanced Queries	128
3.8	Experimental Evaluation	129
3.8.1	Trusted Computing Base of BIBIFI	129
3.8.2	Running Time Overhead	129
3.9	Quantifying Information Flow	131
3.10	Related Work	135
3.11	Conclusion	140

Chapter 4:Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell 146

4.1	Introduction	146
4.1.1	Acknowledgements	151

4.2	Typeclasses in Liquid Haskell	152
4.2.1	Refinement Types for Typeclasses	152
4.2.2	Verifying Laws of Standard Typeclass Instances	157
4.3	Implementing Typeclass Refinements	158
4.3.1	GHC Typeclass Elaboration	161
4.3.2	Interaction with GHC	163
4.3.3	Reasoning About Coherence	168
4.4	Case Study: Verified Replicated Data Types	172
4.4.1	Background: Conflict-free Replicated Data Types (CRDTs)	172
4.4.2	Verifying CRDTs with Typeclass Refinements	174
4.4.3	Proofs	181
4.4.4	Verifying CRDT Semantics	182
4.4.5	Applications	184
4.5	Related Work	187
4.5.1	Verification of Haskell’s Typeclass Laws	187
4.5.2	Type System Expressiveness vs. Coherence of Elaboration	189
4.5.3	Verifying Replicated Data Types	190
4.6	Conclusion	193
Chapter 5: Conclusion		195
5.1	Future Work	196

List of Tables

2.1	Example scoring results for a three-team contest with one optional test and one performance test ($M = 50$). Bugs S_1 and S_2 are security vulnerabilities; bug C is a correctness bug.	22
2.2	Contestants, by self-reported country.	51
2.3	Demographics of contestants from qualifying teams. Some participants declined to specify gender.	52
2.4	Factors and baselines for build-it models.	54
2.6	Break-it teams in each contest submitted bug reports, which were judged by the automated oracle. Build-it teams then submitted fixes, each of which could potentially address multiple bug reports.	57
2.7	Final logistic model measuring log-likelihood of the discovery of a security bug in a team’s submission. Nagelkerke $R^2 = 0.619$	62
2.8	The number and percentage of teams that had different types of security bugs by language category. Percentages are relative to total submissions in that language category, across all contests. Integrity, confidentiality, and availability bugs were not distinguished for the Multiuser DB problem during that contest. We group them in their own column.	64
2.9	Factors and baselines for break-it models.	66
2.10	Final linear regression model of teams’ break-it scores, indicating how many points each selected factor adds to the total score. $R^2 = 0.15$	67
2.11	Final linear regression modeling the count of security bugs found by each team. Coefficients indicate how many security bugs each factor adds to the count. $R^2 = 0.203$	70
3.1	Latency comparison between the Vanilla and LWeb implementations of the BIBIFI application. The mean, standard deviation, and tail latency in milliseconds over 1,000 trials are presented. In addition, the response size in kilobytes and the overhead of LWeb are shown.	130
4.1	Total lines of proofs for each typeclass instance and the average verification time in seconds. Each reported time covers the laws on its row and those on the following rows up to the next reported time.	160
4.3	Total lines of proofs for each typeclass instance and the average verification time in seconds. Verifications times for <code>lawCommut</code> and <code>lawCompatCommut</code> are combined.	183

List of Figures

1.1	Haskell’s list <code>head</code> and <code>append (++)</code> functions augmented with refinement types to capture pre- and post-conditions; and <code>lAssoc</code> , a statement and proof that <code>append</code> is associative.	8
2.1	Overview of BIBIFI’s implementation.	26
2.2	MITM replay attack.	35
2.3	Grammar for the Multiuser DB command language as BNF. Here, x and y represent arbitrary variables; p and q represent arbitrary principals; and s represents an arbitrary string. Commands submitted to the server should match the non-terminal <code><prog></code>	38
2.6	Final resilience scores, ordered by team, and plotted for each contest problem. Build-it teams who did not bother to fix bugs generally had lower scores.	60
2.8	Scores of break-it teams prior to the fix-it phase, broken down by points from security and correctness bugs. The final score of the break-it team (after fix-it phase) is noted as a dot. Note the different ranges in the y -axes. In general, the Secure Log contest had the least proportion of points coming from security breaks.	69
2.9	Count of security bugs found by each break-it team, organized by contest and whether the team also participated in build-it. The heavy vertical line in the box is the median, the boxes show the first and third quartiles, and the whiskers extend to the most outlying data within $\pm 1.5\times$ the interquartile range. Dots indicate further outliers.	71
3.1	Structure of <code>LWeb</code>	85
3.2	The <code>Label</code> class	85
3.3	Example <code>LWeb</code> database table definition. The green is Yesod syntax and the blue is the <code>LWeb</code> policy.	88
3.4	<code>Label</code> type class extended with <code>law*</code> methods to define the lattice laws as refinement types.	94
3.5	Syntax of λ_{LIO}	96
3.6	Operational semantics of λ_{LIO}	141
3.7	Simulation between <code>eval</code> and <code>eval</code>	142
3.8	Definition of λ_{LWeb} database	142
3.9	Extension of programs and terms with a database.	142
3.10	Evaluation of monadic database terms.	143

3.11	Erasure of programs and databases.	144
3.12	BIBIFI labels.	144
3.13	Basic BIBIFI <code>User</code> table.	144
3.14	Table <code>UserInfo</code> contains additional BIBIFI user information.	144
3.15	Definition of <code>Announcement</code> , <code>Team</code> , and <code>TeamMember</code> tables and their policies.	145
3.16	Definition of <code>BreakSubmission</code> table and its policy.	145
3.17	Throughput (req/s) of the <code>Vanilla</code> and <code>LWeb</code> versions of the BIBIFI application.	145
4.1	Typeclasses with Refinement Types	152
4.2	Typeclass definitions for <code>Functor</code> , <code>Applicative</code> , and <code>Monad</code> and their associated laws.	159
4.3	Changes to Liquid Haskell’s architecture.	165
4.4	Definition of the <code>VRDT</code> typeclass and its <code>Max</code> instance	175
4.5	Implementation of <code>TwoPMap</code>	178
4.6	Data type for a calendar event that is made up of <code>VRDTs</code>	180
4.7	Denotational semantics of <code>Multiset</code>	184

List of Abbreviations

AES	Advanced Encryption Standard
AIC	Akaike Information Criterion
API	Application Programming Interface
ATM	Automated Teller Machine
BIBIFI	Build-it Break-it Fix-it
CRDT	Conflict-free Replicated Data Types
CTF	Capture the Flag
CVE	Common Vulnerabilities and Exposures
DB	Database
DSL	Domain Specific Language
FRP	Functional Reactive Programming
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IFC	Information Flow Control
IO	Input Output
JSON	JavaScript Object Notation
LH	Liquid Haskell
LIO	Labeled IO
LSP	Language Server Protocol
MITM	Man in the middle
MOOC	Massive Open Online Courses
NSF	National Science Foundation
PKI	Public Key Infrastructure
PLE	Proof by Logical Evaluation
SEC	Strong Eventual Consistency
SMT	Satisfiability Modulo Theories
SQL	Structured Query Language

SSL Secure Sockets Layer
TCB Trusted Computing Base
VRDT Verified Replicated Data Types

Chapter 1: Introduction

Having a reliable means to build secure software has been a goal since at least the seminal work of Saltzer and Schroeder [1]. Unfortunately, software bugs and vulnerabilities are still rampant decades later. According to MITRE’s common vulnerabilities and exposures (CVE) database [2], tens of thousands of vulnerabilities are publicly disclosed annually. Insecure code has negative repercussions in many areas including finance, transportation, health, and defense. Failure to secure buggy programs can lead to service downtime, hardware failure, and compromised sensitive information. Such events have real world consequences, risking money, time, and even life. In this dissertation, we investigate factors that influence secure development practices and explore techniques for building correct and secure networked systems.

Programming language-based techniques have the potential to improve the quality of software. To investigate this potential, we ran the **Build-it, Break-it, Fix-it** (BIBIFI) contest as a quasi-controlled experiment (Chapter 2). We aimed to understand how programming language-based techniques and other factors influence the introduction of incorrect, vulnerable code. BIBIFI’s objective is to assess the ability to securely build software, not just break it.

There are three rounds in a BIBIFI contest. During the first round, called *Build-it*, small development teams are given a problem specification with security requirements, and the teams write software that satisfies the specification. The programs they submit are evaluated based on correctness, efficiency, and featurefulness. In the second, *Break-it*, round, teams are given the source code of other contestants and tasked with identifying bugs and vulnerabilities in the other teams' build-it code. Defects are demonstrated with test cases that are evaluated against the target implementation and an oracle implementation. Attacking teams gain break-it points for successful breaks, while target teams lose build-it points for breaks against their code. Build-it and break-it scores are independent, and top performers in each category receive prizes. In the last, *Fix-it*, round, builders can fix the bugs submitted against them, allowing them to recover some of the redundantly lost points when multiple breaker teams attacked the same bug.

BIBIFI's structure and scoring system aim to encourage meaningful outcomes, e.g., to ensure that top-scoring build-it teams really produce secure and efficient software. For example, break-it teams may submit a limited number of bug reports per build-it submission, and they will lose points during fix-it for test cases that expose the same underlying defect or a defect also identified by other teams. As such, they are encouraged to look for bugs broadly (in many submissions) and deeply (to uncover hard-to-find bugs).

We held three contests with 156 participating teams and three different problem specifications. Quantitative analysis on data from the contests unveiled statistically significant effects. For build-it scores: Writing code in C/C++ increased

build-it scores initially, but also increased chances of a security bug being found later; C/C++ programs were $11\times$ more likely to have a reported security bug than programs written in a statically type-safe language. Considering break-it scores: Larger teams found more bugs during the break-it phase, and teams that also qualified during the build-it phase found more security bugs than those that did not. Other trends emerged, but did not reach significance. Build-it teams with more developers tended to produce more secure implementations. More programming experience was helpful for breakers.

Evidence from the BIBIFI competition shows that properties of programming languages do impact the security of software. Supported by this result, we investigated more advanced language-based techniques that improve software correctness and security. Specifically, we used Liquid Haskell [3] to verify correctness and security properties about programs. Liquid Haskell is a relatively recent verification tool that extends the Haskell [4, 5] programming language with refinement types. There are many other language-based verifications tools like F* [6, 7], which also uses refinement types, while Coq [8, 9], Isabelle [10], Idris [11], Agda [12, 13], and Lean [14] are dependently-typed languages. Liquid Haskell is particular promising in being applicable to the real world programs since it verifies Haskell code, which is used both in research and industry. In addition, it provides proof automation for the user by leveraging SMT solvers and implementing a technique called proof by logical evaluation (PLE). A contribution of the work in this dissertation is as the first significant verification efforts in Liquid Haskell.

Language-based Information Flow Policy Enforcement with LWeb. We created **LWeb**, a tool that protects the confidentiality and integrity of data in modern web applications (Chapter 3). Traditional ad hoc enforcement mechanisms such as (manual) access control may fail to block illicit information flows between components, e.g., from database to server to client. **LWeb** guarantees that such flows cannot occur by dynamically and automatically enforcing information flow control (IFC) [15] policies.

LWeb is an extension of an existing Haskell language framework called **LIO** [16]. **LIO** works by dynamically tracking a *current label* that represents the security label of all values read during the current computation. Labels are lattice ordered (as is typical [17]), with the degenerate case being a secret (high) label and public (low) one. If data is ever written to a channel that is lower than the current label, **LIO** halts execution which prevents leaks of sensitive information. **LWeb** extends **LIO** with support for database transactions. Each table has a label that protects its length. In addition, each row may have a more refined policy to protect its contents. The label for a field in a row may be specified as a function of other fields in the same row. This allows, for example, having a row specifying a user and some sensitive user data; the former can act as a label to protect the latter.

We implemented **LWeb** as a Haskell library that integrates with the **Yesod** web programming framework. **LWeb** dynamically tracks the current label in a monad transformer called **LMonad** so that it can be layered on top of arbitrary monadic computations. To integrate with **Yesod**, we apply **LMonad** on top of the monad that handles HTTP requests and runs database transactions. Then we extended

`Yesod`'s database API to permit defining label-based information flow policies on user-defined database schemas. Finally, we insert IFC checks into SQL queries to ensure that information flow policies are enforced.

To guarantee that the IFC checks we perform for database transactions are correct, we formalize `LWeb` in Liquid Haskell and prove that it enjoys noninterference. This mechanization constitutes the largest-ever development in Liquid Haskell and is the first Liquid Haskell application to prove a language metatheory.

To evaluate `LWeb`, we integrated `LWeb` into the BIBIFI contest infrastructure as a case study. Porting the BIBIFI codebase to use `LWeb` reduced the trusted computing base of the entire application to just 80 lines of its code (1%) plus the `LWeb` codebase (for a total of 21%). `LWeb` imposes modest overhead on BIBIFI query latencies—experiments show between 2% and 21%.

Verifying the Correctness of Replicated Data Types. We would like to verify that the implementation of `LWeb` is correct, but doing so required further extensions to Liquid Haskell. To help develop these, we turned our attention to a different verification effort. The final advanced language-based technique we developed verifies the correctness of distributed applications based on *replicated data types* (RDTs) (Chapter 4). Replication is ubiquitous in distributed systems to guard against machine failures and keep data physically close to clients who need it, but it introduces the problem of keeping replicas consistent with one another in the face of network partitions and unpredictable message latency. RDTs are data structures whose operations must satisfy certain mathematical properties that can be leveraged to ensure

strong convergence [18], meaning that replicas are guaranteed to have equivalent state given that they have received and applied the same *unordered* set of update operations. Specifically, operations on RDTs must be commutative in order to converge.

It is natural to define RDTs using *typeclasses* [19], which are used extensively throughout the Haskell ecosystem. A typeclass *definition* specifies a type constructor and a collection of method declarations over that type. A typeclass *instance* defines an implementation of that constructor and those methods. This allows multiple data types, or instances, to provide a uniform, modular interface similar to traits in Rust or interfaces in Java. Unfortunately, Liquid Haskell cannot verify properties of typeclasses.

To remedy this situation, we extended Liquid Haskell with support for typeclasses. Liquid Haskell now has the ability to state properties about typeclasses using refinement types and prove that instances satisfy those properties.

With Liquid Haskell’s ability to reason about typeclasses, we define RDTs with a typeclass called `VRDT`. Refinement types on `VRDT` encode the necessary properties of RDTs including commutativity. We implement several primitive instances and prove that they satisfy the `VRDT` properties. We also defined several larger `VRDT` instances by modularly combining both the code and proofs of smaller ones. We state and prove, in Liquid Haskell, the strong convergence property that `VRDT` instances enjoy. Our `VRDT` instances are sufficiently expressive that with them we were able to build a shared calendar event planner, and also a collaborative text editor, though the latter relies on a `VRDT` we have not yet fully verified, but expect to. Because Liquid

Haskell is an extension of standard Haskell, our applications are real, running Haskell applications, but now with mechanically verified RDT implementations.

In this dissertation, we investigate the thesis that *programming language-based techniques can be used to improve the correctness and security of programs*. Results from the BIBIFI competition provide evidence that software written in statically-typed programming languages are less likely to have a security bug present. The **LWeb** tool uses dynamic enforcement to prevent illicit information flows in database-backed applications to preserve confidentiality and integrity. Furthermore, the mechanization of its meta-theory uncovered bugs in the original implementation, improving its reliability. To ensure that RDTs in distributed applications converge, we implement several **VRDT** instances and prove that they satisfy their required commutativity properties. To do so, we extend Liquid Haskell with the ability to augment typeclasses with refinement types and verify that instances satisfy the refinement types.

1.1 Background on Liquid Haskell

Liquid Haskell [3, 20] is an SMT-based refinement type checker for Haskell programs. Liquid Haskell permits refinement type specifications on Haskell source code. It converts the code into SMT queries to validate that the code satisfies the specifications.

A *refinement type* augments a base type T with a predicate ϕ that restricts the set of valid values [21, 22, 23]. In Liquid Haskell, a refinement type has the form

```

head :: {xs:[a] | length xs > 0} → a
head (h:_) = h

(++ ) :: xs:[a] → ys:[a] → { v:[a] | length v == length xs +
    length ys }
[]      ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)

lAssoc :: x:[a] → y:[a] → z:[a] → { x ++ (y ++ z) == (x ++ y) ++
    z }
lAssoc [] _ _ = ()
lAssoc (_:x) y z = lAssoc x y z

```

Figure 1.1: Haskell’s list `head` and append `(++)` functions augmented with refinement types to capture pre- and post-conditions; and `lAssoc`, a statement and proof that append is associative.

$\{ x:T \mid \phi \}$ —the base type T is refined according to predicate ϕ , which may refer to values of the base type via the variable x (if it appears free in ϕ). For example, a refinement type for positive integers would be $\{ x:\text{int} \mid x > 0 \}$, i.e., $\phi = x > 0$.

In Figure 1.1, the function `head` uses this kind of refinement on the type of its input list `xs`, stating the *precondition* that the list’s length be positive. This refinement thus prevents calling `head` with a possibly empty list, thus precluding the exception that could otherwise result.

Also in the figure we see code for Haskell’s standard list append operator, `(++)`, which uses a refinement to state a *postcondition*. The (standard) code states that appending an empty list `[]` with a list `ys` yields `ys` (line 2), while appending a non-empty list (with a head element `x` and a tail `xs`) with a list `ys` is the result of cons’ing `x` to the front of `xs` appended to `ys` (line 3). The refinement type states that the output list’s length is equal to the sum of the lengths of the input lists. The

refinement type predicate is able to refer to the function’s inputs via names `xs` and `ys`, which annotate the parameters’ types. Liquid Haskell proves that postconditions such the one on `(++)` hold by generating appropriate verification conditions from the code and delegating to an SMT solver (in particular, Z3 [24]); we say more on this below.

In the refinement types of `head` and `(++)`, `length` refers to the Haskell `length` function on lists. Such references to normal language terms are lifted into the refinement logic through a process called *refinement reflection* [25]. Refinement reflection uses the definitions of Haskell’s functions to generate singleton refinement types that precisely describe the result of the function. To ensure soundness of type checking, only provably terminating functions can be reflected.

Refinement reflection makes it possible to write and mechanically verify proofs of independent, general properties, e.g., involving many functions and not just a single one. These are called *extrinsic properties*, as they are written externally to any particular function’s definition, as opposed to *intrinsic properties* like the ones on `head` and `(++)`. For example, `lAssoc` in Figure 1.1 is an extrinsic property (and proof) that `append` is associative. The property is the type, which states that for all lists `x`, `y`, `z` we have that `x ++ (y ++ z)` equals `(x ++ y) ++ z`. Note that the postcondition of `lAssoc` is equivalent to $\{ v:\text{unit} \mid x ++ (y ++ z) == (x ++ y) ++ z \}$ —the `v:unit` part is dropped since there is no need to name the result, which is not mentioned in the predicate.

Proofs of extrinsic properties are themselves Liquid Haskell definitions whose type is the desired property. The proof in our example is the body of `lAssoc`,

which expresses that the property holds by induction—the base case is for $[]$ and the recursive case is for $(_:x) y z$. In the base case, there is nothing specific the programmer has to write other than $()$, the “return type” of the property. For the recursive case, the inductive argument occurs by referring to the property on the strictly smaller input $x y z$ (rather than $(_:x) y z$). This proof follows a standard formula [26] in which the handwritten part shown here provides the structure, and the proof details are filled in using a combination of *PLE* (Proof by Logical Evaluation) [25, 27], which automates function unfolding, and SMT solving, which automates reasoning over specific theories (e.g., equality and linear arithmetic). Both strategies preserve decidable type checking. Such automation helps make it possible to write substantial proofs in Liquid Haskell. Proofs can also be done by hand, as needed/desired [26].

Liquid Haskell’s implementation is simplified by making use of GHC, the Glasgow Haskell Compiler [28], to partially evaluate programs. Liquid Haskell first parses refinement types, which are written in the comments of normal Haskell code. Then it passes the Haskell code to GHC, and receives back the code as *Core*, which is GHC’s simplified intermediate language. Liquid Haskell lifts the Core output into the refinement logic using refinement reflection. Finally, it converts the refinement types and corresponding Core output into SMT-LIB2 queries [29] which can automatically be verified by Z3. If any queries are invalid, Liquid Haskell reports an error message to the user.

Chapter 2: Build It, Break It, Fix It: Contesting Secure Development

2.1 Introduction

Security competitions [30, 31, 32, 33, 34] are popular events that allow cybersecurity experts to demonstrate their abilities. These contests emphasize testing skills related to *breaking* systems (e.g., exploiting vulnerabilities or misconfigurations) and *mitigating* vulnerabilities (e.g., rapid patching or reconfiguration). They do not test participants' ability to *build* (i.e., design and implement) systems that are secure. In traditional programming competitions [35, 36, 37], contestants implement solutions to given problem specifications. Submissions are evaluated on correctness and performance, but not typically security. This existing landscape of security and programming competitions is unfortunate because at least as far back as Saltzer and Schroeder [1], experts have argued that security must be treated as a first-order design goal and cannot easily be added to an existing system. There is no a priori reason to assume that skilled breakers produce quality code [38] or that successful programmers build secure software.

Build-it, Break-it, Fix-it (BIBIFI) is a new security contest focused on

building secure systems. In BIBIFI, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teams' submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. Quantitative analysis from these contests found that the most efficient build-it submissions used C/C++, but submissions coded in a statically-typed language were less likely to have a security flaw. Break-it teams that were also successful build-it teams were significantly better at finding security bugs.

We held three BIBIFI contests during 2015 and 2016, involving three different programming problems. For the first contest, participants built a *secure, append-only log* of an art gallery's security system. Movement between rooms of the art gallery were recorded, allowing administrators to query the locations of staff and guests. Attackers that do not know a required "authentication token" must be prevented from reading or modifying the data contained in the log. In the second contest, contestants implemented a pair of *secure, communicating programs* that model an ATM client communicating with a bank server. Man in the middle (MITM) attackers must not be able to intercept personal information (e.g., bank account names or balances) or corrupt account balances.¹ The third contest required participants to build a *access-controlled, multiuser data server* that protects the data of users. Users

¹Such attacks are realistic in practice, as detailed in a 2018 analysis of actual ATMs [39].

are authenticated via password checking, and access control with delegation restricts how data is read and modified. All contests drew participants from a MOOC (Massive Online Open Courseware) course on cybersecurity. MOOC participants had an average of 10 years of programming experience and had just completed a four-course sequence including courses on secure software and cryptography. The second and third contests also involved graduate and undergraduate students with less experience and training. The first contest had 156 MOOC participants (comprising 68 teams). The second contest was composed of 122 MOOC participants (comprising 41 teams) and 23 student participants (comprising 7 teams). The last contest had 68 MOOC participants (comprising 25 teams) and 37 student participants (comprising 15 teams).

BIBIFI's design aims to minimize the manual effort of running a contest, helping it scale. During the first contest, one person was dedicated to running the contest full-time, and two people served as judges part-time. There were over one hundred participants in this contest, and they submitted over 20,000 test cases. Despite this, effort by organizers was minimal. They only needed to make sure the infrastructure was running smoothly and judge whether a few hundred submitted fixes addressed only a single conceptual defect. Other tasks were handled automatically or by contestants themselves.

To understand what influenced success or failure, we manually inspected build-it submissions and break-it test cases. Teams that performed well during build-it usually used third-party libraries—e.g., SSL, NaCL, and BouncyCastle—for cryptographic functionality. Team that produced vulnerable submissions made mis-

takes including failing to use cryptography, implementing cryptography incorrectly, not using enough randomness, and not authenticating messages. Successful break-it teams uncovered ingenious techniques for exploiting vulnerabilities. For instance, some teams took advantage of side channels in their MITMs to uncover secret information.

This work makes two main contributions. First, it presents BIBIFI, a secure programming contest that encourages building, not just breaking software. Second, it offers details on three BIBIFI contests, including insights into successes and failures of participants and a quantitative analysis on data from the contests. The chapter is organized as follows. We present the design of BIBIFI in §2.2 and describe specifics of the contests we ran in §2.3. We present success and failure stories of contestants in §2.4 and a quantitative analysis of the data we collected in §2.5. We review related work in §2.6 and conclude in §2.7. Information, data, and opportunities to participate are available at <https://builditbreakit.org> and the BIBIFI codebase is at <https://github.com/plum-umd/bibifi-code>.

2.1.1 Acknowledgements

Most of the work presented in this chapter previously appeared in peer-reviewed publications [40, 41, 42, 43] and has been a collaboration between Andrew Ruef, Michael Hicks, Michelle Mazurek, Dave Levin, Daniel Votipka, Kelsey Fulton, Matthew Hou, Piotr Mardziel, and Phúc Nguyễn. Andrew initially came up with the idea for the contest, and Andrew, Mike, and I flushed out the design of the contest. I

implemented the majority of the contest infrastructure. Phúc helped rewrite the latest version of the codebase to combine the Break-it and Fix-it rounds.

We thank Jandelyn Plane and Atif Memon who contributed to the initial development of BIBIFI and its preliminary data analysis. Many people in the security community, too numerous to list, contributed ideas and thoughts about BIBIFI during its development—thank you! Manuel Benz, Martin Mory, Luke Valenta, Matt Blaze, Philip Ritchey, Aymeric Fromherz, Lujó Bauer, and Bryan Parno ran the contest at their universities and tested the infrastructure. Bobby Bhattacharjee and anonymous reviewers provided helpful comments on drafts of this work. This project was supported with gifts from Accenture, AT&T, Booz Allen Hamilton, Galois, Leidos, Patriot Technologies, NCC Group, Trail of Bits, Synopsis, ASTech Consulting, Cigital, SuprTek, Cyberpoint, and Lockheed Martin; by a 2016 Google Faculty Research Award; by grants from the NSF under awards EDU-1319147 and CNS-1801545; by DARPA under contract FA8750-15-2-0104; and by the U.S. Department of Commerce, National Institute for Standards and Technology, under Cooperative Agreement 70NANB15H330.

2.2 Build-it, Break-it, Fix-it

This section describes the goals, design, and implementation of the BIBIFI competition. At the highest level, our aim is to create an environment that closely reflects real-world development goals and constraints, and to encourage build-it teams to write the most secure code they can, and break-it teams to perform the

most thorough, creative analysis of others' code they can. We achieve this through a careful design of how the competition is run and how various acts are scored (or penalized). We also aim to minimize the manual work required of the organizers—to allow the contest to scale—by employing automation and proper participant incentives.

2.2.1 Competition phases

We begin by describing the high-level mechanics of a BIBIFI competition. BIBIFI may be administered online, rather than on-site, so teams may be geographically distributed. The contest comprises three phases, each of which last about two weeks for the contests we describe in this work.

BIBIFI begins with the **build-it phase**. Registered contestants aim to implement the target software system according to a published specification created by the contest organizers. A suitable target is one that can be completed by good programmers in a short time (just about two weeks, for the contests we ran), is easily benchmarked for performance, and has an interesting attack surface. The software should have specific security goals—e.g., protecting private information or communications—which could be compromised by poor design and/or implementation. The software should also not be too similar to existing software to ensure that contestants do the coding themselves (while still taking advantage of high-quality libraries and frameworks to the extent possible). The software must build and run on a standard Linux VM made available prior to the start of the contest. Teams

must develop using Git [44]; with each push, the contest infrastructure downloads the submission, builds it, tests it (for correctness and performance), and updates the scoreboard. §2.3 describes the three target problems we developed: (1) an append-only log (aka, Secure Log), (2) a pair of communicating programs that simulate a bank and ATM (aka, ATM), and (3) a multi-user data server with custom access control policies (aka, Multiuser DB).

The next phase is the **break-it phase**. Break-it teams can download, build, and inspect all qualifying build-it submissions, including source code; to qualify, the submission must build properly, pass all correctness tests, and not be purposely obfuscated (accusations of obfuscation are manually judged by the contest organizers). We randomize each break-it team’s view of the build-it teams’ submissions, but organize them by meta-data, such as programming language used. (Randomization aims to encourage equal scrutiny of submissions by discouraging break-it teams from investigating projects in the same order.) When they think they have found a defect, breakers submit a test case that exposes the defect and an explanation of the issue. We impose an upper bound on the number of test cases a break-it team can submit against a single build-it submission, to encourage teams to look at many submissions. BIBIFI’s infrastructure automatically judges whether a submitted test case truly reveals a defect. For example, for a correctness bug, it will run the test against a reference implementation (“the oracle”) and the targeted submission, and only if the test passes on the former but fails on the latter will it be accepted. Teams can also earn points by reporting bugs in the oracle, i.e., where its behavior contradicts the written specification; these reports are judged by the organizers. More

points are awarded to clear security problems, which may be demonstrated using alternative test formats. The auto-judgment approaches we developed for the three different contest problems are described in §2.3.

The final phase is the **fix-it phase**. Build-it teams are provided with the bug reports and test cases implicating their submission. They may fix flaws these test cases identify; if a single fix corrects more than one failing test case, the test cases are “morally the same,” and thus points are only deducted for one of them. The organizers determine, based on information provided by the build-it teams and other assessment, whether a submitted fix is “atomic” in the sense that it corrects only one conceptual flaw; if not, the fix is rejected.

Once the final phase concludes, prizes are awarded to the builders and breakers with the best scores, as determined by the scoring system described next.

2.2.2 Competition scoring

BIBIFI’s scoring system aims to encourage the contest’s basic goals, which are that the winners of the build-it phase truly produced the highest quality software, and that the winners of the break-it phase performed the most thorough, effective analysis of others’ code. The scoring rules, and the fact that scores are published in real time while the contest takes place, create incentives for good behavior (and disincentives for bad behavior).

2.2.2.1 Build-it scores

To reflect real-world development concerns, the winning build-it team would ideally develop software that is correct, secure, featureful, and efficient. While security is of primary interest to our contest, developers in practice must balance other aspects of quality against security [45, 46], creating trade-offs that cannot be ignored if we wish to motivate realistic developer decision-making.

As such, each build-it team’s score is the sum of the *ship* score² and the *resilience* score. The ship score is composed of points gained for correctness tests and performance tests. Each mandatory correctness test is worth M points, for some constant M , while each optional correctness test is worth $M/2$ points. Each performance test has a numeric measure depending on the specific nature of the programming project—e.g., latency, space consumed, files left unprocessed—where lower measures are better. A test’s worth is $M \cdot (worst - v)/(worst - best)$, where v is the measured result, $best$ is the measure for the best-performing submission, and $worst$ is the worst performing. As such, each performance test’s value ranges from 0 to M . As scores are published in real time, teams can see whether they are scoring better than other participants. Their relative standing may motivate them to improve their implementation to improve its score before the build-it phase ends.

The resilience score is determined after the break-it and fix-it phases, at which point the set of unique defects against a submission is known. For each *unique* bug found against a team’s submission we subtract P points from its resilience score;

²The name is meant to evoke a quality measure at the time software is shipped.

as such, the best possible resilience score is 0. For correctness bugs, we set P to $M/2$; for crashes that violate memory safety we set P to M , and for exploits and other security property failures we set P to $2M$. (We discuss the rationale for these choices below.) Once again, real-time scoring helps incentivize fixing, to get points back.

2.2.2.2 Break-it scores

Our primary goal with break-it teams is to encourage them to find as many defects as possible in the submitted software, as this would give greater confidence in our assessment that one build-it team's software is of higher quality than another's. While we are particularly interested in obvious security defects, correctness defects are also important, as they can have non-obvious security implications.

A break-it team's score is the summed value of all defects they have found, using the above P valuations. This score is shown in real time during the break-it phase, incentivizing teams to improve their standing. After the fix-it phase, this score is reduced. In particular, if a break-it team submitted multiple test cases against a project that identify the same defect, the duplicates are discounted. Moreover, each of the N break-it teams' scores that identified the same defect are adjusted to receive P/N points for that defect, splitting the P points among them.

Through a combination of requiring concrete test cases and scoring, BIBIFI encourages break-it teams to follow the spirit of the competition. First, by requiring them to provide test cases as evidence of a defect or vulnerability, we ensure they

are providing useful bug reports. By providing $4\times$ more points for security-relevant bugs than for correctness bugs, we nudge break-it teams to look for these sorts of flaws and to not just focus on correctness issues. (But a different ratio might work better; see below.) Because break-it teams are limited to a fixed number of test cases per submission, they are discouraged from submitting many tests they suspect are “morally the same;” as they could lose points for them during the fix-it phase they are better off submitting tests demonstrating different bugs. Limiting per-submission test cases also encourages examining many submissions. Finally, because points for defects found by other teams are shared, break-it teams are encouraged to look for hard-to-find bugs, rather than just low-hanging fruit.

Scoring example. Table 2.1 presents an example scoreboard for a three-team contest. For simplicity, there is only one optional test and one performance test. We set M to be 50 points.

Consider the *ship score*. All teams receive 250 points for passing all correctness tests. Team 2 receives 25 additional points for implementing the optional test. Team 1 receives 50 additional points for having the fastest performance test; team 2 gets 40 points for being relatively 20% slower; and team 3 receives no performance points since their implementation is slowest. Now consider *resilience score*. This is a team’s ship score minus points lost for each unique bug against its implementation. Each team has one bug against it, where bugs S_1 and S_2 are security vulnerabilities (100 points), while bug C_1 is a correctness bug (25 points). Finally, a team’s *break score* is the number of points it receives for discovering bugs, split between teams that

Team	Correctness Tests	Optional Test	Performance Test Runtime (s)	Ship Score	Bugs Against	Resilience Score	Bugs Reported	Break Score
Team 1	Pass	Fail	4	300	S_1	200	S_2	100
Team 2	Pass	Pass	5	315	C_1	290	S_1	50
Team 3	Pass	Fail	9	250	S_2	150	S_1, C_1	75

Table 2.1: Example scoring results for a three-team contest with one optional test and one performance test ($M = 50$). Bugs S_1 and S_2 are security vulnerabilities; bug C is a correctness bug.

discover the same bug. Teams 2 and 3 split the 100 points for discovering exploit S_1 against team 1.

2.2.2.3 Discouraging collusion

BIBIFI contestants may form teams however they wish, and may participate remotely. This encourages wider participation, but it also opens the possibility of collusion between teams, as there cannot be a judge overseeing their communication and coordination. There are three broad possibilities for collusion, each of which BIBIFI’s scoring discourages.

First, two break-it teams could consider sharing bugs they find with one another. By scaling the points each finder of a particular bug obtains, we remove incentive for them to both submit the same bugs, as they would risk diluting how many points they both obtain.

The second class of collusion is between a build-it team and a break-it team, but neither have incentive to assist one another. The zero-sum nature of the scoring between breakers and builders places them at odds with one another; revealing a bug to a break-it team hurts the builder, and not reporting a bug hurts the breaker.

Finally, two build-it teams could collude, for instance by sharing code with one another. It might be in their interests to do this in the event that the competition offers prizes to two or more build-it teams, since collusion could obtain more than one prize-position. We use judging and automated tools (and feedback from break-it teams) to detect if two teams share the same code (and disqualify them), but it is not clear how to detect whether two teams provided out-of-band feedback to one another prior to submitting code (e.g., by holding their own informal “break-it” and “fix-it” stages). We view this as a minor threat to validity; at the surface, such assistance appears unfair, but it is not clear that it is contrary to the goals of the contest, that is, to developing secure code.

2.2.3 Discussion

The contest’s design also aims to enable scalability by reducing work on contest organizers. In our experience, BIBIFT’s design succeeds at what it sets out to achieve, but has limitations.

Minimizing manual effort. Once the contest begins, manual effort by the organizers is limited by design. All bug reports submitted during the break-it phase are automatically judged by the oracle; organizers only need to vet any bug reports against the oracle itself. Organizers may also need to judge accusations by breakers of code obfuscation by builders. Finally, organizers must judge whether submitted fixes address a single defect; this is the most time consuming task. It is necessary because we cannot automatically determine whether multiple bug reports against one team

map to the same software defect; techniques for automatic testcase deduplication are still a matter of research (see §2.6). As such, we incentivize build-it teams to demonstrate overlap through fixes, which organizers manually confirm address only a single defect, not several.

Previewing some of the results presented later, we can confirm that the design works reasonably well. For example, as detailed in Table 2.6, 68 teams submitted 24,796 test cases for the Secure Log contest. The oracle auto-rejected 15,314 of these, and build-it teams addressed 2,252 of those remaining with 375 fixes, a 6× reduction. Most confirmations that a fix addresses a single bug took 1-2 minutes. Only 30 of these fixes were rejected. No accusations of code obfuscation were made by break-it teams, and few bug reports were submitted against the oracle. All told, the Secure Log contest was successfully managed by one full-time person, with two others helping with judging.

Limitations. While we believe BIBIFI’s structural and scoring incentives are properly designed, we should emphasize several limitations.

First, there is no guarantee that all implementation defects will be found. Break-it teams may lack the time or skill to find problems in all submissions, and not all submissions may receive equal scrutiny. Break-it teams may also act contrary to incentives and focus on easy-to-find and/or duplicated bugs, rather than the harder and/or unique ones. In addition, certain vulnerabilities, like insufficient randomness in key generation, may take more effort to exploit, so breakers may skip such vulnerabilities. Finally, break-it teams may find defects that the BIBIFI infras-

structure cannot automatically validate, meaning those defects will go unreported. However, with a large enough pool of break-it teams, and sufficiently general defect validations automation, we still anticipate good coverage both in breadth and depth.

Second, builders may fail to fix bugs in a manner that is in their best interests. For example, in not wanting to have a fix rejected as addressing more than one conceptual defect, teams may use several specific fixes when a more general fix would have been allowed. Additionally, teams that are out of contention for prizes may simply not participate in the fix-it phase.³ We observed these behaviors in our contests. Both actions decrease a team’s resilience score (and correspondingly increase breakers’ scores). For our most recent contest, we attempted to create an incentive to fix bugs by offering prizes to participants that scale with their final score, rather than offering prizes only to winners. Unfortunately, this change in prize structure did not increase fix-it participation. We discuss fix-it behavior in more depth in §2.5.5.

Finally, there are several design points in a problem’s definition and testing code that may skew results. For example, too few correctness tests may leave too many correctness bugs to be found during break-it, distracting break-it teams’ attention from security issues. Too many correctness tests may leave too few bugs, meaning teams are differentiated insufficiently by general bug-finding ability. Scoring prioritizes security problems 4 to 1 over correctness problems, but it is hard to say what ratio makes the most sense when trying to maximize real-world outcomes;

³Hiding scores during the contest might help mitigate this, but would harm incentives during break-it to go after submissions with no bugs reported against them.

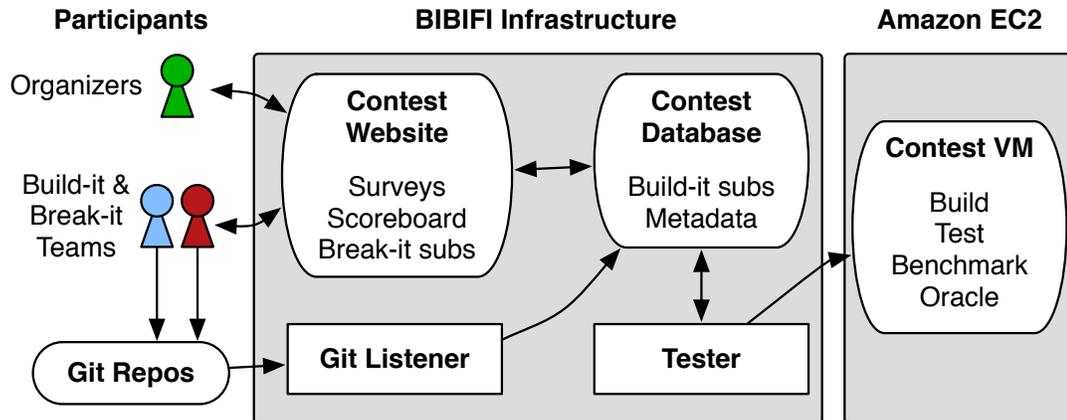


Figure 2.1: Overview of BIBIFI’s implementation.

both higher and lower ratios could be argued. How security bugs are classified will also affect behavior; two of our contests had strong limits on the number of possible security bugs per project, while the third’s definition was far more (in fact, too) liberal, as discussed in §2.5.6. Finally, performance tests may fail to expose important design tradeoffs (e.g., space vs. time), affecting the ways that teams approach maximizing their ship scores. For the contests we report in this work, we are fairly comfortable with these design points. In particular, our pilot contest [47] prioritized security bugs 2 to 1 and had fewer interesting performance tests, and outcomes were better when we increased the ratio.

2.2.4 Implementation

Figure 2.1 provides an overview of the BIBIFI implementation. It consists of a web frontend, providing the interface to both participants and organizers, and a backend for testing builds and breaks. Key goals of the infrastructure are security—we do not want participants to succeed by hacking BIBIFI itself—and scalability.

Web frontend. Contestants sign up for the contest through our web application frontend, and fill out a survey when doing so, to gather demographic data potentially relevant to the contest outcome (e.g., programming experience and security training). During the contest, the web application tests build-it submissions and break-it bug reports, keeps the current scores updated, and provides a workbench for the judges for considering whether or not a submitted fix covers one bug or not.

To secure the web application against unscrupulous participants, we implemented it in ~ 11500 lines of Haskell using the Yesod [48] web framework backed by a PostgreSQL [49] database. Haskell’s strong type system defends against use-after-free, buffer overrun, and other memory safety-based attacks. The use of Yesod adds further automatic protection against various attacks like CSRF, XSS, and SQL injection. As one further layer of defense, the web application incorporates the information flow control framework LWeb, which is derived from LIO [16], in order to protect against inadvertent information leaks and privilege escalations. LWeb dynamically guarantees that users can only access their own information, as established by a mechanized proof of correctness (in Liquid Haskell [50]).

Testing backend. The backend infrastructure is used for testing during the build-it phase for correctness and performance, and during the break-it phase to assess potential vulnerabilities. It consists of ~ 5500 lines of Haskell code (and a little Python).

To automate testing, we require contestants to specify a URL to a Git [44] repository (hosted on either Gitlab, Github, or Bitbucket) and shared with a desig-

nated `bibifi` username, read-only. The backend “listens” to each contestant repository for pushes, upon which it downloads and archives each commit. Testing is then handled by a scheduler that spins up a (Docker or Amazon EC2) virtual machine which builds and tests each submission. We require that teams’ code builds and runs, without any network access, in an Ubuntu Linux VM that we share in advance. Teams can request that we install additional open-source packages not present on the VM. The use of VMs supports both scalability (Amazon EC2 instances are dynamically provisioned) and security—using fresh VM instances prevents a team from affecting the results of future tests, or of tests on other teams’ submissions.

All qualifying build-it submissions may be downloaded by break-it teams at the start of the break-it phase. As break-it teams identify bugs, they prepare a (JSON-based) file specifying the buggy submission along with a sequence of commands with expected outputs that demonstrate the bug. Break-it teams commit and push this file (to their Git repository). The backend uses the file to set up a test of the implicated submission to see if it indeed is a bug.

The code that tests build and break submissions differs for each contest problem. To increase modularity, we have created a problem API so that testing code run on the VM can easily be swapped out for different contest problems. Contest organizers can create their own problems by conforming to this API. The infrastructure will set up the VM and provide submission information to the problem’s testing software via JSON. The problem’s software runs the submission and outputs the result as JSON, which the infrastructure records and uses to update scores accordingly. Details are available in the documentation of the contest repository.

2.3 Contest Problems

This section presents the three programming problems we have developed for BIBIFI contests. These three problems were used during open competitions in 2015 and 2016, and in our own and others' undergraduate security courses since then. We discuss each problem and its specific notions of security defect, as well as how breaks exploiting such defects are automatically judged.

2.3.1 Secure Log

The Secure Log problem was motivated as support for an art gallery security system. Contestants write two programs. The first, `logappend`, appends events to the log; these events indicate when employees and visitors enter and exit gallery rooms. The second, `logread`, queries the log about past events. To qualify, submissions must implement two basic queries (involving the current state of the gallery and the movements of particular individuals), but they could implement two more for extra points (involving time spent in the museum, and intersections among different individuals' histories). An empty log is created by `logappend` with a given authentication token, and later calls to `logappend` and `logread` on the same log must use that token or the requests will be denied.

Here is a basic example of invocations to `logappend`. The first command creates a log file called `logfile` because one does not yet exist, and protects it with the authentication token `secret`. In addition, it records that `Fred` entered the art gallery. Subsequent executions of `logappend` record the events of `Jill` entering the

gallery and both guests entering room 1.

```
$ ./logappend -K secret -A -G Fred logfile
$ ./logappend -K secret -A -G Jill logfile
$ ./logappend -K secret -A -G Fred -R 1 logfile
$ ./logappend -K secret -A -G Jill -R 1 logfile
```

Here is an example of `logread`, using the logfile just created. It queries who is in the gallery and what rooms they are currently in.

```
$ ./logread -K secret -S logfile
Fred
Jill
1: Fred,Jill
```

The problem states that an attacker is allowed direct access to the logfile, and yet integrity and privacy must be maintained. A canonical way of implementing the log is therefore to treat the authentication token as a symmetric key for authenticated encryption, e.g., using a combination of AES and HMAC. There are several tempting shortcuts that we anticipated build-it teams would take (and that break-it teams would exploit). For instance, one may be tempted to encrypt and sign individual log records as opposed to the entire log, thereby making `logappend` faster. But this could permit integrity breaks that duplicate or reorder log records. Teams may also be tempted to implement their own encryption rather than use existing libraries, or to simply sidestep encryption altogether. §2.4 reports several cases we observed.

A submission's performance is measured in terms of time to perform a particular sequence of operations, and space consumed by the resulting log. Correctness (and *crash*) bug reports are defined as sequences of `logread` and/or `logappend` operations with expected outputs (vetted by the oracle). Security is defined by *privacy*

and *integrity*: any attempt to learn something about the log’s contents, or to change them, without the using `logread` and `logappend` and the proper token should be disallowed. How violations of these properties are specified and tested is described next.

Privacy breaks. When providing a build-it submission to the break-it teams, we also included a set of log files that were generated using a sequence of invocations of that submission’s `logappend` program. We generated different logs for different build-it submissions, using a distinct command sequence and authentication token for each. All logs were distributed to break-it teams without the authentication token; some were distributed without revealing the sequence of commands (the “transcript”) that generated them. For these, a break-it team could submit a test case involving a call to `logread` (with the authentication token omitted) that queries the file. The BIBIFI infrastructure would run the query on the specified file with the authentication token, and if the output matched that specified by the breaker, then a privacy violation is confirmed. For example, before the break-it round, the infrastructure would run a bunch of randomly generated commands against a given team’s implementation.

```
$ ./logappend -K secret -A -G Fred logfile
$ ./logappend -K secret -A -G Fred -R 816706605 logfile
```

Breakers are only given the `logfile` and not the secret token or the transcript of the commands (for privacy breaks). A breaker would demonstrate a privacy violation by submitting the following string, which matches the invocation of `./logread -K`

```
secret -S logfile.
```

```
Fred  
816706605: Fred
```

The system knows the breaker has successfully broken privacy since the breaker is able to present confidential information without knowing the secret token. In practice, the transcript of commands is significantly longer and a random secret is used.

Integrity breaks. For about half of the generated log files we also provided the transcript of the `logappend` operations used to generate the file. A team could submit a test case specifying the name of the log file, the contents of a corrupted version of that file, and a `logread` query over it (without the authentication token). For both the specified log file and the corrupted one, the BIBIFI infrastructure would run the query using the correct authentication token. An integrity violation is detected if the query command produces a non-error answer for the corrupted log that differs from the correct answer (which can be confirmed against the transcript using the oracle).

This approach to determining privacy and integrity breaks has the benefit and drawback that it does not reveal the *source* of the issue, only that there is (at least) one, and that it is exploitable. As such, we only count up to one integrity break and one privacy break against the score of each build-it submission, even if there are multiple defects that could be exploited to produce privacy/integrity violations (since we could not automatically tell them apart).

2.3.2 ATM

The ATM problem asks builders to construct two communicating programs: `atm` acts as an ATM client, allowing customers to set up an account, and deposit and withdraw money; `bank` is a server that tracks client bank balances and processes their requests, received via TCP/IP. `atm` and `bank` should only permit a customer with a correct *card file* to learn or modify the balance of their account, and only in an appropriate way (e.g., they may not withdraw more money than they have). In addition, `atm` and `bank` should only communicate if they can authenticate each other. They can use an *auth file* for this purpose; it will be shared between the two via a trusted channel unavailable to the attacker.⁴ Since the `atm` is communicating with `bank` over the network, a “man in the middle” (MITM) could observe and modify exchanged messages, or insert new messages. The MITM could try to compromise security despite not having access to auth or card files. Such compromise scenarios are realistic, even in 2018 [39].

Here is an example run of the `bank` and `atm` programs.

```
$ ./bank -s bank.auth &
```

This invocation starts the `bank` server, which creates the file `bank.auth`. This file will be used by the `atm` client to authenticate the `bank`. The `atm` is started as follows:

```
$ ./atm -s bank.auth -c bob.card -a bob -n 1000.00  
{"account":"bob","initial_balance":1000}
```

The client initiates creation of a new account for user `bob` with an initial balance

⁴In a real deployment, this might be done by “burning” the auth file into the ATM’s ROM prior to installing it.

of \$1,000. It also creates a file `bob.card` that is used to authenticate `bob` (this is basically Bob's PIN) from here on. A receipt of the transaction from the server is printed as JSON. The `atm` client can now use `bob`'s card to perform further actions on his account. For example, this command withdraws \$63.10 from `bob`'s account:

```
$ ./atm -s bank.auth -c bob.card -a bob -w 63.10
{"account":"bob","withdraw":63.1}
```

A canonical way of implementing the `atm` and `bank` programs would be to use public key-based authenticated and encrypted communications. The `auth` file is used as the `bank`'s public key to ensure that key negotiation initiated by the `atm` is with the `bank` and not a MITM. When creating an account, the card file should be a suitably large random number, so that the MITM is unable to feasibly predict it. It is also necessary to protect against replay attacks by using nonces or similar mechanisms. As with Secure Log, a wise approach would be use a library like OpenSSL to implement these features. Both good and bad implementations are discussed further in §2.4.

Build-it submissions' performance is measured as the time to complete a series of benchmarks involving various `atm/bank` interactions.⁵ Correctness (and *crash*) bug reports are defined as sequences of `atm` commands where the targeted submission produces different outputs than the oracle (or crashes). Security defects are specified as follows.

⁵The transcript of interactions is always serial, so there was no motivation to support parallelism for higher throughput.

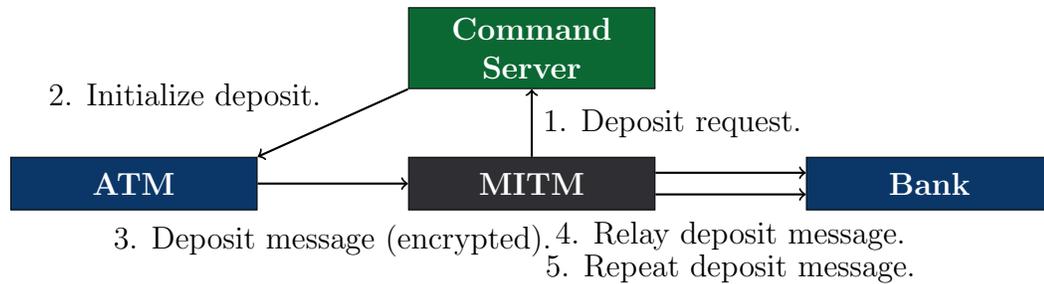


Figure 2.2: MITM replay attack.

Integrity breaks. Integrity violations are demonstrated using a custom MITM program that acts as a proxy: It listens on a specified IP address and TCP port, and accepts a connection from the `atm` while connecting to the `bank`. We provided a Python-based proxy as a starter MITM; it forwards communications between the endpoints after establishing the connection. A breaker's MITM would modify this baseline behavior, observing and/or modifying messages sent between `atm` and `bank`, and perhaps dropping messages or initiating its own.

To demonstrate an integrity violation, the MITM will send requests to a *command server*. It can tell the server to run inputs on the `atm` and it can ask for the card file for any account whose creation it initiated. Eventually the MITM will declare the test complete. At this point, the same set of `atm` commands is run using the oracle's `atm` and `bank` *without the MITM*. This means that any messages that the MITM sends directly to the target submission's `atm` or `bank` will not be replayed/sent to the oracle. If the oracle and target both complete the command list without error, but they differ on the outputs of one or more commands, or on the balances of accounts at the bank whose card files were not revealed to the MITM during the test, then there is evidence of an integrity violation.

As an example (based on a real attack we observed), consider a submission that uses deterministic encryption without nonces in messages. The MITM could direct the command server to deposit money from an account, and then replay the message it observes. When run on the vulnerable submission, this would credit the account twice. But when run on the oracle without the MITM, no message is replayed, leading to differing final account balances. A correct submission would reject the replayed message, which would invalidate the break. This example is illustrated in Figure 2.2.

Privacy breaks. Privacy violations are also demonstrated using a MITM. In this case, at the start of a test the command server will generate two random values, *amount* and *account name*. If by the end of the test no errors have occurred and the attacker can prove it knows the actual value of either secret (by sending a command that specifies it), the break is considered successful. Before demonstrating knowledge of the secret, the MITM can send commands to the and server with a *symbolic* reference to *amount* and *account name*; the server will fill in the actual secrets before forwarding these messages. The command server does not automatically create a secret account or an account with a secret balance; it is up to the breaker to do that (referencing the secrets symbolically when doing so).

As an example, suppose the target does not encrypt exchanged messages. Then a privacy attack might be for the MITM to direct the command server to create an account whose balance contains the secret amount. Then the MITM can observe an unencrypted message sent from `atm` to `bank`; this message will contain the actual

amount filled in by the command server. The MITM can then send its guess to the command server showing that it knows the amount.

As with the Secure Log problem, we cannot tell whether an integrity or privacy test is exploiting the same underlying weakness in a submission, so we only accept one violation of each category against each submission.

Timeouts and denial of service. One difficulty with our use of a breaker-provided MITM is that we cannot reliably detect bugs in `atm` or `bank` implementations that would result in infinite loops, missed messages, or corrupted messages. This is because such bugs can be simulated by the MITM by dropping or corrupting messages it receives. Since the builders are free to implement any protocol they like, our auto-testing infrastructure cannot tell if a protocol error or timeout is due to a bug in the target or due to misbehavior of the MITM. As such, we conservatively disallow any MITM test run that results in the target `atm` or `bank` hanging (timing out) or returning with a protocol error (e.g., due to a corrupted packet). This means that flaws in builder implementations might exist but evidence of those bugs might not be realizable in our testing system.

2.3.3 Multiuser DB

The Multiuser DB problem requires builders to implement a server that maintains a multi-user key-value store, where users' data is protected by customizable access control policies. The data server accepts queries written in a text-based command language delivered over TCP/IP (we assume the communication channel is

```

<prog>      ::= as principal p password s do \n <cmd> ***
<cmd>       ::= exit \n | return <expr> \n | <prim_cmd> \n <cmd>
<expr>      ::= <value> | [] | <fieldvals>
<fieldvals> ::= x = <value> | x = <value> , <fieldvals>
<value>     ::= x | x . y | s
<prim_cmd> ::= create principal p s
              | change password p s
              | set x = <expr>
              | append to x with <expr>
              | local x = <expr>
              | foreach y in x replacewith <expr>
              | set delegation <tgt> q <right> -> p
              | delete delegation <tgt> q <right> -> p
              | default delegator = p
<tgt>       ::= all | x
<right>     ::= read | write | append | delegate

```

Figure 2.3: Grammar for the Multiuser DB command language as BNF. Here, x and y represent arbitrary variables; p and q represent arbitrary principals; and s represents an arbitrary string. Commands submitted to the server should match the non-terminal `<prog>`.

trusted, for simplicity). Each program begins by indicating the querying user, authenticated with a password. It then runs a sequence of commands to read and write data stored by the server, where data values can be strings, lists, or records. The full grammar of the command language is shown in Figure 2.3 where the start symbol (corresponding to a client command) is `<prog>`. Accesses to data may be denied if the authenticated user lacks the necessary permission. A user can delegate permissions, like reading and writing variables, to other principals. If running the command program results in a security violations or error then all of its effects will be rolled back.

Here is an example run of the data server. To start, the server is launched and listens on TCP port 1024.

```
$ ./server 1024 &
```

Next, the client submits the following program.

```
as principal admin password "admin" do
  create principal alice "alices_password"
  set msg = "Hi Alice. Good luck!"
  set delegation msg admin read -> alice
  return "success"
***
```

The program starts by declaring it is running on behalf of principal `admin`, whose password is `"admin"`. If authentication is successful, the program creates a new principal `alice`, creates a new variable `msg` containing a string, and then delegates read permission on the variable to Alice. The server sends back a transcript of the successful commands to the client, in JSON format:

```
{"status": "CREATE_PRINCIPAL"}
{"status": "SET"}
{"status": "SET_DELEGATION"}
{"status": "RETURNING", "output": "success"}
```

Next, suppose Alice sends the following program, which simply logs in and reads the `msg` variable:

```
as principal alice password "alices_password" do
  return msg
***
```

The server response indicates the result:

```
{"status": "RETURNING", "output": "Hi Alice. Good luck!"}
```

The data server is implemented by writing a parser to read the input command programs. The server needs a store to keep the value of each variable as well as an access control list that tracks the permissions for the variables. It also needs to keep track of delegated permissions, which can form chains; e.g., Alice could delegate

read permission to all of her variables to Bob, who could then delegate permission to read one of those variables to Charlie. If when executing the program a security violation or other error occurs (e.g., reading a variable that doesn't exist), the server needs to roll back its state to what it was prior to processing the input program. All responses back to the client are encoded as JSON.

Scoring. A data server's performance is measured in elapsed runtime to process sequences of programs. Correctness (and *crash*) violations are demonstrated by providing a sequence of command programs where the data server's output differs from that of the oracle (or the implementation crashes). Security violations can be to data privacy, integrity, or availability, by comparing the behavior of the target against that of an oracle implementation.

Privacy breaks. A privacy violation occurs when the oracle would deny a request to read a variable, but the target implementation allows it. Consider the following example where a variable, `secret`, is created, but Bob is not allowed to read it.

```
as principal admin password "admin" do
  create principal bob "bobs_password"
  set secret = "Super secret"
  return "success"
***

{"status":"CREATE_PRINCIPAL"}
{"status":"SET"}
{"status":"RETURNING","output":"success"}
```

Now Bob attempts to read the `secret` variable with the following query.

```
as principal bob password "bobs_password" do
  return secret
***
```

Bob does not have permission to read `secret`, so the oracle returns `{"status": "DENIED"}`.

If the implementation returns the `secret` contents of `{"status": "RETURNING", "output": "Super secret"}`, we know a confidentiality violation has occurred.

Integrity breaks. Integrity violations are demonstrated in a similar manner, but occur when unprivileged users modify variables they don't have permission to write to. With the example above, the variable `secret`, is created, but Bob is not allowed to write to it. Now Bob attempts to write to `secret` with the following query.

```
as principal bob password "bobs_password" do
  set secret = "Bob's grade is an A!"
  return "success"
***
```

Bob does not have write permission on `secret`, so the oracle returns `{"status": "DENIED"}`.

If the implementation returns the following, an integrity violation has been demonstrated.

```
{"status": "SET"}
{"status": "RETURNING", "output": "success"}
```

Availability breaks. Unlike the ATM problem, we are able to assess availability violations for Multiuser DB (since we are not using a MITM). In this case, a security violation is possible when the server implementation is unable to process legal command programs. This is demonstrated when the target incorrectly denies a program by reporting an error, but the oracle successfully executes the program. Availability security violations also happen when the server implementation fails to respond to an input program within a fixed period of time.

Unlike the other two problems, the Multiuser DB problem does not place a limit on the number of security breaks submitted. In addition, the overall bug submission limit is reduced to 5, as opposed to 10 for the other two problems. Recall that for Secure Log and ATM, a break constitutes direct evidence that a vulnerability has been exploited, but not *which* vulnerability, if more than one is present. As such, if a build-it team were to fix a vulnerability during the fix-it phase, doing so would not shed light on which breaks derived from that vulnerability, vs. others. The contest thus limits break-it teams to one break each for confidentiality and integrity, per target team. (See §2.3.1 and §2.3.2.) For Multiuser DB, a security vulnerability is associated with a test run, so a fix of that vulnerability *will* unify all breaks that exploit that vulnerability, just as correctness fixes do. That means we need not impose a limit on them. The consequences of this design are discussed in §2.5.

2.4 Build-it Submissions: Successes and Failures

After running a BIBIFI contest, we have all of the code written by the build-it teams, and the bug reports submitted by the break-it teams. Looking at these artifacts, we can get a sense of what build-it teams did right, and what they did wrong. This section presents a sample of failure and success stories, while §2.5 presents a broader statistical analysis that suggests overall trends. A previous paper [51] provides a more detailed review of the types of vulnerabilities based on an extensive, in-depth qualitative analysis of submitted code.

2.4.1 Failure Stories

The failure modes for build-it submissions are distributed along a spectrum ranging from “failed to provide any security at all” to “vulnerable to extremely subtle timing attacks.” This is interesting because a similar dynamic is observed in the software marketplace today.

Secure Log. Many implementations of the Secure Log problem failed to use encryption or authentication codes, presumably because the builders did not appreciate the need for them. Exploiting these design flaws was trivial for break-it teams. Sometimes log data was written as plain text, other times log data was serialized using the Java object serialization protocol.

One break-it team discovered a privacy flaw which they could exploit with at most fifty probes. The target submission truncated the authentication token (i.e., the key) so that it was vulnerable to a brute force attack.

Some failures were common across Secure Log implementations: if an implementation used encryption, it might not use authentication. If it used authentication, it would authenticate records stored in the file individually, not globally. The implementations would also relate the ordering of entries in the file to the ordering of events in time, allowing for an integrity attack that changes history by re-ordering entries in the file.

There were five crashes due to memory errors, and they all occurred in C/C++ submissions. We examined two of the crashes and confirmed that they were ex-

exploitable. The first was a null pointer dereference, and the other was a buffer overflow from the use of `strcpy`.

ATM. The ATM problem allows for interactive attacks (not possible for the log), and the attacks became cleverer as implementations used cryptographic constructions incorrectly. One implementation used cryptography, but implemented RC4 from scratch and did not add any randomness to the key or the cipher stream. An attacker observed that the ciphertext of messages was distinguishable and largely unchanged from transaction to transaction, and was able to flip bits in a message to change the withdrawn amount.

Another implementation used encryption with authentication, but did not use randomness; as such error messages were always distinguishable from success messages. An attack was constructed against this implementation where the attack leaked the bank balance by observing different withdrawal attempts, distinguishing the successful from failed transactions, and performing a binary search to identify the bank balance given a series of withdraw attempts.

Some failures were common across ATM problem implementations. Many implementations kept the key fixed across the lifetime of the `bank` and `atm` programs and did not use a nonce in the messages. This allowed attackers to replay messages freely between the `bank` and the `atm`, violating integrity via unauthorized withdrawals. Several implementations used encryption, but without authentication. (This sort of mistake has been observed in real-world ATMs, as has, amazingly, a complete lack of encryption use [39].) These implementations used a library such

as OpenSSL, the Java cryptographic framework, or the Python pycrypto library to have access to a symmetric cipher such as AES, but either did not use these libraries at a level where authentication was provided in addition to encryption, or they did not enable authentication.

Multiuser DB. The Multiuser DB problem asks participants to consider a complex logical security problem. In this scenario, the specification was much more complicated. All but one team developed a system of home-grown access control checks. This led to a variety of failures when participants were unable to cover all possible security edge cases.

In some instances, vulnerabilities were introduced because they did not properly implement the specification. Some participants hardcoded passwords making them easily guessable by an attacker. Other participants did not include checks for the delegation command to ensure that the principal had the right to delegate along with the right they were trying to delegate.

Other participants failed to consider the security implications of their design decisions when the specification did not provide explicit instructions. For example, many of the participants did not check the delegation chain back to its root. Therefore, once a principal received an access right, they maintained this right even if it no longer belonged to the principal that delegated it to them.

There were two crashes targeting Multiuser DB implementations. Both were against C/C++ submissions. We inspected one crash and determined it was caused by code in the parser that dereferenced and executed an invalid (non-null) pointer.

Finally, other teams simply made errors when implementing the access control logic. In some cases, these mistakes introduced faulty logic into the access control checks. One team made a mistake in their control flow logic such that if a principal had no delegated rights, the access control checks were skipped—because a lookup error would have occurred. In other cases, these mistakes led to uncaught runtime errors that allowed the attacker to kill the server, making it unavailable to other users.

2.4.2 Success Stories

In contrast to the broken submissions, successful submissions followed understood best practices. For example, submissions made heavy use of existing high-level cryptographic libraries with few “knobs” that allow for incorrect usage [52]. Similarly, successful submissions limited the size and location of security-critical code [53].

ATM and Secure Log. One implementation of the ATM problem, written in Python, made use of the SSL PKI infrastructure. The implementation used generated SSL private keys to establish a root of trust that authenticated the `atm` program to the `bank` program. Both the `atm` and `bank` required that the connection be signed with the certificate generated at runtime. Both the `bank` and the `atm` implemented their communication protocol as plain text then wrapped in HTTPS. To find bugs in this system, other contestants would need to break the security of OpenSSL.

Another implementation, written in Java, used the NaCl library. This library

intentionally provides a very high level API to “box” and “unbox” secret values, freeing the user from dangerous choices. As above, to break this system, other contestants would need to break NaCl first.

A Java-based implementation of the Secure Log problem used the BouncyCastle library’s high-level API to construct a valid encrypt-then-MAC scheme over the entire log file. BouncyCastle allowed them to easily authenticate the whole log file, protecting them from integrity attacks that swapped the order of encrypted binary data in the log.

Multiuser DB. The most successful solutions for the Multiuser DB problem were localized access control logic checks to a single function with a general interface, rather repeating checking code for each command that needed it. Doing so reduced the likelihood of a mistake. One of the most successful teams used a fairly complex graphical representation of access control rules, but by limiting the number of places this graph was manipulated they could efficiently and correctly check access rights without introducing vulnerabilities.

2.5 Quantitative Analysis

This section quantitatively analyzes data we gathered from our 2015 and 2016 contests.⁶ We consider participants’ performance in each contest phase, identifying factors that contribute to high scores after the build-it round, resistance to breaking by other teams, and strong performance as breakers.

⁶We also ran a contest during Fall’14 [47] but exclude its data due to differences in how it was administered.

We find that on average, teams that program using statically-typed languages are $11\times$ less likely to have security bugs identified in their code compared to those using C and C++. Success in breaking, and particularly in identifying security bugs in other teams' code, is correlated with having more team members, as well as with participating successfully in the build-it phase (and therefore having given thought to how to secure an implementation). The use of advanced techniques like fuzzing and static analysis was dropped from the final model, indicating that their effect was not statistically significant. We note that such tools tend to focus on bugs, like memory errors and taint/code injection attacks, that were rare in our contests (per Section 2.4). Overall, integrity bugs were far more common than privacy bugs or crashes. The contests that used the ATM problem and the Multiuser DB problem were associated with more security bugs than the Secure Log contest.

2.5.1 Data collection

For each team, we collected a variety of observed and self-reported data. When signing up for the contest, teams reported standard demographics and features such as coding experience and programming language familiarity. After the contest, each team member optionally completed a survey about their performance. In addition, we extracted information about lines of code written, number of commits, etc. from teams' Git repositories.

Participant data was anonymized and stored in a manner approved by our institution's human-subjects review board. Participants consented to have data

related to their activities collected, anonymized, stored, and analyzed. A few participants did not consent to research involvement, so their personal data was not used in the data analysis.

2.5.2 Analysis approach

To examine factors that correlated with success in building and breaking, we apply regression analysis. Each regression model attempts to explain some outcome variable using one or more measured factors. For most outcomes, such as participants' scores, we can use ordinary linear regression, which estimates how many points a given factor contributes to (or takes away from) a team's score. To analyze binary outcomes, such as whether or not a security bug was found, we apply logistic regression, which estimates how each factor impacts the likelihood of an outcome.

We consider many variables that could potentially impact teams' results. To avoid over-fitting, we select as potential factors those variables that we believe are of most interest, within acceptable limits for power and effect size. As we will detail later, we use the same factors as the analysis in our earlier conference paper [42], plus one more, which identifies participation in the added contest (Multiuser DB). The impact of the added data on the analysis, compared to the analysis in the earlier paper, is considered in Section 2.5.8. We test models with all possible combinations of our chosen potential factors and select the model with the minimum Akaike Information Criterion (AIC) [54]. The final models are presented.

Each model is presented as a table with each factor as well as the p -value for

that factor. Significant p -values (< 0.05) are marked with an asterisk. Linear models include the coefficient relative to the baseline factor and the 95% confidence interval. Logistic models also include the exponential coefficient and the 95% confidence interval for the exponential coefficient. The exponential coefficient indicates how many times more likely the measured result occurs relative to the baseline factor.

We describe the results of each model below. This was not a completely controlled experiment (e.g., we do not use random assignment), so our models demonstrate correlation rather than causation. Our observed effects may involve confounds, and many factors used as independent variables in our data are correlated with each other. This analysis also assumes that the factors we examine have linear effect on participants' scores (or on likelihood of binary outcomes); while this may not be the case in reality, it is a common simplification for considering the effects of many factors. We also note that some of the data we analyze is self-reported, so may not be entirely precise (e.g., some participants exaggerating about which programming languages they know); however, minor deviations, distributed across the population, act like noise and have little impact on the regression outcomes.

2.5.3 Contestants

We consider three contests offered at different times:

Secure Log: We held one contest using the Secure Log problem (§2.3.1) during May–June 2015 as the capstone to a Cybersecurity MOOC sequence.⁷ Before completing in the capstone, participants passed courses on software security,

⁷<https://www.coursera.org/specializations/cyber-security>

Contest	USA	India	Russia	Brazil	Other
Spring 2015	30	7	12	12	120
Fall 2015	64	14	12	20	110
Fall 2016	44	13	4	12	103

Table 2.2: Contestants, by self-reported country.

cryptography, usable security, and hardware security.

ATM: During Oct.–Nov. 2015 we offered the ATM problem (§2.3.2) as two contests simultaneously, one as a MOOC capstone, and the other open to U.S.-based graduate and undergraduate students. We merged the contests after the build-it phase, due to low participation in the open contest. MOOC and open participants were ranked independently to determine grades and prizes.

Multiuser DB: In Sep.–Oct. 2016 we ran one contest offering the Multiuser DB problem (§2.3.3) open to both MOOC capstone participants as well as graduate and undergraduate students.

The U.S. was the most represented country in our contestant pool, but was not the majority. There was also representation from developed countries with a reputation both for high technology and hacking acumen. Details of the most popular countries of origin can be found in Table 2.2, and additional information about contestant demographics is presented in Table 2.3. In total, 156 teams participated in either the build-it or break-it phases, most of which participated in both.

2.5.4 Ship scores

We first consider factors correlating with a team’s *ship* score, which assesses their submission’s quality before it is attacked by the other teams (§2.2.1). This data

Contest	Spring 15	Fall 15	Fall 16
Problem	Secure Log	ATM	Multiuser DB
# Contestants	156	145	105
% Male	91 %	91 %	84 %
% Female	5 %	8 %	4 %
Age (mean/min/max)	34.8/20/61	32.2/17/69	29.9/18/55
% with CS degrees	35 %	35 %	39 %
Years programming	9.6/0/30	9.4/0/37	9.0/0/36
# Build-it teams	61	40	29
Build-it team size	2.2/1/5	3.1/1/6	2.5/1/8
# Break-it teams (that also built)	65 (58)	43 (35)	33 (22)
Break-it team size	2.4/1/5	3.1/1/6	2.6/1/8
# PLs known per team	6.8/1/22	9.1/1/20	7.8/1/17
% MOOC	100 %	84 %	65 %

Table 2.3: Demographics of contestants from qualifying teams. Some participants declined to specify gender.

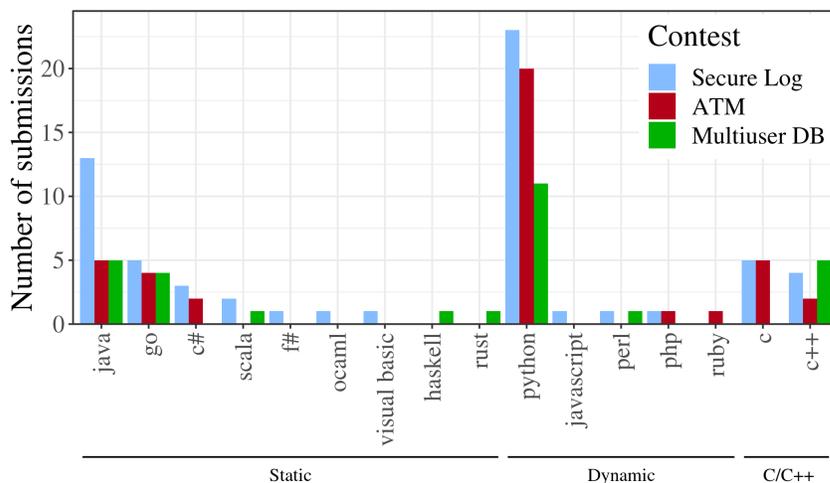


Figure 2.4: The number of build-it submissions in each contest, organized by primary programming language used. The languages are grouped by category.

set contains all 130 teams from the Secure Log, ATM, and Multiuser DB contests that qualified after the build-it phase. The contests have nearly the same number of correctness and performance tests, but different numbers of participants. We set the constant multiplier M to be 50 for the contests, which effectively normalizes the scores (see Section 2.2.2).

Model setup. To ensure enough statistical power to find meaningful relationships, our modeling was designed for a prospective effect size roughly equivalent to Cohen’s *medium* effect heuristic, $f^2 = 0.15$ [55]. An effect of this size corresponds to a coefficient of determination $R^2 = 0.13$, suggesting we could find an effect if our model can explain at least 13% of the variance in the outcome variable. We report the observed coefficient of determination for the final model with the regression results below.

As mentioned above, we reuse the factors chosen for the analysis in our earlier paper [42]. Their number was guided by a power analysis of the contest data we had at the time, which involved the $N = 101$ build-it teams that participated in Secure Log and ATM. With an assumed power of 0.75, the power analysis suggested we limit the covariate factors used in our model to nine degrees of freedom, which yields a prospective $f^2 = 0.154$. With the addition of Multiuser DB data, we add one more factor, which is choice of Multiuser DB as an option for which contest the submission belongs to. This adds a 10th degree of freedom, as well as 29 additional teams for a total $N = 130$. At 0.75 power, this yields a prospective $f^2 = 0.122$, which is better than in the earlier paper’s analysis.

We selected the factors listed in Table 2.4. *Knowledge of C* is included as a proxy for comfort with low-level implementation details, a skill often viewed as a prerequisite for successful secure building or breaking. *# Languages known* is how many unique programming languages team members collectively claim to know (see the second to last row of Table 2.3). For example, on a two-member team where member A claims to know C++, Java, and Perl and member B claims to

Factor	Description	Baseline
Contest	Secure Log, ATM, or Multiuser DB contest.	Secure Log
# Team members	A team’s size.	—
Knowledge of C	The fraction of team members who know C or C++.	—
# Languages known	Number of programming languages team members know.	—
Coding experience	Average years of programming experience.	—
Language category	C/C++, statically-typed, or dynamically-typed language.	C/C++
Lines of code	Lines of code count for the team’s final submission.	—
MOOC	If the team was participating in the MOOC capstone.	non-MOOC

Table 2.4: Factors and baselines for build-it models.

know Java, Perl, Python, and Ruby, the language count would be 5. *Language category* is the “primary” language category we manually identified in each team’s submission. Languages were categorized as either C/C++, statically-typed (e.g., Java, Go, but not C/C++), or dynamically-typed (e.g., Perl, Python). Precise category allocations, and total submissions for each language, segregated by contest, are given in Figure 2.4.

Results. The final model (Table 2.5) with $R^2 = 0.232$ captures almost $\frac{1}{4}$ of the variance. We find this number encouraging given how relatively uncontrolled the environment is and how many contributing, but unmeasured, factors there could be. Our regression results indicate that ship score is strongly correlated with language choice. Teams that programmed in C or C++ performed on average 133 and 112 points better than those who programmed in dynamically typed or statically typed languages, respectively. Figure 2.5 illustrates that while teams in many language

categories performed well in this phase, only teams that did not use C or C++ scored poorly.

The high scores for C/C++ teams could be due to better scores on performance tests and/or due to implementing optional features. We confirmed the main cause is the former. Every C/C++ team for the Secure Log contest implemented all optional features, while six teams in the other categories implemented only six of ten and one team implemented none; the ATM contest offered no optional features; for the Multiuser DB contest, four C/C++ teams implemented all optional features while one C/C++ team implemented five of nine. We artificially increased the scores of all teams as if they had implemented all optional features and reran the regression model. In the resulting model, the difference in coefficients between C/C++ and the other language categories dropped only slightly. This indicates that the majority of improvement in C/C++ ship score comes from performance.

The number of languages known by a team is not quite statistically significant, but the confidence interval in the model suggests that each programming language known increases ship scores by between 0 and 12 points. Intuitively, this makes sense since contestants that know more languages have more programming experience and have been exposed to different paradigms.

Lines of code is also not statistically significant, but the model hints that each additional line of code in a team's submission is associated with a minor drop in ship score. Based on our qualitative observations (see §2.4), we hypothesize this may relate to more reuse of code from libraries, which frequently are not counted in a team's LOC (most libraries were installed directly on the VM, not in the submission

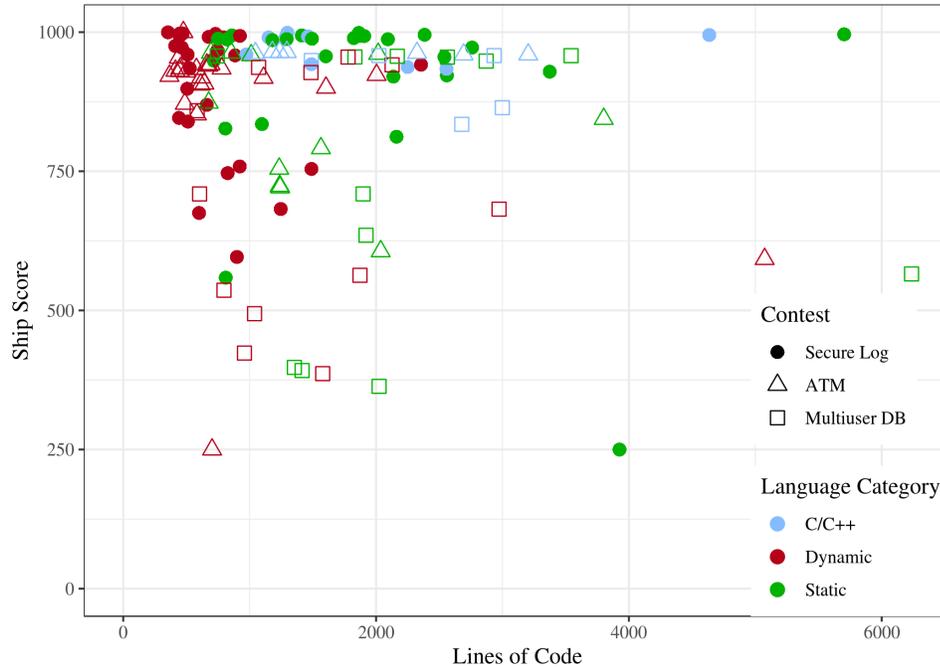


Figure 2.5: Each team’s ship score, compared to the lines of code in its implementation and organized by language category and contest. Using C/C++ correlates with a higher ship score.

itself). We also found that, as further noted above, submissions that used libraries with more sophisticated, lower-level interfaces tended to have more code and more mistakes; i.e., more steps took place in the application (more code) but some steps were missed or carried out incorrectly (less secure/correct). Figure 2.5 shows that LOC is (as expected) associated with the category of language being used. While LOC varied widely within each language type, dynamic submissions were generally shortest, followed by static submissions and then those written in C/C++ (which has the largest minimum size).⁸

⁸Our earlier model for the Secure Log and ATM contests found that lines of code was actually statistically significant. We discuss this further in §2.5.8.

Factor	Coef.	CI	<i>p</i> -value
Secure Log	—	—	—
ATM	-47.708	[-110.34, 14.92]	0.138
Multiuser DB	-163.901	[-234.2, -93.6]	<0.001*
C/C++	—	—	—
Statically typed	-112.912	[-192.07, -33.75]	0.006*
Dynamically typed	-133.057	[-215.26, -50.86]	0.002*
# Languages known	6.272	[-0.06, 12.6]	0.054
Lines of code	-0.023	[-0.05, 0.01]	0.118

Table 2.5: Final linear regression model of teams’ ship scores, indicating how many points each selected factor adds to the total score. $R^2 = 0.232$.

	Secure Log	ATM	Multiuser DB
Bug reports submitted	24,796	3,701	3,749
Bug reports accepted	9,482	2,482	2,046
Fixes submitted	375	166	320
Bug reports addressed by fixes	2,252	966	926

Table 2.6: Break-it teams in each contest submitted bug reports, which were judged by the automated oracle. Build-it teams then submitted fixes, each of which could potentially address multiple bug reports.

2.5.5 Resilience

Now we turn to measures of a build-it submission’s quality, starting with *resilience*. Resilience is a non-positive score that derives from break-it teams’ bug reports, which are accompanied by test cases that prove the presence of defects. The overall build-it score is the sum of ship score, just discussed, and resilience. Builders may increase the resilience component during the fix-it phase, as fixes prevent double-counting bug reports that identify the same defect (see §2.2.1).

Unfortunately, upon studying the data we found that a large percentage of

build-it teams opted not to fix any bugs reported against their code, forgoing the scoring advantage of doing so. We can see this in Figure 2.6, which graphs the resilience scores (Y-axis) of all teams, ordered by score, for the three contests. The circles in the plot indicate teams that fixed at least one bug, whereas the triangles indicate teams that fixed no bugs. We can see that, overwhelmingly, the teams with the lower resilience scores did not fix any bugs. Table 2.6 digs a little further into the situation. It shows that of the bug reports deemed acceptable by the oracle (the second row), submitted fixes (row 3) addressed only 23% of those from the Secure Log contest, 38% of those from the ATM contest, and 45% of those from the Multiuser DB contest (row 4 divided by row 2). It turns out that when counting only “active” fixers who fixed at least one bug, these averages were 56.9%, 72.5%, and 64.6% respectively.

Incentivizing fixing. This situation is disappointing, as we cannot treat resilience score as a good measure of code quality (when added to ship score). After the first two contests, we hypothesized that participants were not sufficiently incentivized to fix bugs, for two reasons. First, teams that were sufficiently far from the lead may have chosen to fix no bugs because winning was unlikely. Second, for MOOC students, once a minimum score is achieved they were assured to pass; it may be that fixing (many) bugs was unnecessary for attaining this minimum score.

We attempted to more strongly incentivize all teams to fix (duplicated) bugs by modifying the prize structure for the Multiuser DB contest. Instead of only giving away prizes to top teams, non-MOOC participants could still win monetary prizes

if they scored outside of third place. Placements were split into different brackets, and one team from each bracket was randomly selected to receive a prize. Prizes increased based on bracket position (*ex*, the fourth and fifth place bracket winner received \$500, while the sixth and seventh place bracket winner received \$375). Our hope was that builders would submit fixes to bump themselves into a higher bracket which would have a larger payout. Unfortunately, it does not appear that fix participation increased for non-MOOC participants for the Multiuser DB contest. To confirm this, we ran a linear regression model, but according to the model, incentive structure was not a factor in fix participation. The model did confirm that teams with a higher score at the end of break-it fixed a greater percentage of the bugs against them.

Additionally, we randomly sampled 60% of Multiuser DB teams to identify the types of vulnerabilities they chose to fix. We manually analyzed each break to determine the underlying vulnerability to determine whether the expected fix difficulty impacted team decisions. We did not observe any clear trend in the vulnerabilities teams chose to fix, with all vulnerability types both fixed by some teams and not fixed by others. Instead, we found teams most often made a binary decision, choosing to fix all (38%) or none (38%) of their vulnerabilities. The remaining teams only slightly strayed from a binary choice by either fixing all but one vulnerability (16%) or only one vulnerability (8%).

Wi et al. [56] developed Git-based CTF which is another contest that is inspired by BIBIFI. A key feature of this contest is that contestants periodically lose points for breaks that remain unfixed. This creates an incentive for participants

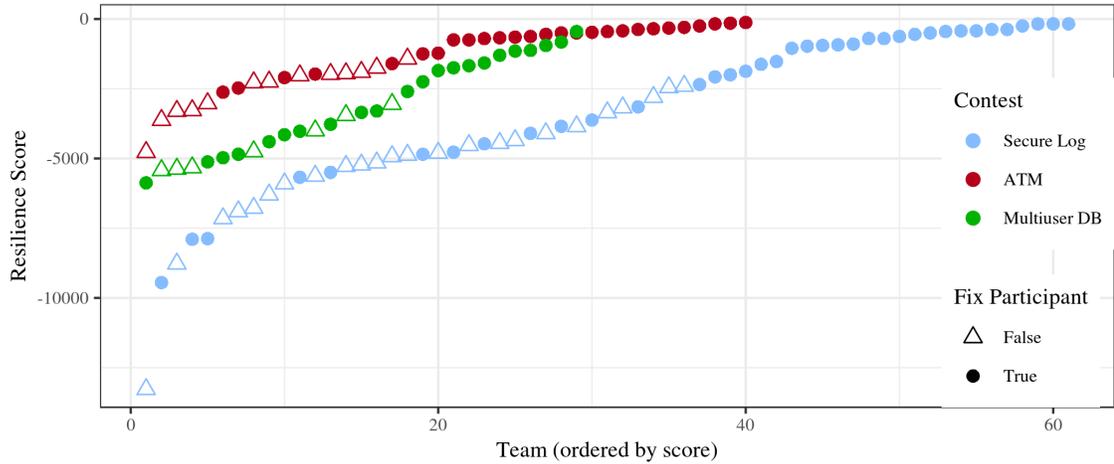


Figure 2.6: Final resilience scores, ordered by team, and plotted for each contest problem. Build-it teams who did not bother to fix bugs generally had lower scores.

to fix bugs as quickly as possible and makes a more real-time environment for the contest. We have incorporated this idea into BIBIFI by combining the break-it and fix-it rounds. As soon as a breaker team submits a bug, they receive points and the target builder team loses points. After a period of time (24 hours by default), the amount of points gained and lost increases linearly over time. Once builders submit a bug fix, they stop losing points for the breaks fixed by the fix submission. We include the 24 hour grace period so that builer teams do not need to be available at all times of day. If multiple breaker teams submitted the same bug, their points are split up when the fix is submitted and prorated for how long their break was active. In future contests, we plan to use this updated format and hope it will increase fix participation.

2.5.6 Presence of security bugs

While resilience score is not sufficiently meaningful, a useful alternative is the likelihood that a build-it submission contains a security-relevant bug; by this we mean any submission against which at least one crash, privacy, integrity, or availability defect is demonstrated. In this model we used logistic regression over the same set of factors as the ship model.

Table 2.7 lists the results of this logistic regression; the coefficients represent the change in log likelihood associated with each factor. Negative coefficients indicate lower likelihood of finding a security bug. For categorical factors, the exponential of the coefficient ($\exp(\text{coef})$) indicates how strongly that factor being true affects the likelihood relative to the baseline category.⁹ For numeric factors, the exponential indicates how the likelihood changes with each unit change in that factor. R^2 as traditionally understood does not make sense for a logistic regression. There are multiple approximations proposed in the literature, each of which have various pros and cons. We present Nagelkerke ($R^2 = 0.619$) which suggests the model explains an estimated 61% of variance [57].

ATM implementations were far more likely than Secure Log implementations to have a discovered security bug.¹⁰ We hypothesize this is due to the increased security design space in the ATM problem as compared to the Secure Log problem.

⁹In cases (such as the ATM contest) where the rate of security bug discovery is close to 100%, the change in log likelihood starts to approach infinity, somewhat distorting this coefficient upwards.

¹⁰This coefficient (corresponding to $103\times$) is somewhat exaggerated (see prior footnote), but the difference between contests is large and significant.

Factor	Coef.	Exp(coef)	Exp CI	<i>p</i> -value
Secure Log	—	—	—	—
ATM	4.639	103.415	[18, 594.11]	<0.001*
Multiuser DB	3.462	31.892	[7.06, 144.07]	<0.001*
C/C++	—	—	—	—
Statically typed	-2.422	0.089	[0.02, 0.51]	0.006*
Dynamically typed	-0.99	0.372	[0.07, 2.12]	0.266
# Team members	-0.35	0.705	[0.5, 1]	0.051
Knowledge of C	-1.44	0.237	[0.05, 1.09]	0.064
Lines of code	0.001	1.001	[1, 1]	0.090

Table 2.7: Final logistic model measuring log-likelihood of the discovery of a security bug in a team’s submission. Nagelkerke $R^2 = 0.619$.

Although it is easier to demonstrate a security error in the Secure Log problem, the ATM problem allows for a much more powerful adversary (the MITM) that can interact with the implementation; breakers often took advantage of this capability, as discussed in §2.4.

Multiuser DB implementations were $31\times$ as likely as Secure Log implementations to have a discovered security bug. We hypothesize this is due to increased difficulty in implementing a custom access control system. There are limited libraries available that directly provide the required functionality, so contestants needed to implement access control manually, leaving more room for error. For the Secure Log problem, builders could utilize cryptographic libraries to secure their applications. In addition, it was potentially easier for breakers to discover attacks with the Multiuser DB problem since they could reuse break tests against multiple build submissions.¹¹

¹¹One caveat here is that a quirk of the problem definition permitted breakers to escalate correctness bugs into security problems by causing the state of Multiuser DB submissions

The model also shows that C/C++ implementations were more likely to contain an identified security bug than either static- or dynamic-language implementations. For static languages, this effect is significant and indicates that—assuming all other features are the same—a C/C++ program was about $11\times$ (that is, $1/0.089$ given in Table 2.7¹²) more likely to contain an identified bug. This effect is clear in Figure 2.7, which plots the fraction of implementations that contain a security bug, broken down by language type and contest problem. Of the 21 C/C++ submissions (see Figure 2.4), 17 of them had a security bug: 5/9 for the Secure Log contest, 7/7 for the ATM contest, and 5/5 for the Multiuser DB contest. All five of the buggy implementations from the Secure Log contest had a crash defect, and crashes were the only security-related problem for three of them; none of the ATM implementations had crash defects; two of the Multiuser DB C/C++ submissions had crash defects. All crash defects were due to violation of memory safety. More details about the crashes are presented in §2.4.1. Table 2.8 breaks down the number and percentage of teams that had different categories of security bugs.

The model shows four factors that played a role in the outcome, but not in a statistically significant way: using a dynamically typed language, lines of code of an implementation, developer knowledge of C, and number of team members. We see to become out of sync with the oracle implementation, and behave in a way that seemed to violate availability. We only realized after the contest was over that these should have been classified as correctness bugs. For the data analysis, we retroactively reclassified these bugs as correctness problems. Had they been classified properly during the contest, break-it team behavior might have changed, i.e., to spend more time hunting proper security defects.

¹²Here we use the inverse of the exponential coefficient for **Statically Typed** because we are describing the relationship between variables in the opposite direction than as presented in the table, i.e., the baseline C/C++ in comparison to **Statically Typed** as opposed to **Statically Typed** in comparison to baseline C/C++.

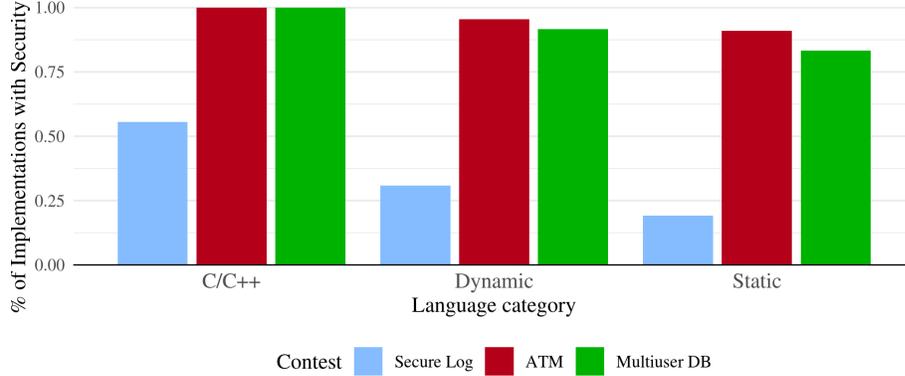


Figure 2.7: The fraction of teams in whose submission a security bug was found, by contest and language category.

Language Category	Integrity	Confidentiality	Integrity, Confidentiality, or Availability (Multiuser DB)	Crash
C/C++	9 / 43%	4 / 19%	5 / 24%	7 / 33%
Dynamic	27 / 45%	17 / 28%	11 / 18%	0 / 0%
Static	15 / 31%	10 / 20%	10 / 20%	0 / 0%

Table 2.8: The number and percentage of teams that had different types of security bugs by language category. Percentages are relative to total submissions in that language category, across all contests. Integrity, confidentiality, and availability bugs were not distinguished for the Multiuser DB problem during that contest. We group them in their own column.

the effect of the first in Figure 2.7. We note that the number of team members is just outside the threshold of being significant. This suggests that an implementation is $1.4\times (1/0.705)$ less likely to have a security bug present for each team member.

Finally, we note that MOOC participation was not included in our final model, indicating that (this kind of) security education did not have a significant effect in the outcome. Prior research [58] similarly did not find a significant effect of education in secure programming contexts. Our previous work investigating differences between experts (hackers) and non-experts (software testers) suggests improvements in vulnerability finding skill are driven by direct experiences with a variety of vulner-

abilities (e.g., discovering them, or being shown specific examples) [59]. Therefore, we hypothesize the hands-on experience of BIBIFI may support secure development improvement in ways that MOOC lectures, without direct experience, do not.

2.5.7 Breaking success

Now we turn our attention to break-it team performance, i.e., how effective teams were at finding defects in build-it teams' submissions. First, we consider how and why teams performed as indicated by their (normalized) break-it score *prior to the fix-it phase*. We do this to measure a team's raw output, ignoring whether other teams found the same bug (which we cannot assess with confidence due to the lack of fix-it phase participation per §2.5.5). This data set includes 141 teams that participated in the break-it phase for the Secure Log, ATM, and Multiuser DB contests. We also model which factors contributed to *security bug count*, or how many total security bugs a break-it team found. Doing this disregards a break-it team's effort at finding correctness bugs.

We model both break-it score and security bug count using several of the same potential factors as discussed previously, but applied to the breaking team rather than the building team. In particular, we include the *Contest* they participated in, whether they were *MOOC* participants, the number of break-it *Team members*, average team-member *Coding experience*, average team-member *Knowledge of C*, and unique *Languages known* by the break-it team members. We also add two new potential factors. 1) Whether the breaking team also qualified as a *Build*

Factor	Description	Baseline
Contest	Secure Log, ATM, or Multiuser DB contest.	Secure Log
# Team members	A team’s size.	—
Knowledge of C	The fraction of team members who know C or C++.	—
# Languages known	Number of programming languages team members know.	—
Coding experience	Average years of programming experience.	—
MOOC	If the team was participating in the MOOC capstone.	non-MOOC
Build participant	If the breaking team qualified as a build participant.	non-builder
Advanced techniques	If the breaking team used software analysis or fuzzing.	Not advanced

Table 2.9: Factors and baselines for break-it models.

participant. 2) Whether the breaking team reported using *Advanced techniques* like software analysis or fuzzing to aid in bug finding. Teams that only used manual inspection and testing are not categorized as advanced. 34 break-it teams (24%) reported using advanced techniques. These factors are summarized in Table 2.9.

When carrying out the power analysis for these two models, we aimed for a *medium* effect size by Cohen’s heuristic [55]. Assuming a power of 0.75, our conference paper considered a population of $N = 108$ for the Secure Log and ATM contests; with the eight degrees of freedom it yields a prospective effect size $f^2 = 0.136$. Including the Multiuser DB contest increases the degrees of freedom to nine and raises the population to $N = 141$. This yields a prospective effect size $f^2 = 0.107$, which (again) is an improvement over the initial analysis.

Break score. The model considering break-it score is given in Table 2.10. It has a coefficient of determination $R^2 = 0.15$ which is adequate. The model shows that

Factor	Coef.	CI	<i>p</i> -value
Secure Log	—	—	—
ATM	-2401.047	[-3781.59, -1020.5]	<0.001*
Multiuser DB	-61.25	[-1581.61, 1459.11]	0.937
# Team members	386.975	[45.48, 728.47]	0.028*
Coding experience	87.591	[-1.73, 176.91]	0.057
Build participant	1260.199	[-315.62, 2836.02]	0.119
Knowledge of C	-1358.488	[-3151.99, 435.02]	0.14

Table 2.10: Final linear regression model of teams’ break-it scores, indicating how many points each selected factor adds to the total score. $R^2 = 0.15$.

teams with more members performed better, with an average of 387 additional points per team member. Auditing code for errors is an easily parallelized task, so teams with more members could divide their effort and achieve better coverage.

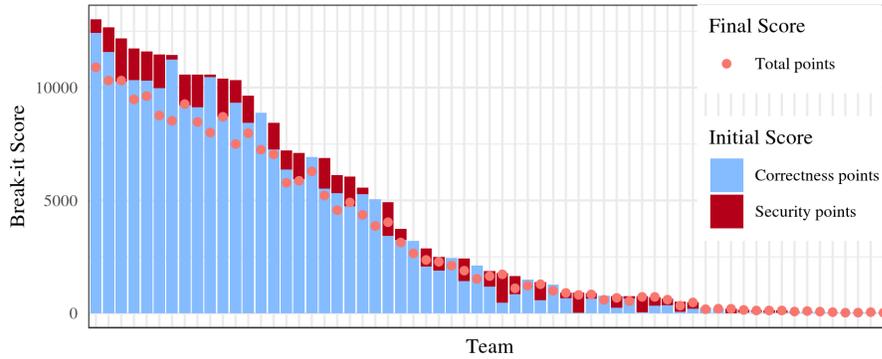
The model also indicates that Secure Log teams performed significantly better than ATM teams, and Multiuser DB teams performed similarly to ATM teams. Figure 2.8 illustrates that correctness bugs, despite being worth fewer points than security bugs, dominate overall break-it scores for the Secure Log contest. In the ATM contest, the scores are more evenly distributed between correctness and security bugs. This outcome is not surprising to us, as it was somewhat by design. The Secure Log problem defines a rich command-line interface with many opportunities for subtle correctness errors that break-it teams can target. It also allowed a break-it team to submit up to 10 correctness bugs per build-it submission. To nudge teams toward finding more security-relevant bugs, we reduced the submission limit from 10 to 5, and designed the ATM and Multiuser DB interface to be far simpler. For the Multiuser DB contest, an even greater portion of break-it scores come from

security bugs. This again was by design as we increased the security bug limit. Instead of submitting a maximum of two security bugs against a specific build-it team, breakers could submit up to five security (or correctness) bugs against a given team.

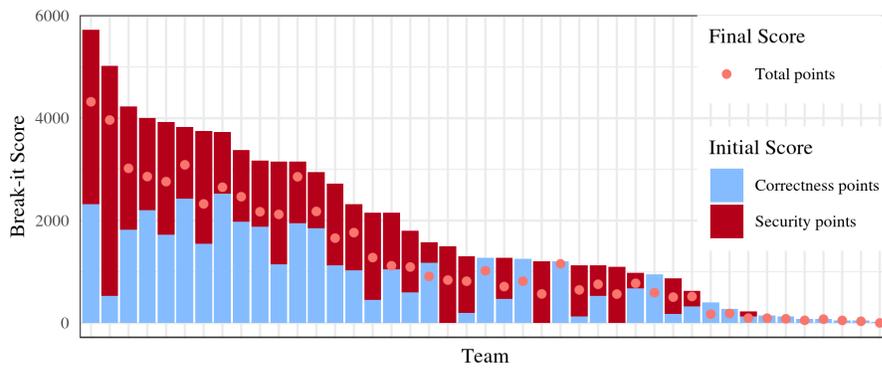
Interestingly, making use of advanced analysis techniques did not factor into the final model; i.e., such techniques did not provide a meaningful advantage in our context. This makes sense when we consider that such techniques tend to find generic errors such as crashes, bounds violations, or null pointer dereferences. Security violations for our problems are more often semantic, e.g., involving incorrect design or improper use of cryptography. Many correctness bugs were non-generic too, e.g., involving incorrect argument processing or mishandling of inconsistent or incorrect inputs.

Being a build participant and having more coding experience is identified as a positive factor in the break-it score, according to the model, but neither is statistically significant (though they are close to the threshold). Interestingly, knowledge of C is identified as a strongly negative factor in break-it score (though again, not statistically significant). Looking closely at the results, we find that *lack* of C knowledge is extremely *uncommon*, but that the handful of teams in this category did unusually well. However, there are too few of them for the result to be significant.

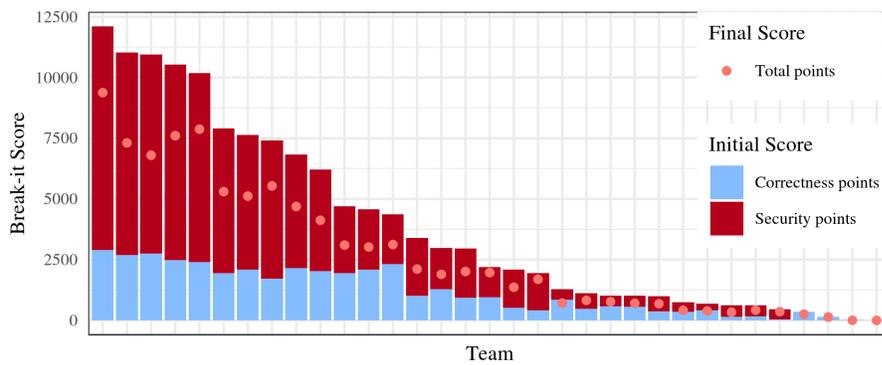
Again, we note MOOC participation was not included in our final model, suggesting this security education had at most a limited effect on breaking success.



(a) Secure Log



(b) ATM



(c) Multiuser DB

Figure 2.8: Scores of break-it teams prior to the fix-it phase, broken down by points from security and correctness bugs. The final score of the break-it team (after fix-it phase) is noted as a dot. Note the different ranges in the y -axes. In general, the Secure Log contest had the least proportion of points coming from security breaks.

Factor	Coef.	CI	<i>p</i> -value
Secure Log	—	—	—
Multuser DB	9.617	[5.84, 13.39]	<0.001*
ATM	3.736	[0.3, 7.18]	0.035*
# Team members	1.196	[0.35, 2.04]	0.006*
Build participant	4.026	[0.13, 7.92]	0.045*

Table 2.11: Final linear regression modeling the count of security bugs found by each team. Coefficients indicate how many security bugs each factor adds to the count. $R^2 = 0.203$.

Security bugs found. We next consider breaking success as measured by the count of security bugs a breaking team found. This model (Table 2.11) explains 20% of variance ($R^2 = 0.203$). The model again shows that team size is important, with an average of one extra security bug found for each additional team member. Being a qualified builder also significantly helps one’s score; this makes intuitive sense, as one would expect to gain a great deal of insight into how a system could fail after successfully building a similar system. Figure 2.9 shows the distribution of the number of security bugs found, per contest, for break-it teams that were and were not qualified build-it teams. Note that all but eight of the 141 break-it teams made some attempt, as defined by having made a commit, to participate during the build-it phase—most of these (115) qualified, but 18 did not. If the reason was that these teams were less capable programmers, that may imply that programming ability generally has some correlation with break-it success.

On average, four more security bugs were found by ATM teams than Secure Log teams. This contrasts with the finding that Secure Log teams had higher overall break-it scores, but corresponds to the finding that more ATM submissions had

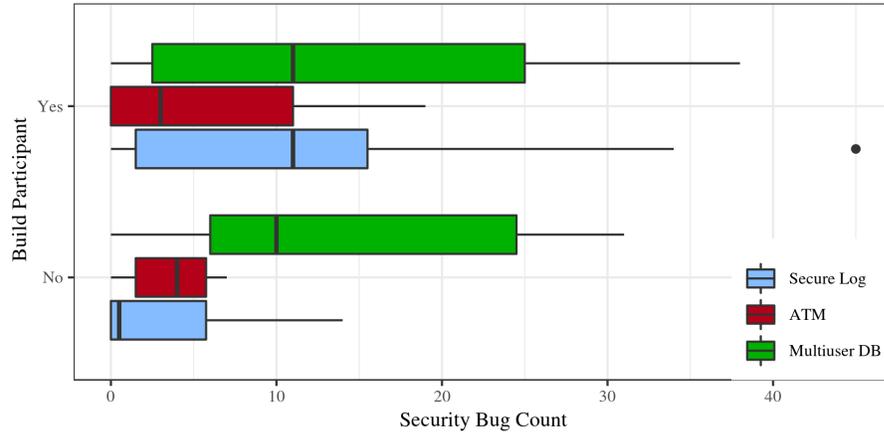


Figure 2.9: Count of security bugs found by each break-it team, organized by contest and whether the team also participated in build-it. The heavy vertical line in the box is the median, the boxes show the first and third quartiles, and the whiskers extend to the most outlying data within $\pm 1.5 \times$ the interquartile range. Dots indicate further outliers.

security bugs found against them. As discussed above, this is because correctness bugs dominated the Secure Log contest but were not as dominant in the ATM contest. Once again, the reasons may have been the smaller budget on per-submission correctness bugs for the ATM contest, and the greater potential attack surface in the ATM problem.

Multiuser DB teams found ten more security bugs on average than Secure Log teams. One possible reason is that the Multiuser DB contest permitted teams to submit up to five security bug reports per target, rather than just two. Another is that with Multiuser DB it was easier for breakers to reuse break tests to see when multiple targets were susceptible to the same bug.

2.5.8 Model differences

In the conference version of this work [42], we presented previous versions of these models with only data from the Secure Log and ATM contests. The updated models with Multiuser DB data are very similar to the original models, but there are some differences. We describe the differences to each model in this subsection.

Ship scores. In the original ship score model, students of the MOOC capstone performed 119 points better than non-MOOC teams. This correlation goes away when the Multiuser DB data is included. We hypothesize that this is due to prior coursework. MOOC students took three prior security courses that cover cryptography which is directly relevant to the Secure Log and ATM problem, but not the Multiuser DB problem.

Lines of code is not statistically significant in the updated model, but it was significant in the original model. Each additional line of code corresponded with a drop of 0.03 points in ship score. Code size may not have improved ship scores for the Multiuser DB contest due to the nature of the problem. Teams needed to implement custom access control policies and there are fewer libraries available that implement this functionality.

Presence of security bugs. In the original presence-of-security-bugs model, lines of code was a significant factor. Each additional line of code slightly increased the likelihood of a security bug being present (1.001 \times). Lines of code is not in the latest model, which is similar to the change in the ship score model (§2.5.4). We

hypothesize this change occurred for the same reasons.

Break score. The break score model basically remained the same between versions. The only difference is the coefficients slightly changed with the addition of the Multiuser DB contest.

Security bugs found. The linear regression model for the number of security bugs found essentially remained unchanged. The only material change was the addition of the factor that found that Multiuser DB breakers found more security bugs than Secure Log problem breakers.

2.5.9 Summary

The results of our quantitative analysis provide insights into how different factors correlate with success in building and breaking software. Programs written in C and C++ received higher ship scores due to better performance. C/C++ submissions were also $11\times$ more likely to have a reported security flaw than submissions written in statically typed languages.

Break-it teams with more team members found more security bugs and received more break-it points. Searching for vulnerabilities is easily parallelizable, so teams with more members could split the workload and audit more code. Successful build participants found more security bugs. This is intuitive as successfully building a program gives insights into the mistakes other similar programs might make.

2.6 Related work

BIBIFI bears similarity to existing programming and security contests but is unique in its focus on building secure systems. BIBIFI also is related to studies of code and secure development, but differs in its open-ended contest format.

Contests. Cybersecurity contests typically focus on vulnerability discovery and exploitation, and sometimes involve system administration for defense. One popular style of contest, dubbed *capture the flag* (CTF), is exemplified by a contest held at DEFCON [60]. Here, teams run an identical system that has buggy components. The goal is to find and exploit the bugs in other competitors' systems while mitigating the bugs in your own. Compromising a system enables a team to acquire the system's key and thus "capture the flag." In addition to DEFCON CTF, there are other CTFs such as iCTF [61, 62, 63], S3 [64], KotH [65] and PicoCTF [66]. The use of this style of contest in an educational setting has been explored in prior work [67, 68, 69]. The Collegiate Cyber Defense Challenge [31, 70, 71] and the Maryland Cyber Challenge & Competition [30] have contestants defend a system, so their responsibilities end at the identification and mitigation of vulnerabilities. These contests focus on bugs in systems as a key factor of play, but neglect software development.

Since BIBIFI's inception, additional contests have been developed in its style. Make it and Break it [72] is an evaluation of the Build-it, Break-it, Fix-it type of contest. Two teams were tasked with building a secure internet of things (IoT) smart

home with functionality including remote control of locks, speakers, and lighting. Teams then broke each other's implementations and found vulnerabilities like SQL injection and unauthorized control of locks. The contest organizers found this style of contest was beneficial in the development of cybersecurity skills and plan to run additional contests in the future. Git-based CTF [56] is similar to BIBIFI in that students were asked to implement a program according to a given specification. It differs in the fact that the CTF was fully run on Github and contestants used issue-tracking to submit breaks. In addition, builders were encouraged to fix breaks as soon as breaks were submitted since they periodically lost points for unfixed breaks. This seems to have been an effective motivation for convincing builders to fix their mistakes. We have integrated this idea into BIBIFI's infrastructure and plan to use it for future contests.

Programming contests challenge students to build clever, efficient software, usually with constraints and while under (extreme) time pressure. The ACM programming contest [36] asks teams to write several programs in C/C++ or Java during a 5-hour time period. Google Code Jam [73] sets tasks that must be solved in minutes, which are then graded according to development speed (and implicitly, correctness). Topcoder [35] runs several contests; the Algorithm competitions are small projects that take a few hours to a week, whereas Design and Development competitions are for larger projects that must meet a broader specification. Code is judged for correctness (by passing tests), performance, and sometimes subjectively in terms of code quality or practicality of design. All of these resemble the build-it phase of BIBIFI but typically consider smaller tasks; they do not consider the

security of the produced code.

Secure Development Practices and Advice. There is a growing literature of recommended practices for secure development. The BSIMM (“building security in” maturity model) [74] collects information from companies and places it within a taxonomy. Microsoft’s Security Development Lifecycle (SDL) [75] describes possible strategies for incorporating security concerns into the development process. Several authors make recommendations about development lifecycle and coding practices [76, 77, 78, 79, 80, 81]. Acar et al. collect and categorize 19 such resources [82].

Studies of secure software development. Researchers have considered how to include security in the development process. Work by Finifter and Wagner [83] and Prechelt [84] relates to both our build-it and break-it phases: they asked different teams to develop the same web application using different frameworks, and then subjected each implementation to automated (black box) testing and manual review. They found that both forms of review were effective in different ways, and that framework support for mitigating certain vulnerabilities improved overall security. Other studies focused on the effectiveness of vulnerability discovery techniques, e.g., as might be used during our break-it phase. Edmundson et al. [85] considered manual code review; Scandariato et al. [86] compared different vulnerability detection tools; other studies looked at software properties that might co-occur with security problems [87, 88, 89]. BIBIFI differs from all of these in its open-ended, contest

format: Participants can employ any technique they like, and with a large enough population and/or measurable impact, the effectiveness of a given technique will be evident in final outcomes.

Other researchers have examined what factors influence the development of security errors. Common findings include developers who do not understand the threat model, security APIs with confusing options and poorly chosen defaults, and “temporary” test configurations that were not corrected prior to deployment [90, 91, 92]. Interview studies with developers suggest that security is often perceived as someone else’s responsibility, not useful for career advancement, and not part of the “standard” developer mindset [93, 94]. Anecdotal recommendations resulting from these interviews include mandating and rewarding secure coding practices, ensuring that secure tools and APIs are more attractive than less secure ones, enable “security champions” with broadly defined roles, and favoring ongoing dialogue over checklists [95, 96, 97]. Developer Observatory [98, 99, 100] is an online platform that enables large-scale controlled security experiments by asking software developers to complete a security relevant programming tasks in the browser. Using this platform, Acar et al. studied how developer experience and API design for cryptographic libraries impact software security. Oliveira et al. [101] performed an experiment on security blindspots, which they define as a misconception, misunderstanding, or oversight by the developer in the use of an API. Their results indicate that API blindspots reduce a developer’s ability to identify security concerns, I/O functionality is likely to cause blindspots, and experience does not influence a developer’s ability to identify blindspots. Thompson [102] analyzed thousands of open

source repositories and found that code review of pull requests reduced the number of reported security bugs.

Crash de-duplication. For accurate scoring, BIBIFI identifies duplicate bug reports by unifying the ones addressed by the same (atomic) fix. But this approach is manual, and relies on imperfect incentives. Other works have attempted to automatically de-duplicate bug reports, notably those involving crashes. Stack hashing [103] and AFL [104] coverage profiles offer potential solutions, however Klees et al. [105] show that fuzzers are poor at identifying which underlying bugs cause crashing inputs. Semantic crash bucketing [106] and symbolic analysis [107] show better results at mapping crashing inputs to unique bugs by taking into account semantic information of the program. The former supports BIBIFI’s view that program fixes correspond to unique bugs.

2.7 Conclusion

This work has presented Build-it, Break-it, Fix-it (BIBIFI), a new security contest that brings together features from typical security contests, which focus on vulnerability detection and mitigation but not secure development, and programming contests, which focus on development but not security. During the first phase of the contest, teams construct software they intend to be correct, efficient, and secure. During the second phase, break-it teams report security vulnerabilities and other defects in submitted software. In the final, fix-it, phase, builders fix reported bugs and thereby identify redundant defect reports. Final scores, following

an incentives-conscious scoring system, reward the best builders and breakers.

During 2015 and 2016, we ran three contests involving a total of 156 teams and three different programming problems. Quantitative analysis from these contests found that the best performing build-it submissions used C/C++, but submissions coded in a statically-typed language were less likely to have a security flaw. Break-it teams that were also successful build-it teams were significantly better at finding security bugs. Break-it teams with more members were more successful at breaking since auditing code is a task that is easily subdivided.

The BIBIFI contest administration code is available at <https://github.com/plum-umd/bibifi-code>; data from our contests is available in limited form, upon request. More information, data, and opportunities to participate are available at <https://builditbreakit.org>.

Chapter 3: LWeb: Information Flow Security for Multi-tier Web Applications

3.1 Introduction

Modern web applications must protect the confidentiality and integrity of their data. As seen in the outcomes of BIBIFI, ad hoc enforcement of security can lead to missing important design and implementation components, while automated enforcement through the use of safe programming languages can result in better security. Similarly, employing access control and/or manual enforcement mechanisms may fail to block illicit information flows between components, e.g., from database to server to client. Information flow control (IFC) [15] policies can govern such flows, but enforcing them poses practical problems. Static enforcement (e.g., by typing [108, 109, 110, 111, 112] or static analysis [113, 114, 115]) can produce too many false alarms, which hamper adoption [116]. Dynamic enforcement [117, 118, 119, 120, 121] is more precise but can impose high overheads.

A promising solution to these problems is embodied in the LIO system [16] for Haskell. LIO is a drop-in replacement for the Haskell IO monad, extending IO with an internal *current label* and *clearance label*. Such labels are lattice ordered (as is

typical [17]), with the degenerate case being a secret (high) label and public (low) one. LIO’s current label constitutes the least upper bound of the security labels of all values read during the current computation. Effectful operations such as reading/writing from stable storage, or communicating with other processes, are checked against the current label. If the operation’s security label (e.g., that on a channel being written to) is lower than the current label, then the operation is rejected as potentially insecure. The clearance serves as an upper bound that the current label may never cross, even prior to performing any I/O, so as to reduce the chance of side channels. Haskell’s clear, type-enforced separation of pure computation from effects makes LIO easy to implement soundly and efficiently, compared to other dynamic enforcement mechanisms.

This chapter presents **LWeb**, an extension to LIO that aims to bring its benefits to Haskell-based web applications. We present the three main contributions of our work.

First, we present an extension to a core LIO formalism with support for database transactions. Each table has a label that protects its length. In our implementation we use DC labels [122], which have both confidentiality and integrity components. The confidentiality component of the table label controls who can query it (as the result may reveal something about the table’s length), and the integrity component controls who can add or delete rows (since both may change the length). In addition, each row may have a more refined policy to protect its contents. The label for a field in a row may be specified as a function of other fields in the same row (those fields are protected by a specific, global label). This allows,

for example, having a row specifying a user and some sensitive user data; the former can act as a label to protect the latter.

We mechanized our formalism in Liquid Haskell [3] and proved that it enjoys noninterference. Our development proceeds in two steps: a core LIO formalism called λ_{LIO} (§ 3.3), and an extension to it, called λ_{LWeb} , that adds database operations (§ 3.4). The mechanization process was fruitful: it revealed two bugs in our original rules that constituted real leaks. Moreover, this mechanization constitutes the largest-ever development in Liquid Haskell and is the first Liquid Haskell application to prove a language metatheory (§ 3.5).

As our next contribution, we describe a full implementation of LWeb in Haskell as an extension to the Yesod web programming framework (§ 3.2 and § 3.6). Our implementation was carried out in two steps. First, we extracted the core label tracking functionality of LIO into a monad transformer called LMonad so that it can be layered on monads other than IO. For LWeb, we layered it on top of the Handler monad provided by the Yesod. This monad encapsulates mechanisms for client/server HTTP communications and database transactions, so layering LMonad on top of Handler provides the basic functionality to enforce security. Then we extended Yesod’s database API to permit defining label-based information flow policies, generalizing the approach from our formalism whereby each row may have many fields, each of which may be protected by other fields in the same row. We support simple key/value lookups and more general SQL queries, extending the Esqueleto framework [123]. We use Template Haskell [124] to insert checks that properly enforce policies in our extension.

Finally, we describe our experience using `LWeb` to build the web site hosting the BIBIFI contest, described in Chapter 2 (§ 3.7). The contest site has a variety of roles (participants, teams, judges, admins) and policies that govern their various privileges. When we first deployed the contest, it lacked `LWeb` support, and we found it had authorization bugs. Retrofitting it with `LWeb` was straightforward and eliminated those problems, reducing the trusted computing base from the entire application to just 80 lines of its code (1%) plus the `LWeb` codebase (for a total of 21%). `LWeb` imposes modest overhead on BIBIFI query latencies—experiments show between 2% and 21% (§ 3.8).

`LWeb` prevents the leakage of information as it flows through programs, but programs do release some information in practice. We investigate quantifying information flow (QIF) techniques that safely declassify information according to user-defined policies (§ 3.9). By modeling abstract domains in Liquid Haskell, we soundly and completely quantify how much information query functions release.

`LWeb` is not the first framework to use IFC to enforce database security in web applications. Examples of prior efforts include SIF/Swift [110, 112], Jacqueline [120], Hails [125, 126], SELinks [127], SeLINQ [111], UrFlow [128], and IFDB [129]. `LWeb` distinguishes itself by providing end-to-end IFC security (between/across server and database), backed by a formal proof (mechanized in Liquid Haskell), for a mature, full-featured web framework (`Yesod`) while supporting expressive policies (e.g., where one field can serve as the label of another) and efficient queries (a large subset of SQL). The IFC checks needed during query processing were tricky to get right—our formalization effort uncovered bugs in our original implementation by which

information could leak owing to the checks themselves. § 3.10 discusses related work in detail.

The code for `LWeb` and its mechanized proof are freely available at <https://github.com/jprider63/lmonad>, <https://github.com/jprider63/lmonad-yesod>, and <https://github.com/plum-umd/lmonad-meta>.

3.1.1 Acknowledgements

The work in this chapter previously appeared in Parker et al. [130] and was partly developed for my master’s thesis [131]. It was a collaboration with Niki Vazou and Michael Hicks. We all contributed to the meta-theory; Niki and I developed the mechanization; I implemented the Haskell library that dynamically enforced non-interference and applied it to the BIBIFI contest site.

We would like to thank Alejandro Russo and anonymous reviewers for helpful comments on a draft of this work. This work was supported in part by the National Science Foundation under grant CNS-1801545 and by DARPA under contract FA8750-16-C-0022.

3.2 Overview

The architecture of `LWeb` is shown in fig. 3.1. Database queries/updates precipitated by user interactions are processed by the `LMonad` component, which constitutes the core of `LIO` and confirms that label-based security policies are not violated. Then, the queries/updates are handled via `Yesod`, where the results continue to be

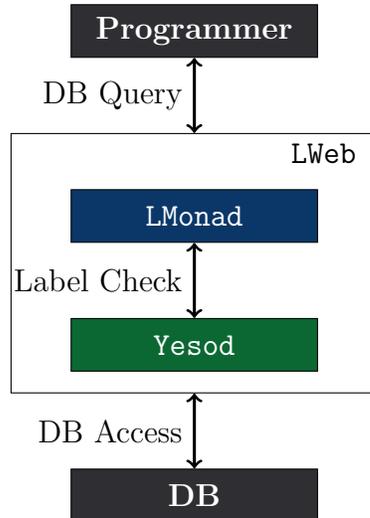


Figure 3.1: Structure of LWeb.

```

class Eq a ⇒ Label a where
  ⊥    :: a
  (⊔)  :: a → a → a
  (⊓)  :: a → a → a
  (⊑)  :: a → a → Bool

```

Figure 3.2: The Label class

subject to policy enforcement by LMonad.

3.2.1 Label-Based Information Flow Control with LIO

We start by presenting LIO [16] and how it is used to enforce noninterference for label-based information flow policies.

Labels and noninterference. As a trivial security label, consider a datatype with constructors `Secret` and `Public`. Protected data is assigned a label, and an IFC system ensures that `Secret`-labeled data can only be learned by those with `Secret`-label privilege or greater. The label system can be generalized to any lattice [17]

where IFC is checked using the lattice’s partial order relation \sqsubseteq . Such a system enjoys *noninterference* [132] if an adversary with privileges at label l_1 can learn nothing about data labeled with l_2 where $l_2 \not\sqsubseteq l_1$.

In fig. 3.2 we define the label interface as the type class `Label` that defines the bottom (least protected) label, least upper bound (join, \sqcup) of two labels, the greatest lower bound (meet, \sqcap), and whether one label can flow to (\sqsubseteq) another, defining a partial ordering. Instantiating this type class for `Public` and `Secret` would set `Public` as the bottom label and `Public` \sqsubseteq `Secret` (with join and meet operations to match).

The LIO monad. LIO enforces IFC on labeled data using dynamic checks. The type `LIO l a` denotes a monadic computation that returns a value of type `a` at label `l`. LIO provides two methods to label and unlabel data.

```
label  :: (Label l) => l -> a -> LIO l (Labeled l a)
unlabel :: (Label l) => Labeled l a -> LIO l a
```

The method `label l v` takes as input a label and some data and returns a `Labeled` value, i.e., the data `v` marked with the label `l`. The method `unlabel v` takes as input a labeled value and returns just its data. The LIO monad maintains an ambient label—the *current label* `lc`—that represents the label of the current computation. As such, labelling and unlabelling a value affects `lc`. In particular, `unlabel v` updates `lc` by joining it to `v`’s label, while `label l v` is only permitted if `lc` \sqsubseteq `l`, i.e., the current label can flow to `l`. If this check fails, LIO raises an exception.

As an example, on the left, a computation with current label `Public` labels

data "a secret" as `Secret`, preserving the same current label, and then unlabels the data, thus raising the current label to `Secret`. On the right, a computation with current label `Secret` attempts to label data as `Public`, which fails, since the computation is already tainted with (i.e., dependent on) secret data.

```

-- lc := Public
v ← label Secret "a secret"
-- ok: Public ⊆ Secret and lc := Public
x ← unlabel v
-- lc := Secret

-- lc := Secret
v ← label Public "public"
-- exception: Secret ⊈ Public

```

LIO also supports labeled mutable references, and a scoping mechanism for temporarily (but safely) raising the current label until a computation completes, and then restoring it. LIO also has what is called the *clearance* label that serves as an upper bound for the current label, and thus can serve to identify potentially unsafe computations sooner.

A normal Haskell program can run an LIO computation via `runLIO`, whose type is as follows.

```
runLIO :: (Label l) ⇒ LIO l a → IO a
```

Evaluating `runLIO m` initializes the current label to \perp and computes `m`. The returned result is an `IO` computation, since LIO allows `IO` interactions, e.g., with a file system. If any security checks fail, `runLIO` throws an exception.

3.2.2 Yesod

`Yesod` [133] is mature framework for developing type-safe and high performance web applications in Haskell. In a nutshell, `LWeb` adds LIO-style support to

```

Friends <⊥,Const Admin>
  user1 Text <⊥,Const Admin>
  user2 Text <⊥,Const Admin>
  date  Text <Field User1 ⊓ Field User2,Const Admin>

```

Figure 3.3: Example LWeb database table definition. The green is Yesod syntax and the blue is the *LWeb* policy.

Yesod-based web applications, with a focus on supporting database security policies.

The green part of fig. 3.3 uses Yesod’s domain specific language (DSL) to define the table `Friends`. The table has three `Text`¹ fields corresponding to two users (`user1` and `user2`) and the date of their friendship. A primary key field with type `FriendsId` is also automatically added. In § 3.2.3 we explain how the blue part of the definition is used for policy enforcement.

Yesod uses Template Haskell [124] to generate, at compile time, a database schema from such table definitions. These are the Haskell types that Yesod generates for the `Friends` table.

```

data FriendsId = FriendsId Int
data Friends = Friends { friendsUser1 :: Text, friendsUser2 :: Text
                        , friendsDate  :: Text }

```

Note that though each row has a key of type `FriendsId`, it is elided from the `Friends` data record. Each generated key type is a member of the `Key` type family; in this case `Key Friends` is a type alias for `FriendsId`.

Yesod provides an API to define and run queries. Here is a simplified version of this API.

```

runDB  :: YesodDB a → Handler a

```

¹Text is an efficient Haskell string type.

```

get    :: Key v → YesodDB (Maybe v)
insert :: v      → YesodDB (Key v)
delete :: Key v → YesodDB ()
update :: Key v → [Update v] → YesodDB ()

```

The type alias `YesodDB a` denotes the monadic type of a computation that queries (or updates) the database. The function `runDB` runs the query argument on the database. `Handler` is `Yesod`'s underlying monad used to respond to HTTP requests. The functions `get`, `insert`, `delete`, and `update` generate query computations. For example, we can query the database for the date of a specific friendship using `get`.

```

getFriendshipDate :: FriendsId → Handler (Maybe Text)
getFriendshipDate friendId = do
  r ← runDB (get friendId)
  return (friendsDate <$> r)

```

`Yesod` also supports more sophisticated SQL-style queries via an interface called `Esqueleto` [123]. Such queries may include inner and outer joins, conditionals, and filtering.

3.2.3 LWeb: Yesod with LIO

`LWeb` extends `Yesod` with `LIO`-style IFC enforcement. The implementation has two parts. As a first step, we generalize `LIO` to support an arbitrary underlying monad by making it a *monad transformer*, applying it to `Yesod`'s core monad. Then we extend `Yesod` operations to incorporate label-based policies that work with this extended monad.

LMonad: LIO as a monad transformer. `LMonad` generalizes the underlying `IO` monad of `LIO` to *any* monad `m`. In particular, `LMonad` is a monad transformer `LMonadT 1 m`

that adds the IFC operations to the underlying monad `m`, rather than making it specific to the `IO` monad.

```
label      :: (Label l, Monad m) => l -> a -> LMonadT l m (Labeled l a)
unlabel    :: (Label l, Monad m) => Labeled l a -> LMonadT l m a
runLMonad  :: (Label l, Monad m) => LMonadT l m a -> m a
```

`LMonadT` is implemented as a state monad transformer that tracks the current label. Computations that run in the underlying `m` monad cannot be executed directly due to Haskell’s type system. Instead, safe variants that enforce IFC must be written so that they can be executed in `LMonadT l m`. Thus, the `LIO` monad is an instantiation of the monad variable `m` with `IO`: `LIO l = LMonadT l IO`. For `LWeb` we instantiate `LMonadT` with `Yesod`’s `Handler` monad.

```
type LHandler l a = LMonadT l Handler a
```

Doing this adds information flow checking to `Yesod` applications, but it still remains to define policies to be checked. Thus we extend `Yesod` to permit defining label-based policies on database schemas, and to enforce those policies during query processing.

Label-annotated database schemas. `LWeb` labels are based on DC labels [122], which have the form `<l,r>`, where the left protects the *confidentiality* and the right protects the *integrity* of the labeled value. Integrity lattices are dual to confidentiality lattices. They track who can influence the construction of a value.

Database policies are written as label annotations `p` on table definitions, following this grammar:

```

p := <l, l>
l := Const c | Field f | Id |  $\top$  |  $\perp$  |  $l \sqcap l$  |  $l \sqcup l$ 

```

Here, c is the name of a data constructor and f is a field name. A database policy consists of a single *table label* and one label for each field in the database. We explain these by example.

The security labels of the `Friends` table are given by the blue part of fig. 3.3. The first line's label `Friends < \perp , Const Admin>` defines the table label, which protects the *length* of the table. This example states that anyone can learn the length of the table (e.g., by querying it), but only the administrator can change the length (i.e., by adding or removing entries). `LWeb` requires the table label to be constant, i.e., it may not depend on run-time entries of the table. Allowing it to do so would significantly complicate enforcing noninterference.

The last line `date Text <Field User1 \sqcap Field User2, Const Admin>` defines that either of the users listed in the first two fields can read the `date` field but only the administrator can write it. This label is *dynamic*, since the values of the `user1` and `user2` fields may differ from row to row. We call fields, like `user1` and `user2`, which are referenced in another field's label annotation, *dependency fields*. When a field's label is not given explicitly, the label `< \perp , \top >` is assumed. To simplify security enforcement, `LWeb` requires the label of a dependency field to be constant and flow into (be bounded by) the table label. For `user1` and `user2` this holds since their labels match the table's label.

The invariants about the table label and the dependency field labels are enforced by a compile-time check, when processing the table's policy annotations.

Note that `Labeled` values may not be directly stored in the database as there is no way to directly express such a type in a source program. Per fig. 3.3, field types like `Text`, `Bool`, and `Int` are allowed, and their effective label is indicated by annotation, rather than directly expressed in the type.²

Policy enforcement. `LWeb` enforces the table-declared policies by providing wrappers around each `Yesod` database API function.

```
runDB  :: Label l => LWebDB l a -> LHandler l a
get    :: Label l => Key v -> LWebDB l (Maybe v)
insert :: Label l => v     -> LWebDB l (Key v)
delete :: Label l => Key v -> LWebDB l ()
update :: Label l => Key v -> [Update v] -> LWebDB l ()
```

Now the queries are modified to return `LWebDB` computations that are evaluated (using `runDB`) inside the `LHandler` monad. For each query operation, `LWeb` wraps the underlying database query with information flow control checks that enforce the defined policies. For instance, if `x` has type `FriendsId`, then `r ← runDB $ get x` joins the current label with the label of the selected row, here `user1` \sqcap `user2`.

`LWeb` also extends IFC checking to advanced SQL queries expressed in `Esqueleto` [123]. As explained in § 4.3, `LWeb` uses a DSL syntax, as a `lsql` quasiquotation, to wrap these queries with IFC checks. For example, the following query joins the `Friends` table with a `User` table:

```
rs ← runDB [lsql|select * from Friends inner join User on Friends.user1
  == User.id|]
```

²The formalism encodes all of these invariants with refinement types in the database definition.

3.3 Mechanizing Noninterference of LIO in Liquid Haskell

A contribution of this work is a formalization of `LWeb`'s extension to `LIO` to support database security policies, along with a proof that this extension satisfies (termination insensitive) noninterference. We mechanize our formalization in Liquid Haskell (§ 1.1). Our mechanized formalization and proof of noninterference constitutes the first significant metatheoretical mechanization carried out in Liquid Haskell.

We present our mechanized `LWeb` formalism in two parts. In this section, we present λ_{LIO} , a formalization and proof of noninterference for `LIO`. The next section presents λ_{LWeb} , an extension of λ_{LIO} that supports database operations. Our Liquid Haskell mechanization defines λ_{LIO} 's syntax and operational semantics as Haskell definitions, as a definitional interpreter. We present them the same way, rather than reformatting them as mathematical inference rules. Metatheoretic properties are expressed as refinement types, following Vazou et al. [134, 135], and proofs are Haskell functions with these types (checked by the SMT solver). We assess our experience using Liquid Haskell for metatheory in comparison to related approaches in § 3.5.

3.3.1 Security Lattice as a Type Class

Figure 3.4 duplicates the `Label1` class definition of Figure 3.2 but extends it with several methods that use refinement types to express properties of lattices that labels are expected to have.

```

class Label l where
  (⊆) :: l → l → Bool
  (∩) :: l → l → l
  (∪) :: l → l → l
  ⊥    :: l

  lawBot          :: l:l → { ⊥ ⊆ l }
  lawFlowReflexivity :: l:l → { l ⊆ l }
  lawFlowAntisymmetry :: l1:l → l2:l → { (l1 ⊆ l2 ∧ l2 ⊆ l1) ⇒ l1 == l2 }
  lawFlowTransitivity :: l1:l → l2:l → l3:l → { (l1 ⊆ l2 ∧ l2 ⊆ l3) ⇒ l1 ⊆ l3 }

  lawMeet :: z:l → l1:l → l2:l → l:l
           → { z == l1 ∩ l2 ⇒ z ⊆ l1 ∧ z ⊆ l2 ∧ (l ⊆ l1 ∧ l ⊆ l2 ⇒ l ⊆ z) }
  lawJoin :: z:l → l1:l → l2:l → l:l
           → { z == l1 ∪ l2 ⇒ l1 ⊆ z ∧ l2 ⊆ z ∧ (l1 ⊆ l ∧ l2 ⊆ l ⇒ z ⊆ l) }

```

Figure 3.4: `Label` type class extended with `law*` methods to define the lattice laws as refinement types.

Partial order. The method `(⊆)` defines a partial order for each `Label` element. That is, `(⊆)` is reflexive, antisymmetric, and transitive, as respectively encoded by the refinement types of the methods `lawFlowReflexivity`, `lawFlowAntisymmetry`, and `lawFlowTransitivity`. For instance, `lawFlowReflexivity` is a method that takes a label `l` to a Haskell unit (i.e., `l → ()`). This type is refined to encode the reflexivity property `l:l → {v:() | l ⊆ l }` and further simplifies to ignore the irrelevant `v:()` part as `l:l → { l ⊆ l }`. With that refinement, application of `lawFlowReflexivity` to a concrete label `l` gives back a proof that `l` can flow to itself (i.e., `l ⊆ l`). At an instance definition of the class `Label`, the reflexivity proof needs to be explicitly provided.

Lattice. Similarly, we refine the `lawMeet` method to define the properties of the (\sqcap) lattice operator. Namely, for all labels `l1` and `l2`, we define `z == l1 \sqcap l2` so that (i) `z` can flow to `l1` and `l2` (`z \sqsubseteq l1 \wedge z \sqsubseteq l2`) and (ii) all labels that can flow to `l1` and `l2`, can also flow to `z` (`forall1 l. l \sqsubseteq l1 \wedge l \sqsubseteq l2 \Rightarrow l \sqsubseteq z`). Dually, we refine the `lawJoin` method to describe `l1 \sqcup l2` as the minimum label that is greater than `l1` and `l2`.

Using the lattice laws. The lattice laws are class methods, which can be used for each `l` that satisfies the `Label` class constraints. For example, we prove that for all labels `l1`, `l2`, and `l3`, `l1 \sqcup l2` cannot flow into `l3` *iff* `l1` and `l2` cannot both flow into `l3`.

```
joinIff :: Label l => l1:l -> l2:l -> l3:l -> {l1  $\sqsubseteq$  l3  $\wedge$  l2  $\sqsubseteq$  l3  $\Leftrightarrow$  (l1
   $\sqcup$  l2)  $\sqsubseteq$  l3}
joinIff l1 l2 l3 = lawJoin (l1  $\sqcup$  l2) l1 l2 l3 ? lawFlowTransitivity l1
  l2 l3
```

The theorem is expressed as a Haskell function that is given three labels and returns a unit value refined with the desired property. The proof proceeds by calling the laws of join and transitivity, combined with the proof combinator (?) that ignores its second argument (i.e., defined as `x ? _ = x`) while passing the refinements of both arguments to the SMT solver. The contrapositive step is automatically enforced by refinement type checking, using the SMT solver.

3.3.2 λ_{LIO} : Syntax and Semantics

Now we present the syntax and operational semantics of λ_{LIO} .

```

data Program l = Pg { pLabel :: l, pTerm :: Term l } | PgHole

data Term l
  -- pure terms
= TUnit | TInt Int | TLabel l | TLabeled l (Term l) | TLabelOf (Term l)
  | TVar Var | TLam Var (Term l) | TApp (Term l) (Term l) | THole | ...
  -- monadic terms
| TBind (Term l) (Term l) | TReturn (Term l) | TGetLabel | TLIO (Term
  l)
| TLabel (Term l) (Term l) | TUnlabel (Term l) | TException
| TToLabeled (Term l) (Term l)

```

Figure 3.5: Syntax of λ_{LIO} .

3.3.2.1 Syntax

Figure 3.5 defines a program as either an actual program (`Pg`) with a current label `pLabel` under which the program’s term `pTerm` is evaluated, or as a hole (`PgHole`). The hole is not a proper program; it is used for to define adversary observability when proving noninterference (§ 3.3.3). We omit the clearance label in the formalism as a simplification since its rules are straightforward (when the current label changes, check that it flows into the clearance label). Terms are divided into *pure* terms whose evaluation is independent of the current label and *monadic* terms, which either manipulate or whose evaluation depends on the current label.

Pure terms. Pure terms include unit `TUnit`, integers `TInt i` for some Haskell integer `i`, and the label value `TLabel l`, where `l` is some instance of the labeled class of Figure 3.4. The labeled value `TLabeled l t` wraps the term `t` with the label `l`. The term `TLabelOf t` returns the label of the term `t`, if `t` is a labeled term. Pure terms include the standard lambda calculus terms for variables (`TVar`), application

(TApp) and abstraction (TLam). Finally, similar to programs, a hole term (THole) is required for the meta-theory. It is straightforward to extend pure terms to more interesting calculi. In our mechanization we extended pure terms with lattice label operations, branches, lists, and inductive fixpoints; we omit them here for space reasons.

Monadic terms. Monadic terms are evaluated under a state that captures the current label. Bind (TBind) and return (TReturn) are the standard monadic operations, that respectively propagate and return the current state. The current label is accessed with the TGetLabel term and the monadic term TLIO wraps monadic values, i.e., computations that cannot be further evaluated. The term TLabel lt t labels the term t with the label term lt and dually the term TUnlabel t unlabels the labeled term t . An exception (TException) is thrown if a policy is violated. Finally, the term TToLabeled $t1$ t locally raises the current label to $t1$ to evaluate the monadic term t , dropping it again when the computation completes.

3.3.2.2 Semantics

Figure 3.6 summarizes the operational semantics of λ_{LIO} as three main functions, (i) `eval` evaluates monadic terms taking into account the current label of the program, (ii) `evalTerm` evaluates pure terms, and (iii) `eval***` is the transitive closure of `eval`.

Program evaluation. The `bind` of two terms t_1 and t_2 fully evaluates t_1 into a monadic value, using evaluation's transitive closure `eval$*$`. The result is passed to t_2 . The returned program uses the label of the evaluation of t_1 , which is safe since evaluation only increases the current label. In the definition of evaluation, we use Haskell's guard syntax `Pg lc' (TLIO t1') ← eval$*$ (Pg lc t1)` to denote that evaluation of `bind` only occurs when `eval$*$ (Pg lc t1)` returns a program whose term is a monadic value `TLIO`. Using refinement types, we prove that assuming that programs cannot diverge and are well-typed (i.e., t_1 is a monadic term), `eval$*$ (Pg lc t1)` always returns a program with a monadic value, so evaluation of `bind` always succeeds. Evaluation of the `TReturn` term simply returns a monadic value and evaluation of `TGetLabel` returns the current label. Evaluation of `TLabel` `(TLabel l) t` returns the term t labeled with l , when the current label can flow to l , otherwise it returns an exception. Dually, unlabeled `TLabeled l t` returns the term t with the current label joined with l . The term `ToLabeled (TLabel l) t` under current label lc fully evaluates the term t into a monadic value t' with returned label lc' . If both the current and returned labels can flow into l , then evaluation returns the term t labeled with the returned label lc' , while the current label remains the same. That is, evaluation of t can arbitrarily raise the label, since its result is labeled under l . Otherwise, an exception is thrown. The rest of the terms are pure, and their evaluation rules are given below. Finally, evaluation of a hole is an identity.

Term evaluation. Evaluation of the term `TLabelOf t` returns the label of t , if t is a labeled term; otherwise it propagates evaluation until t is evaluated to a labeled term. Evaluation of application uses the standard call-by-name semantics. The definition of substitution is standard and omitted. The rest of the pure terms are either values or a variable, whose evaluation is defined to be the identity. We define `eval**` to be the transitive closure of `eval`. That is, `eval**` repeats evaluation until a monadic value is reached.

3.3.3 Noninterference

Now we prove noninterference for λ_{LIO} . Noninterference holds when the *low view* of a program is preserved by its evaluation. This low view is characterized by an *erasure* function, which removes program elements whose security label is higher than the adversary’s label, replacing them with a “hole.” Two versions of the program given possibly different secrets will start with the same low view, and if the program is noninterfering, they will end with the same low view. We prove noninterference of λ_{LIO} by employing a simulation lemma, in the style of Stefan et al. [16], Li and Zdancewic [136], Russo et al. [137]. We use refinement types to express this lemma and the property of noninterference, and rely on Liquid Haskell to certify our proof.

3.3.3.1 Erasure

The functions ϵ and ϵTerm erase the sensitive data of programs and terms,

resp.

```

 $\epsilon :: \text{Label } l \Rightarrow l \rightarrow \text{Program } l \rightarrow \text{Program } l$ 
 $\epsilon \ l \ (\text{Pg } lc \ t)$ 
  |  $lc \sqsubseteq l$       =  $\text{Pg } lc \ (\epsilon\text{Term } l \ t)$ 
  | otherwise     =  $\text{PgHole}$ 
 $\epsilon \_ \text{PgHole}$    =  $\text{PgHole}$ 

 $\epsilon\text{Term} :: \text{Label } l \Rightarrow l \rightarrow \text{Term } l \rightarrow \text{Term}$ 
 $\epsilon\text{Term } l \ (\text{TLabeled } l1 \ t)$ 
  |  $l1 \sqsubseteq l$       =  $\text{TLabeled } l1 \ (\epsilon\text{Term } l \ t)$ 
  | otherwise     =  $\text{TLabeled } l1 \ \text{THole}$ 
 $\epsilon\text{Term } l \ (\text{TLabel } (\text{TLabel } l1) \ t)$ 
  |  $l1 \sqsubseteq l$       =  $\text{TLabel } (\text{TLabel } l1) \ (\epsilon\text{Term } l \ t)$ 
  | otherwise     =  $\text{TLabel } (\text{TLabel } l1) \ \text{THole}$ 
...

```

The term erasure function $\epsilon\text{Term } l$ replaces terms labeled with a label $l1$ with a hole, if $l1$ cannot flow into the erasure label l . Similarly, term erasure preemptively replaces the term t in $\text{TLabel } (\text{TLabel } l1) \ t$ with a hole when $l1$ cannot flow into the erasure label l , since evaluation will lead to a labeled term. For the remaining terms, erasure is a homomorphism. Program erasure with label l of a program with current label lc erases the term of the program, if lc can flow into l ; otherwise it returns a program hole hiding from the attacker all the program configuration (i.e., both the term and the current label). Erasure of a program hole is an identity.

3.3.3.2 Simulation

In Figure 3.7 we state that for every label l , `eval` and ϵl . `eval` form a simulation. That is, evaluation of a program p and evaluation of its erased version $\epsilon l p$ cannot be distinguished after erasure. We prove this property by induction on the input program term.

Termination. Simulation (and later, noninterference) is termination-insensitive: it is defined only for executions that terminate, as indicated by the `terminates` predicate. (λ_{LIO} includes untyped lambda calculus, so λ_{LIO} programs are not strongly normalizing.) This is necessary because, for soundness, Liquid Haskell disallows non-terminating functions, like `eval`, from being lifted into refinement types. To lift `eval` in the logic we constrained it to only be called on terminating programs. To do so, we defined two logical, uninterpreted functions.

```
measure terminates :: Program l → Bool
measure evalSteps  :: Program l → Int
```

We use a refinement-type precondition to prescribe that `eval` is only called on programs p that satisfy the `terminates` predicate, and prove termination of `eval` by checking that the steps of evaluation (`evalSteps p`) are decreasing at each recursive call.

```
eval :: Label l ⇒ p:{Program l | terminates p} → Program l / [
  evalSteps p]
```

While the functions `terminates` and `evalSteps` cannot be defined as Haskell functions, we can instead *axiomatize* properties that are true under the assumption of

termination. In particular,

- if a program terminates, so do its subprograms, and
- if a program terminates, its evaluation steps are strictly smaller than those of its subprograms.

To express these properties, we define axioms involving these functions in refinements for each source program construct. For instance, the following assumption (encoded as a Haskell function) handles bind terms:

```
assume evalStepsBindAxiom :: lc:l → db:DB l → t1:Term l
  → t2:{Term l | terminates (Pg lc db (TBind t1 t2)) } →
{ (evalSteps (Pg lc db t1) < evalSteps (Pg lc db (TBind t1 t2)))
  && (0 <= evalSteps (Pg lc db t1))
  && (terminates (Pg lc db t1))} }
evalStepsBindAxiom _ _ _ _ = ()
```

Here, `evalStepsBindAxiom` encodes that if the program `Pg lc db (TBind t1 t2)` terminates, then so does `Pg lc db t1` with fewer evaluation steps. This assumption is required to prove simulation in the inductive case of the `TBind`, since we need to

- apply the simulation lemma for the `Pg lc db t1` program, thus we need to know that it terminates; and
- prove that the induction is well founded, which we do by proving that the evaluation step counts of each subprogram are a decreasing natural number.

3.3.3.3 Noninterference

The noninterference theorem states that if two terminating λ_{LIO} programs `p1` and `p2` are equal after erasure with label `l`, then their evaluation is also equal after

erasure with label `l`. As with simulation, noninterference is termination insensitive—potentially diverging programs could violate noninterference.

We express the noninterference theorem as a refinement type.

```
nonInterference :: Label l => l:l
  → p1:{Program l | terminates p1 } → p2:{Program l | terminates p2 }
  → { ε l p1 == ε l p2 } → { ε l (eval p1) == ε l (eval p2) }
```

The proof proceeds by simple rewriting using the simulation property at each input program and the low equivalence precondition.

```
nonInterference l p1 p2 lowEquivalent
=   ε l (eval p1)           ? simulation l p1
==. ε l (eval (ε l p1)) ? lowEquivalent
==. ε l (eval (ε l p2)) ? simulation l p2
==. ε l (eval p2)
*** QED
```

The body of `nonInterference` starts from the left hand side of the equality and, using equational reasoning and invocation of the `lowEquivalent` and the `simulation` theorem on the input programs `p1` and `p2`, reaches the right hand side of the equality. As explained in [3.3.1](#) the proof combinator `x ? p` returns its first argument and extends the SMT environment with the knowledge of the theorem `p`. The proof combinator `x ==. y = y` equates its two arguments and returns the second argument to continue the equational steps. Finally, `x $***$ QED = ()` casts its first argument into `unit`, so that the equational proof returns a `unit` type.

3.4 Label-based Security for Database Operations

In this section we extend λ_{LIO} with support for databases with label-based policies. We call the extended calculus λ_{LWeb} . In § 3.4.1, we define a database that stores rows with three values: a key, a first field with a static label, and a second field whose label is a function of the first field. This simplification of the full generality of **LWeb**'s implementation (which permits any field to be a label) captures the key idea that fields can serve as labels for other fields in the same row, and fields that act as labels must be labeled as well. In § 3.4.2 we define operations to insert, select, delete, and update the database. For each of these operations, in § 3.4.3 we define a monadic term that respects the database policies. Finally in § 3.4.4 we define erasure of the database and prove noninterference.

3.4.1 Database Definition

Figure 3.8 contains Haskell definitions used to express the semantics of database operations in λ_{LWeb} . Rather than having concrete syntax (e.g., as in Figure 3.3) for database definitions, in our formalization we assume that databases are defined directly in the semantic model.

A database `DB 1` maps names (`Name`) to tables (`Table 1`). A table consists of a policy (`TPolicy 1`) and a list of rows (`[Row 1]`). Each row contains three terms: the key and two values. We limit values that can be stored in the database to basic terms such as unit, integers, label values, etc. This restriction is expressed by predicate `isDBValue`. Labeled terms are not permitted—labels of stored data are specified us-

ing the table policy. In § 3.4.4 we define erasure of the database to replace values with holes, thus `isDBValue` should be true for holes too, but is false for any other term.

```

isDBValue :: Term l → Bool
isDBValue THole      = True
isDBValue (TInt _) = True
isDBValue TUnit      = True
isDBValue (TLabel _) = True
isDBValue _          = False

```

We define the refinement type alias `DBTerm` to be terms refined to satisfy the `isDBValue` predicate and define rows to contain values of type `DBTerm`.

Table policy. The table policy `TPolicy l` defines the security policy for a table. The field `tpTableLabel` is the label required to access the length of the table. The field `tpLabelField1` is the label required to access the first value stored in each row of the table. This label is the same for each row and it is refined to flow into the `tpTableLabel`. The field `tpLabelField2` defines the label of the second value stored in a row as a function of the first. Finally, the field `tpFresh` is used to provide a unique term key for each row. The term key is an integer term that is increased at each row insertion.

Helper functions. For each field of `TPolicy`, we define a function that given a table accesses its respective policy field.

```

labelT   t = tpTableLabel (tpolicy t)
labelF1  t = tpLabelField1 (tpolicy t)
labelF2  t v = tpLabelField2 (tpolicy t) v
freshKey t = tpFresh (tpolicy t)

```

We use the indexing function `db!!n` to lookup the table named `n` in the database.

```

(!!) :: DB l → Name → Maybe (Table l)

```

3.4.2 Querying the Database

Predicates. We use predicates to query database rows. In the `LWeb` implementation, predicates are written in a domain-specific query language, called `lsq1`, which the `LWeb` compiler can analyze. Rather than formalizing that query language in λ_{LWeb} , we model predicates abstractly using the following datatype:

```
data Pred = Pred { pVal :: Bool , pArity :: { i:Int | 0 <= i <= 2 } }
```

Here, `pVal` represents the outcome of evaluating the predicate on an arbitrary row, and `pArity` represents which of the row's fields were examined during evaluation. That is, a `pArity` value of 0, 1, or 2, denotes whether the predicate depends on (i.e., computes over) none, the first, or both fields of a row, respectively.

Then, we define a logical *uninterpreted* function `evalPredicate` that evaluates the predicate for some argument of type `a`:

```
measure evalPredicate :: Pred → a → Bool
```

We define a Haskell (executable) function `evalPredicate` and use an axiom to connect it with the synonymous logical uninterpreted function [134]:

```
assume evalPredicate :: p:Pred → x:a → {v:Bool | v == evalPredicate p x}  
evalPredicate p x = pVal p
```

This way, even though the Haskell function `evalPredicate p x` returns a constant boolean ignoring its argument `x`, the Liquid Haskell model assumes that it behaves as an uninterpreted function that does depend on the `x` argument (with dependencies assumed by the `pArity` definition).

Primitive queries. It is straightforward to define primitive operators that manipulate the database but do not perform IFC checks. We define operators to insert, delete, select, and update databases.

```
(+)= :: db:DB l → n:Name → r:Row l → DB l
      -- insert
(=?=) :: db:DB l → n:Name → p:Pred → Term l
      -- select
(-=) :: db:DB l → n:Name → p:Pred → DB l
      -- delete
(:=) :: db:DB l → n:Name → p:Pred → v1:DBTerm l → v2:DBTerm l → DB
      l -- update
```

Insert: `db += n r` inserts the row `r` in the `n` table in the database and increases `n`'s unique field.

Select: `db ?= n p` selects all the rows of the `n` table that satisfy the predicate `p` as a list of labeled terms.

Delete: `db -= n p` deletes all the rows of the `n` table that satisfy the predicate `p`.

Update: `db := n p v1 v2` updates each row with key `k` of the `n` table that satisfies the predicate `p` with `Row k v1 v2`.

Next we extend the monadic programs of § 3.3 with database operations to define monadic query operators that enforce the table and field policies.

3.4.3 Monadic Database Queries

3.4.3.1 Syntax

Figure 3.9 defines λ_{LWeb} 's syntax as an extension of λ_{LIO} . Programs are extended to carry the state of the database. Erasure of a program at an observation level 1 leads to a `PgHole` that now carries a database erased at level 1. Erasure is defined in § 3.4.4; here we note that preserving the database at program erasure is required since even though the result of the program is erased, its effects on the database persist. For instance, when evaluating `TBind t1 t2` the effects of `t1` on the database affect computing `t2`.

Terms are extended with monadic database queries. `TInsert n (TLabelled l1 v1) (TLabelled l2 v2)` inserts into the table `n` database values `v1` and `v2` labeled with `l1` and `l2`, respectively. `TSelect n p` selects the rows of the table `n` that satisfy the predicate `p`. `TDelete n p` deletes the rows of the table `n` that satisfy the predicate `p`. Finally, `TUpdate n p (TLabelled l1 v1) (TLabelled l2 v2)` updates the fields for each row of table `n` that satisfies the predicate `p` to be `v1` and `v2`, where the database values `v1` and `v2` are labeled with `l1` and `l2`, respectively.

3.4.3.2 Semantics

Figure 3.10 defines the operational semantics for the monadic database queries in λ_{LWeb} . Before we explain the evaluation rules, note that both insert and update attempt to insert a labeled value `TLabelled li vi` in the database, thus `vi` should

be a value, and unlabeled, i.e., satisfy the `isDBValue` predicate.³ In the `LWeb` implementation we use Haskell’s type system to enforce this requirement. In λ_{LWeb} , we capture this property in a predicate ζ that constrains labeled values in insert and update to be database values:

```

 $\zeta$  :: Program l  $\rightarrow$  Bool
 $\zeta$  (Pg _ _ t) =  $\zeta$ Term t

 $\zeta$ Term :: Term l  $\rightarrow$  Bool
 $\zeta$ Term (TInsert _ (TLabelled _ v1) (TLabelled _ v2)) = isDBValue v1 &&
  isDBValue v2
 $\zeta$ Term (TUpdate _ _ (TLabelled _ v1) (TLabelled _ v2)) = isDBValue v1 &&
  isDBValue v2
...

```

We specify that `eval` is only called on well-structured programs, i.e., those that satisfy ζ . For terms other than insert and update, well-structuredness is homomorphically defined. Restricting well-structuredness to permit only database values, as opposed to terms that eventually evaluate to database values, was done to reduce the number of cases for the proof, but does not remove any conceptual realism.

Insert. Insert attempts to insert a row with values `v1` and `v2`, labeled with `l1` and `l2` respectively, in the table `n`. To perform the insertion we check that

1. the table named `n` exists in the database, as table `t`.
2. `l1` can flow into the label of the first field of `t`, since the value `v1` labeled with `l1` will write to the first field of the table.
3. `l2` can flow into the label of the second field of `t`, as potentially determined by

³We could allow inserting unlabeled terms, the label for which is just the current label. Explicit labeling is strictly more general.

the first field v_1 (i.e., per $\text{labelF2 } \tau \ v_1$).

4. the current label l can flow to the label of the table, since insert changes the length of the table.

If all these checks succeed, we compute a fresh key $k = \text{freshKey } \tau$, insert the row $\text{Row } k \ v_1 \ v_2$ into the table n , and return the key. If any of the checks fail we return an exception and leave the database unchanged.

Either way, we raise the current label l by joining it with l_1 . This is because checking $l_2 \sqsubseteq \text{labelF2 } \tau \ v_1$ requires examining v_1 , which has label l_1 . That this check succeeds can be discerned by whether the key is returned; if the check fails an exception is thrown, potentially leaking information about v_1 . This subtle point was revealed by the formalization: Our original implementation failed to raise the current label properly.

Select. Select only checks that the table n exists in the database, returning an exception if it does not. If the table n is found as the table τ , then we return the term $\text{db } ?= \ n \ p$ that contains a list of all rows of τ that satisfy the predicate p , leaving the database unchanged. The current label is raised to include the label of the table $\text{labelT } \tau$ since on a trivially true predicate, all the table is returned, thus the size of the table can leak. We raise the current label with the label of the predicate p on the table τ that intuitively permits reading all the values of τ that the predicate p depends on. We define the function $\text{labelPred } p \ \tau$ that computes the label of the predicate p on the table τ .

```

labelPred  :: (Label l) => Pred -> Table l -> l
labelPred p (Table tp rs)
  | pArity p == 2 = foldl (⊔) (labelF1 tp) [labelF2 tp v1 | Row _ v1 _ <-
    rs]
  | pArity p == 1 = labelF1 tp
  | otherwise     = ⊥

```

If the predicate p depends on both fields, then its predicate is the join of the label of the first field and all the labels of the second fields. If p only depends on the first field, then the label of the predicate p is the label of the first field. Otherwise, p depends on no fields and its predicate is \perp .

Note that the primitive selection operator $\text{db } ?= \text{ n } p$ returns labeled terms protected by the labels returned by the `labelF1` and `labelF2` functions. Since terms are labeled, `select` does not need to raise the current label to protect values that the predicate p does not read.

Delete. Deletion checks that the table named n exists in the database as t and that the current label joined with the label of the predicate p on the table t can flow into the label of the table t , since `delete` changes the size of the table. If both checks succeed, then database rows are properly deleted. The current label is raised with the “read label” of the predicate p on the table t that intuitively gives permission to read the label of the predicate p on the same table. The function `labelRead p t` computes the read label of the predicate p on the table t to be the label required to read `labelPredRow p t`, i.e., equal to the label of the first field, if the predicate depends on the second field and bottom otherwise.

```

labelRead :: (Label l) => Pred -> Table l -> l
labelRead p t = if pArity p == 2 then labelF1 t else ⊥

```

Note that $\text{labelRead } p \ t$ always flows into $\text{labelPred } p \ t$, thus the current label is implicitly raised to this read label. When the runtime checks of `delete` fail we return an exception and the database is not changed. If the table n was found in the database, the current label is raised, even in the case of failure, since the label of the predicate was read.

Update. Updating a table n with values $v1$ and $v2$ on a predicate p can be seen as a select-delete-insert operation. But, since the length of the table is not changing, the check that the current label can flow to the label of the table is omitted. Concretely, update checks that

1. the table named n exists in the database, as table t ,
2. $1 \sqcup 11 \sqcup \text{labelPred } p \ t$ can flow into the label of the first field of t , since the value $v1$ labeled with 11 will write on the first field of the table and whether this write is done or not depends on the label of the predicate p as a hole,
3. $1 \sqcup 12 \sqcup \text{labelPred } p \ t$ can flow into the label of the second field of t when the first field is $v1$.

If these checks succeed, then `unit` is returned, the database is updated, and the current label is raised to all the labels of values read during the check, i.e., $11 \sqcup \text{labelF1 } t$. If the checks fail then we return an exception and the database is not updated.

In both cases, the current label is raised by joining with the table label, i.e., $1' = \dots \sqcup \text{labelT } t$. This is because the last check depends on whether the

table is empty or not, and its success can be discerned: if it succeeds, then unit is returned. Interestingly, our original implementation failed to update the current label in this manner. Doing so seemed intuitively unnecessary because an update does not change the table length.

3.4.4 Noninterference

As in § 3.3 to prove noninterference we prove the simulation between `eval` and ϵ 1 . `eval` for λ_{LWeb} programs. Figure 3.11 extends erasure to programs and databases. Erasure of programs is similar to § 3.3 but now we also erase the database. Erasure of a database recursively erases all tables. Erasure of a table removes all of its rows if the label of the table cannot flow into the erasing label, thus hiding the size of the table. Otherwise, it recursively erases each row. Erasure of a row respects the dynamic labels stored in the containing table’s policy. Erasure of a row replaces *both* fields with holes if the label of the first field cannot flow into the erasing label, since the label of the second field is not visible. If the label of the second field cannot flow into the erasing label, it replaces only the second field with a hole. Otherwise, it erases both fields.

With this definition of erasure, we prove the simulation between `eval` and ϵ 1 . `eval`, and with this, noninterference. The refinement properties in the database definition of fig. 3.8 are critical in the proof, as explained below.

Well-structured programs. The simulation proof assumes that the input program is well-structured, i.e., satisfies the predicate ς as defined in § 3.4.3.2, or equivalently

evaluation only inserts values that satisfy the `isDBValue` property. To relax this assumption, an alternative approach could be to check this property at runtime, just before insertion of the values. But, this would break simulation: `TInsert n (TLabelled l1 v1) t` will fail if `v1` is not a database value, but its erased version can succeed if `v1` is erased to a hole (when `l1` cannot flow into the erase label). Thus, the `isDBValue` property cannot be checked before insertion and should be assumed by evaluation. In the implementation this safety check is enforced by Haskell’s type system.

Database values. Simulation of the delete operation requires that values stored in the database must have identity erasure, e.g., cannot be labeled terms. Thus, we prove that all terms that satisfy `isDBValue` also have erasure identity. We do this by stating the property as a refinement on term erasure itself.

$$\epsilon\text{Term} :: \text{Label } l \Rightarrow l \rightarrow i:\text{Term } l \rightarrow \{o:\text{Term } l \mid \text{isDBValue } i \Rightarrow \text{isDBValue } o \}$$

In the delete proof, each time a database term is erased, the proof identity `εTerm l v == v` is immediately available.

Note on refinements. The type `DBTerm l` is a type alias for `Term l` with the attached refinement that the term is a database value. A `DBTerm l` *does not carry* an actual proof that it is a database value. Instead, the refinement type that the term satisfies the `isDBValue` property is statically verified during type checking. As a consequence, comparison of two `DBTerms` does not require proof comparison. At the same time, verification can use the `isDBValue` property. For instance, when opening

a row `Row k v1 v2`, we know that `isDBValue v1` and by the type of term erasure, we know that for each label `l`, `εTerm l v1 == v1`.

3.5 Liquid Haskell for Metatheory

Liquid Haskell was originally developed to support lightweight program verification (e.g., out-of-bounds indexing). The formalization of `LWeb` in Liquid Haskell, presented in § 3.3 and § 3.4, was made possible by recent extensions to support general theorem proving [134]. Our proof of noninterference was a challenging test of this new support, and constitutes the first advanced metatheoretical result mechanized in Liquid Haskell.⁴

The trusted computing base (TCB) of any Liquid Haskell proof relies on the correct implementation of several parts. In particular, we trust that

1. the GHC compiler correctly desugars the Haskell code to the core language of Liquid Haskell,
2. Liquid Haskell correctly generates the verification conditions for the core language, and
3. the SMT solver correctly discharges the verification conditions.

We worked on the noninterference proof, on and off, for 10 months. The proof consists of 5,447 lines of code and requires about 5 hours to be checked. For this proof in particular, we (naturally) trust all of our semantic definitions, and also two

⁴<https://github.com/plum-umd/lmonad-meta>

explicit assumptions, notably the axiomatization of termination and modeling of predicates. These were discussed respectively in § 3.3.3.2 and § 3.4.2.

Carrying out the proof had a clear benefit: As mentioned in § 3.4.3, we uncovered two bugs in our implementation. In both cases, `LWeb` was examining sensitive data when carrying out a security check, but failed to raise the current label with the label of that data. Failure of the mechanized proof to go through exposed these bugs.

The rest of this section summarizes what we view as the (current) advantages and disadvantages of using Liquid Haskell as a theorem prover compared to other alternatives (e.g., Coq and F-star [138]), expanding on a prior assessment [135].

3.5.1 Advantages

As a theorem proving environment, Liquid Haskell offers several advantages.

General purpose programming language. The Liquid Haskell-based formal development is, in essence, a Haskell program. All formal definitions (presented in § 3.3 and § 3.4) and proof terms (e.g., illustrated in § 3.3.3) are Haskell code. Refinement types define lemmas and theorems, referring to these definitions. In fact, some formal definitions (e.g., the `Label` class definition) were taken directly from the implementation. As the main developer of the proof, I am a Haskell programmer, thus I did not need to learn a new programming language (e.g., Coq) to develop the formal proof. During development we used Haskell’s existing development tools, including the build system, test frameworks, and deployment support (e.g., Travis

integration).

SMT automation. Liquid Haskell, like Dafny [139] and F-star [138], uses an SMT solver to automate parts of the proof, especially the ones that make use of boolean reasoning, reducing the need for manual case splitting. For example, proving simulation for row updates normally proceeds by case splitting on the relative can-flow-to relation between four labels. The SMT automates the case splitting.

Semantic termination checking. To prove termination of a recursive function in Liquid Haskell it suffices to declare a non negative integer value that is decreasing at each recursive call. The `LWeb` proof was greatly simplified by the semantic termination checker. In a previous Coq LIO proof [140], the evaluation relation apparently requires an explicit *fuel* argument to count the number of evaluation steps, since the evaluation function (the equivalent to that in fig. 3.6) does not necessarily terminate. In our proof, termination of evaluation was axiomatized (per § 3.3.3.2), which in practice meant that the evaluation steps were counted only in the logic and not in the definition of the evaluation function.

Intrinsic and extrinsic verification. The Liquid Haskell proving style allows us to conveniently switch between (manual) extrinsic and (SMT automated) intrinsic verification. Most of the `LWeb` proof is extrinsic, i.e., functions are defined to state and prove theorems about the model. In few cases, intrinsic specifications are used to ease the proof. For instance, the refinement type specification of `εTerm`, as described in 3.4.4, intrinsically specifies that erasure of `isDBValue` terms returns terms that

also satisfy the `isDBValue` predicate. This property is automatically proven by the SMT without cluttering the executable portion of the definition with proof terms.

3.5.2 Disadvantages

On the other hand, Liquid Haskell has room to improve as a theorem proving environment, especially compared to advanced theorem provers like Coq.

Unpredictable verification time. The first and main disadvantage is the unpredictability of verification times, which owe to the invocation of an SMT solver. One issue we ran across during the development of our proof is that internal transformations performed by `ghc` can cause massive blowups. This is because Liquid Haskell analyzes Haskell’s intermediate code (`CoreSyn`), not the original source. As an example of the problem, using `|x,y` instead of the logical `| x && y` in function guards leads to much slower verification times. While the two alternatives have exactly the same semantics, the first case leads to exponential expansion of the intermediate code.

Lack of tactics. Liquid Haskell currently provides no tactic support, which could simplify proof scripts. For example, we often had to systematically invoke label laws (fig. 3.4) in our proofs, whereas a proof tactic to do so automatically could greatly simplify these cases.

General purpose programming language. Liquid Haskell, developed for light-weight verification of Haskell programs, lacks various features in verification-specific sys-

tems, such as Coq. For example, Liquid Haskell provides only experimental support for curried, higher-order functions, which means that one has to inline higher order functions, like `map`, `fold`, and `lookup`. There is also no interactive proof environment or (substantial) proof libraries.

In sum, our `LWeb` proof shows that Liquid Haskell can be used for sophisticated theorem proving. We are optimistic that current disadvantages can be addressed in future work. We have addressed some of the disadvantages already, as described in Chapter 4.

3.6 Implementation

`LWeb` has been available online since 2016 and consists of 2,664 lines of Haskell code.⁵ It depends on our base `LMonad` package that implements the `LMonadT` monad transformer and consists of 345 lines of code.⁶ `LWeb` also imports `Yesod`, a well established, external Haskell library for type-safe, web applications. This section explains how the implementation extends the formalization, and then discusses the trusted computing base.

3.6.1 Extensions

The `LWeb` implementation generalizes the formalization of Sections 3.3 and 3.4 in several ways.

⁵<https://github.com/jprider63/lmonad-yesod>

⁶<https://github.com/jprider63/lmonad>

Clearance label. The implementation supports a *clearance* label, described in § 3.2.1. Intuitively, the clearance label limits how high the current label can be raised. If the current label ever exceeds the clearance label, an exception is thrown. This label is not needed to enforce noninterference, but serves as an optimization, cutting off transactions whose current label rises to the point that they are doomed to fail. Adding checks to handle the clearance was straightforward.

Full tables and expressive queries. As first illustrated in § 3.2.3, tables may have more than two columns, and a column's label can be determined by other various fields in the same row. The labels of such *dependency fields* must be constant, i.e., not determined by another field, and flow into the table label (which also must be constant). A consequence of this rule is that a field's label cannot depend on itself. Finally, values stored in tables instantiate Yesod's `PersistField` type class. The implementation uses only the predefined instances including `Text`, `Bool`, `Int` but critically, does not define a `PersistField` for labeled values. `LWeb` enforces these invariants at compile time via Haskell type checking and when preprocessing table definitions. `LWeb` rewrites queries to add labels to queried results.

We have implemented database operations beyond those given in § 3.4, to be more in line with typical database support. Some of these operations are simple variations of the ones presented. For example, `LWeb` allows for variations of `update` that only update specific fields (not whole rows). `LWeb` implements these basic queries by wrapping `Persistent` [133], Yesod's database library, with the derived IFC checks. To support more advanced queries, `LWeb` defines an SQL-like domain-

specific language called `lsql`. `lsql` allows users to write expressive SQL queries that include inner joins, outer joins, `where` clauses, orderings, limits, and offsets. Haskell expressions can be included in queries using anti-quotation. At compile-time, `LWeb` parses `lsql` queries using quasi-quotation and Template Haskell [124]. It rewrites the queries to be run using Esqueleto [123], a Haskell library that supports advanced database queries. As part of this rewriting, `LWeb` inserts IFC checks for queries based on the user-defined database policies. We show several examples of `lsql` queries in § 3.7.

Optimizations. Sometimes a label against which to perform a check is derived from data stored in every row. Retrieving every row is especially costly when the query itself would retrieve only a fraction of them. Therefore, when possible we compute an upper bound for such a label. In particular, if a field is fully constrained by a query’s predicate, we use the field’s constrained value to compute any dependent labels. When a field is not fully constrained, we conservatively set dependent labels to \top . Suppose we wish to query the `Friends` table from fig. 3.3, retrieving all rows such that `user1 == 'Alice'` and `date < '2000-01-01'`. The confidentiality portion of `user1`’s label is \perp , but that portion of `date`’s is computed from `user1` \sqcap `user2`. Since `user1` is always `'Alice'` we know the computed label is $\bigsqcup_l \text{Alice} \sqcap l$ for all values `user2 = l` in the database. In this case, we can bound l as \top , and thus use label `Alice`, since it is equivalent to `Alice` $\sqcap \top$. While this bound is technically conservative, in practice we find it makes policy sense. In this example, if the `user2` field can truly vary arbitrarily then $\bigsqcup_l l$ will approach \top .

Declassification. **LWeb** supports forms of *declassification* [141] for cases when the IFC lattice ordering needs to be selectively relaxed. These should be used sparingly (and are, in our BIBIFI case study), as they form part of the trusted computing base, discussed below.

Row ordering. As a final point, we note that our formalization models a database as a list of rows; insertion (via `+=`) simply appends to the list, regardless of the contents of a row. As such, *row ordering* does not depend on the database’s contents and thus reveals nothing about them (it is governed only by the table label). In the implementation, advanced operations may specify an ordering. **LWeb** prevents leaks in this situation by raising the current label with the label of fields used for sorting. If a query does not specify an ordering, **LWeb** takes no specific steps. However, ordering on rows is undefined in SQL, so a backend database could choose to order them by their contents, and thus potentially leak information in a query’s results. In our experience with PostgreSQL, default row ordering depends on when values are written and is independent of the data in the table.

3.6.2 Trusted Computing Base

A key advantage of **LWeb** is that by largely shifting security checks from the application into the **LWeb** IFC framework, we can shrink an application’s trusted computing base (TCB). In particular, for an application that uses **LWeb**, the locus of trust is on **LWeb** itself, which is made (more) trustworthy by our mechanized noninterference proof. A few parts of the application must be trusted, nevertheless.

First, all of the policy specifications are trusted. The policy includes the labels on the various tables and the labels on data read/written from I/O channels. Specifying the latter requires writing some trusted code to interpret data going in or out. For example, in a multi-user application like BIBIFI, code performing authentication on a particular channel must be trusted (§ 3.7.2).

Second, any uses of declassification are trusted, as they constitute local modifications to policy. One kind of declassification can occur selectively on in-application data [15]. We give an example in § 3.7.4. Another kind of declassification is to relax some security checks during database updates. The update query imposes strong runtime checks, e.g., that the label of the predicate should flow into the updated fields as formalized in § 3.4. LWeb provides an unsound update alternative (called `updateDeclassifyTCB`) that ignores this specific check.

3.7 The BIBIFI Case Study

As a real case study, we integrated LWeb into BIBIFI's infrastructure (chapter 2). The BIBIFI web application stores personal information of contestants and has multiple principals. This makes it an ideal use case to demonstrate the effectiveness of LWeb in enforcing complex confidentiality and integrity policies.

3.7.1 BIBIFI Labels

BIBIFI labels include all entities that operate in the system. The `Principal` data type, defined in fig. 3.12, encodes all such entities, including the system itself,

the administrator, users, teams, and judges. Each of these entities is treated as a security level. For instance a policy can encode that data written by a user with id 5, can get protected at the security level of this specific user, so that only he or she can read this data. A more flexible policy encodes that the system administrator can read data written by each user. To encode such policies, we use disjunction category labels (`DCLabel`) [122] to create a security lattice out of our `Principals`. In fig. 3.12 we define `BBFLabel` as the `DCLabel Principal` data type that tracks the security level of values as they flow throughout the web application and database.

3.7.2 Users and Authentication

Users' personal information is stored in the BIBIFI database. Figure 3.14 shows the `User` table with the basics: a user's account id, email address, and whether they have administrator privileges. The label for the `email` field refers to `Id` in its label: This is a shorthand for the key of the present table. The label says that a user can read and write their emails, while the administrator can read every user's email. The label for the `admin` field declares that it may be written by the administrator and read by anyone.

Additional private information is stored in the `UserInfo` table, shown in Figure 3.14, including a user's school, age, and professional experience. The `user` field of this table is a foreign key to the `User` table, as indicated by its type `UserId` (see § 3.2.2). Each of the remaining fields is protected by this field, in part: users can read and write their own information while administrators can read any users'

information.

The current label is set by the code trusted to perform authentication. If a user is not logged in, the current label is set to $\langle \perp, \top \rangle$: the confidentiality label is the upper bound on data read so far (i.e., none, so \perp), and the integrity label is the level of least trust (i.e., \top) for writing data. After authenticating, most users will have the label $\langle \perp, \text{PUser } \text{userId} \rangle$, thus lowering the integrity part (thus increasing the level of trust) to the user itself. Users who are also administrators will have current label lowered further to $\langle \perp, \text{PUser } \text{userId} \sqcap \text{PAdmin} \rangle$. This is shown in the following code snippet. It determines the logged in user via `requireAuth`, and then adds administrator privileges if the user has them (per `userAdmin`).

```
(Entity userId user) ← requireAuth
let userLabel = dcIntegritySingleton (PrincipalUser userId)
lowerLabelTCB $ if userAdmin user
                then userLabel  $\sqcap$  dcIntegritySingleton PrincipalAdmin
                else userLabel
```

The clearance is also set using trusted functions during authentication. For example, for an administrator it would be $\langle \text{PUser } \text{userId} \sqcup \text{PAdmin}, \top \rangle$.

3.7.3 Opening the Contest

To start a contest, administrators write announcements that include information like instructions and problem specifications. It is important that only administrators can post these announcements. Announcements are stored in the database, and their (simplified) table definition is shown in fig. 3.15. The `Announcement` table has two `Text` fields corresponding to an announcement's title and content. Only

administrators can author announcements.

An earlier version BIBIFI relied on manual access control checks rather than monadic `LMonad` enforcement of security. The old version had a security bug: it failed to check that the current user was an administrator when posting a new announcement. Here is a snippet of the old code.

```
postAddAnnouncementR :: Handler Html
postAddAnnouncementR = do
  ((res, widget), enctype) ← runFormPost postForm
  case res of ...
    FormSuccess (FormData title markdown) → do
      runDB (insert (Announcement title markdown))
      redirect AdminAnnouncementsR
```

This function parses POST data and inserts a new announcement. The user is never authenticated, so anyone can post new announcements and potentially deface the website. In the IFC version of the website, the database insertion fails for unauthorized or unauthenticated users as the integrity part of the current label is not sufficiently trusted (the label does not flow into `PAdmin`).

3.7.4 Teams and Declassification

To participate in a contest, a user must join a team. The teams and their members are stored in the eponymous tables of fig. 3.15. Teams serve as another principal in the BIBIFI system and BIBIFI defines a TCB function that appropriately authenticates team members similarly to users (§ 3.7.2), authorizing a team member to read and write data labeled with their team.

BIBIFI uses declassification (as discussed in 3.6.2) to allow team members

to send email messages to their team. The policy on the email field of the `User` table states that only the user or an administrator can read the email address, so BIBIFI cannot give a user's email address to a teammate. Instead, the function `sendEmailToTeam` below sends the email on the teammate's behalf using declassification.

```
sendEmailToTeam :: TeamId → Email → LHandler ()
sendEmailToTeam tId email = do
  protectedEmails ← runDB [lsql| pselect User.email from User inner
    join TeamMember on TeamMember.user == User.id where
      TeamMember.team == #{tId} |]
  mapM_ (\protectedEmail → do
    address ← declassifyTCB protectedEmail
    sendEmail address email
  ) protectedEmails
```

The function `sendEmailToTeam`'s parameters are the team identifier and an email return address. It queries the database for the (labeled) email addresses of the team's members, using `lsql` (see § 3.2.3 and § 3.6.1). The `sendEmailToTeam` function maps over each address, declassifying it via `declassifyTCB`, so that the message can be sent to the address. The `declassifyTCB` function takes a labeled value and extracts its raw value, *ignoring label restrictions*. This is an unsafe operation that breaks noninterference, so the programmer must be careful with its use. Here for example, the function is careful not to reveal the email address to the sender but only use it to send the email.

3.7.5 Breaks and Advanced Queries

During the second round of the BIBIFI contest, teams submit breaks, i.e., test cases that attack another team's submission. After a break is pushed to a registered git repository, BIBIFI's backend infrastructure uploads it to a virtual machine and tests whether the attack succeeds. Results are stored in the `BreakSubmission` table of fig. 3.16, which has fields for the attacking team, the target team, and the (boolean) result of the attack. The integrity label for the result field is `PSys` since only the backend system can grade an attack. The confidentiality label is `PAdmin` \sqcap `PTeam attackerId` \sqcap `PTeam targetId` since administrators, the attacker team, and the target team can see the result of an attack.

BIBIFI has an administration page that lists all break submissions next to which team was attacked. This page's contents are retrieved via the following inner join.

```
runDB $ [lsq| select BreakSubmission.*, Team.name from BreakSubmission
  inner join Team on BreakSubmission.target == Team.id where Team.
  contest == #{contestId} order by BreakSubmission.id desc |]
```

This query performs a join over the `BreakSubmission` and `Team` tables, aligning rows where the target team equals the team's identifier. In addition, it filters rows to the specified contest identifier and orders results by the break submission identifiers.

3.8 Experimental Evaluation

To evaluate `LWeb` we compare the BIBIFI implementation that uses `LMonad` with our initial BIBIFI implementation that manually checked security policies via access control. We call this initial version the *vanilla implementation*. Transitioning from the vanilla to the `LWeb` implementation reduced the trusted computing base (TCB) but imposed a modest runtime overhead.

3.8.1 Trusted Computing Base of BIBIFI

The implementation of the BIBIFI application is currently 11,529 lines of Haskell code. 80 of these lines invoke trusted functions (for authentication or declassification, see § 3.6.2). `LWeb`'s library is 3,009 lines of trusted code. The vanilla implementation is several years old, with 7,367 LOC; there is no IFC mechanism so the whole codebase is trusted. Switching from the vanilla to the `LWeb` implementation only added 151 LOC. The size of the TCB is now 21% of the codebase; considering only the code of the BIBIFI web application (and not `LWeb` too), 1% of the code is trusted.

3.8.2 Running Time Overhead

We measured the query latency, i.e., the response time (in milliseconds) of HTTP requests, for both the `LWeb` and the vanilla implementation. Measurements were performed over `localhost` and we ran 100 requests to warm up. We present the mean, standard deviation, and tail latency over 1,000 trials, as well as the response

Handler	Verb	Vanilla Latency			LWeb Latency			Size (kB)	Overhead
		Mean (ms)	SD (ms)	Tail (ms)	Mean (ms)	SD (ms)	Tail (ms)		
/announcements	GET	4.646	1.215	16	5.529	1.367	20	18.639	19.01%
/announcement/update	POST	9.810	2.600	54	11.395	3.054	52	0.706	16.16%
/profile	GET	2.116	0.512	6	2.167	0.550	6	7.595	2.41%
/buildsubmissions	GET	6.364	1.251	17	7.441	1.706	22	14.434	16.92%
/buildsubmission	GET	28.633	2.772	52	30.570	3.477	75	9.231	6.76%
/breaksubmissions	GET	41.758	7.826	81	49.218	11.679	90	60.044	17.86%
/breaksubmission	GET	4.070	0.538	9	4.923	0.509	9	6.116	20.96%

Table 3.1: Latency comparison between the **Vanilla** and **LWeb** implementations of the BIBIFI application. The mean, standard deviation, and tail latency in milliseconds over 1,000 trials are presented. In addition, the response size in kilobytes and the overhead of LWeb are shown.

size (in kilobytes) and the overhead of **LWeb** over the vanilla implementation. Table 3.1 summarizes this comparison. The server used for benchmarking runs Ubuntu 16.04 with two Intel(R) Xeon(R) E5-2630 2.60GHz CPUs and 64GB of RAM. PostgreSQL 9.5.13 is run locally as the database backend. We used ApacheBench to perform the measurements with a concurrency level of one. Here is a sample invocation of `ab`:

```
ab -g profile_lweb.gp -n 1000 -T "application/x-www-form-urlencoded; charset=UTF-8" -c 1 -C _SESSION=... http://127.0.0.1:4000/profile
```

Most of the requests are GET requests that display contest announcements, retrieve a user’s profile with personal information, get the list of a team’s submissions, and view the results of a specific submission. One POST request is measured that updates the contents of an announcement. Cookies and CSRF tokens were explicitly defined so that a user was logged into the site, and the user had sufficient permissions for all of the pages.

To evaluate **LWeb**’s impact on the throughput of web applications, we conduct similar measurements except we rerun `ab` with concurrency levels of 16 and 32. The rest of the experimental setup matches that of the latency benchmark, including

number of requests, hardware, and handlers. Figure 3.17 shows the number of requests per second for each version of the BIBIFI web application across the various handlers.

Most of the handlers show modest overhead between the vanilla and `LWeb` versions of the website. We measure `LWeb`'s overhead to range from 2% to 21%, which comes from the IFC checks that `LWeb` makes for every database query and the state monad transformer that tracks the current label and clearance label. In practice, this overhead results in a few milliseconds of delay in response times. In most situations, this is a reasonable price to pay in order to reduce the size of the TCB and increase confidence that the web application properly enforces the user defined security policies.

3.9 Quantifying Information Flow

IFC systems like `LWeb` prevent the leakage of information in programs, but in practice programs do need to release some privileged information to users. To safely release information, we would like to define policies that specify how much information is allowed to be released. Quantifying information flow (QIF) is one approach that allows us to define such policies.

QIF measures how much information a program leaks by modeling an adversary's belief as a distribution over secret variables [142]. Initially, the adversary has a prior distribution that represents the probability of potential values for secret variables. After running a program and observing the result, the adversary's belief is

updated to a posterior distribution with new probabilities given the observed result. With a prior and posterior distribution, one can quantify the amount of information flow by observing the change in size of the belief distribution.

Existing tools like Prob [143] use abstract interpretation [144] to soundly approximate the posterior distribution of an adversary that observes the output of a program. Prob uses abstract domains like intervals [145], octagons [146], and polyhedra [147] as a representation for belief distributions. These abstract domains are conjunctions of linear constraints. Interval constraints have the form $a \leq X \leq b$ for program variables X and constants a and b . Octagon constraints have the form $\pm X \pm Y \leq c$ for program variables X and Y and constant c . Polyhedra constraints have the form $aX + bY \leq c$ for variables X and Y and constants a , b , and c .

We statically quantify information flow in Liquid Haskell by encoding abstract domains with Haskell types and by modeling an adversary’s belief distribution using refinement types. For example, we define intervals with the Haskell datatype `IntRange` and a `betweenInt` function that determines whether an integer is a member of the interval.

```
data IntRange = IntRange {
    lower :: Int
    , upper :: Int
}

betweenInt :: Int → IntRange → Bool
betweenInt x IntRange{..} = lower < x && x < upper
```

We build upon this to reason about more complicated types than just integers. For example, we define a ship as record with a capacity and a location.

```

data Ship = Ship {
    shipCapacity :: Int
    , shipLoc :: Loc
}

data ShipRange = ShipRange {
    shipCapacityD :: IntRange
    , shipLocD :: LocRange
}

betweenShip :: Ship → ShipRange → Bool
betweenShip Ship{..} ShipRange{..} = betweenInt shipCapacity shipCapacityD
    && betweenLoc shipLoc shipLocD

```

In addition, we define `ShipRange` as the corresponding distribution type for `Ship` that has intervals for each of the ship's fields. Again, we need a `betweenShip` function to determine whether a `Ship` falls in the `ShipRange` distribution.

With a representation for `Ship` distributions, we can reason about how queries about `Ships` leak information. Consider the following query, written as a Haskell function, that returns whether a ship is within 100 units of the coordinate (200,200) by Manhattan distance.

```

nearby :: Ship → Bool
nearby (Ship _ z) = abs (x z - x 1) + abs (y z - y 1) <= 100
    where
        1 = Loc 200 200

```

We define functions `nearbySound` and `nearbyComplete` that given a prior distribution of a `secret` ship, return the posterior distribution of the adversary for all responses to the `nearby` query.

```

{-@ nearbySound
    :: secret : Ship
    → {prior : ShipRange | betweenShip secret prior}
    → response : Bool
    → {post : ShipRange | subsetShip post prior
        && (betweenShip secret post ⇒ response == nearby secret)}

```

```

@-}
nearbySound secret (ShipRange c (LocRange (IntRange xl xu) (IntRange yl yu
  ))) True = ShipRange c (LocRange (IntRange (max 149 xl) (min 251 xu))
  (IntRange (max 149 yl) (min 251 yu)))
nearbySound secret (ShipRange c (LocRange (IntRange xl xu) (IntRange yl yu
  ))) False = ShipRange c (LocRange (IntRange xl (min xu 150)) (IntRange
  yl (min 150 yu)))

{-@ nearbyComplete
  :: secret : Ship
  → {prior : ShipRange | betweenShip secret prior}
  → response : Bool
  → {post : ShipRange | subsetShip post prior
    && (response == nearby secret ⇒ betweenShip secret post)}
@-}
nearbyComplete secret (ShipRange c (LocRange (IntRange xl xu) (IntRange yl
  yu))) True = ShipRange c (LocRange (IntRange (max 99 xl) (min xu 301)
  ) (IntRange (max 99 yl) (min 301 yu)))
nearbyComplete secret (ShipRange c loc) False = ShipRange c loc

```

The refinement types on the functions require that the resulting posterior distributions are sound and complete, respectively. Soundness means that for all ships in the posterior, the given response matches the result from the `nearby` query on the ship. Completeness is the opposite. Liquid Haskell is able to automatically prove soundness and completeness for these functions.

Non-linear Abstract Domains. Abstract domains are traditionally linear, so that tools can automatically generate and prove properties about them. While Liquid Haskell can automatically solve linear constraints, Liquid Haskell can also reason about non-linear constraints with guidance from a developer. In particular, developers can use equational reasoning to write proofs to reason about non-linear constraints. This enables more precise abstract domains, reducing the over- and under-approximations of existing abstract domains.

For example, circle domains are encoded with constraints of the form $(X-a)**2 + (Y-b)**2 \leq r**2$ for program variables X and Y and constants a , b , and r . We encode them with the `CircleRange` datatype to represent the position and radius of the circle.

```
data CircleRange = CircleRange {
  circleA :: Int
  , circleB :: Int
  , circleR :: {v:Int | v >= 0}
}
```

We mechanically verify the property that if a circle is a subset of another circle, all points in the smaller circle are in the larger circle by using equational reasoning and axiomatizing the law of cosines.

```
assume lawOfCosines :: a:Int → b:Int → {c:Int | isTriangle a b c}
→ {  sqr c <= sqr a + sqr b + 2 * a * b
    && sqr c >= sqr a + sqr b - 2 * a * b }

subsetCircleLemma :: l : Loc → c1 : CircleRange
→ {c2 : CircleRange | subsetCircle c1 c2}
→ { betweenCircle l c1 ⇒ betweenCircle l c2}
```

3.10 Related Work

`LWeb` provides end-to-end information flow control (IFC) security for webapps. Its design aims to provide highly expressive policies and queries in a way that does not compromise security, and adds little overhead to transaction processing, in both space and time. This section compares `LWeb` to prior work, arguing that it occupies a unique, and favorable, spot in the design space.

Information flow control. **LWeb** is part of a long line of work on using lattice-ordered, label-based IFC to enforce security policies in software [15, 17, 148]. Enforcement can occur either *statically* at compile-time, e.g., as part of type checking [108, 109, 149, 150] or a static analysis [113, 114, 115], or *dynamically* at run-time, e.g., via source-to-source rewriting [117, 151] or library/run-time support [16, 118, 119]. Dynamic approaches often work by rewriting a program to insert the needed checks and/or by relying on support from the hardware, operating system, or run-time. Closely related to IFC, *taint tracking* controls *data flows* through the program, rather than overall influence (which includes effects on *control flow*, i.e., *implicit* flows). Taint tracking can avoid the false positives of IFC, which often overapproximates control channels, but will also miss security violations [116].

LWeb builds on the LIO framework [16], which is a dynamic approach to enforcing IFC that takes advantage of Haskell’s static types to help localize checks to I/O boundaries. LIO’s *current label* and *clearance label* draw inspiration from work on Mandatory Access Control (MAC) operating systems [148], including Asbestos [152], HiStar [153], and Flume [154]. The baseline LIO approach has been extended in several interesting ways [155, 156, 157, 158], including to other languages [159].

The proof of security in the original LIO (without use of a database) has been partially mechanized in Coq [140], while the derivative MAC library [155] has been mechanized in Agda [160]. The MAC mechanization considers concurrency, which ours does not. Ours is the first mechanization to use an SMT-based verifier (Liquid Haskell).

IFC for database-using web applications. Several prior works apply IFC to web applications. FlowWatcher [161] enforces information flow policies within a web proxy, which provides the benefit that applications need not be retrofitted, but limits the granularity of policies it can enforce.

SeLINQ [111] is a static IFC system for F# programs that access a database via language-integrated queries (LINQ). SIF [110] uses Jif [108] to enforce static IFC-based protection for web servlets, while Swift [112] also allows client-side (Javascript) code. Unlike LWeb, these systems permit only statically determined database policies, not ones with dynamic labels (e.g., stored in the database). The latter two lack language support for database manipulation, though a back-end database can be made accessible by wrapping it with a Jif signature (which we imagine would require an SeLINQ-style static policy).

UrFlow [128] performs static analysis to prove that information flow policies are properly enforced. These policies are expressed as SQL queries over protected data and known information. Static analysis-based proofs about queries and flows impose no run-time overhead. But static analysis can be overapproximate, rejecting correct programs. Dynamic enforcement schemes do not have this issue, and LWeb’s LIO-based approach imposes little run-time overhead.

SELinks [127] enforces security policies for web applications, including ones resembling the field-dependent policies we have in LWeb. To improve performance, security policy checks were offloaded to the database as stored procedures; LWeb could benefit from a similar optimization. SELinks was originally based on a formalism called Fable [162] in which one could encode IFC policies, but this encoding

was too onerous for practical use, and not present in SELinks, which was limited to access control policies. Qapla [163] also supports rich policies, but like SELinks these focus on access control, and so may fail to plug leaks of protected data via other server state.

Jacqueline [120] uses faceted information flow control [121] to implement policy-agnostic security [164, 165] in web applications. Like LWeb, they have formalized and proved a noninterference property (but not mechanized it). Unlike LWeb that enforces IFC using the underlying LIO monad, Jacqueline at runtime explicitly keeps track of the secret and public views of sensitive values. While expressive, this approach can be expensive in both space and time: results of computations on sensitive values have up to $1.75\times$ slower running times, and require more memory. Latencies for Django and Jacqueline are around 160ms for typical requests to their benchmark application.

The system most closely related to LWeb is Hails [125, 126], which aims to enforce information flow-oriented policies in web applications. Hails is also based on LIO, and is particularly interested in confining third-party extensions (written in Safe Haskell [166]). In Hails, individual record fields can have policies determined by other data in the database, as determined by a general Haskell function provided by the programmer. Thus, Hails policies can encode LWeb policies, and more; e.g., data in one table can be used to determine labels for data in another table. Evaluating the policy function during query processing is potentially expensive. That said, according to their benchmarks, the throughput of database writes of Hails is $2\times$ faster than Ruby Sinatra, comparable to Apache PHP, and $6\times$ slow than Java

Jetty. They did not measure Hails’ overhead, e.g., by measuring the performance difference with and without policy checks.

There are several important differences between `LWeb` and Hails. First, `LWeb` builds on top of a mature, popular web framework (`Yesod`). Extracting `LIO` into `LMonad` makes it easy for `LWeb` to evolve as `Yesod` evolves. As such, `LWeb` can benefit from `Yesod`’s optimized code, bugfixes, etc. Second, `LWeb`’s `lsql` query language is highly expressive, whereas (as far as we can tell) Hails uses a simpler query language targeting MongoDB where predicates can only depend on the document key. Third, there is no formal argument (and little informal argument) that Hails’ policy checks ensure a high-level security property. The ability to run arbitrary code to determine policies seems potentially risky (e.g., if there are mutually interacting policy functions), and there seems to be nothing like our database invariants that are needed for noninterference. Our mechanized formalization proved important: value-oriented policies (where one field’s label depends on another field) were tricky to get right (per § 3.5).

Finally, IFDB [129] defines an approach to integrating information flow tracking in an application and a database. Like Hails and `LWeb`, the application tracks a current “contamination level,” like `LIO`’s current label, that reflects data it has read. In IFDB, one can specify per-row policies using secrecy and integrity labels, but not policies per field. Labels are stored as separate, per-row metadata, implemented by changing the back-end DBMS. Declassification is permitted within trusted code blocks. Performance overhead for HTTP request latencies was similar to `LWeb`, at about 24%. Compared to IFDB, `LWeb` does not require any PSQL/database mod-

ifications; can support per-field, updatable labels; and can treat existing fields as labels, rather than requiring the establishment of a separate (often redundant) field just for the label. IFDB also lacks a clear argument for security, and has no formalization. Once again, we found such a formalization particularly useful for revealing bugs.

3.11 Conclusion

We presented **LWeb**, a information-flow security enforcement mechanism for Haskell web applications. **LWeb** combines **Yesod** with **LMonad**, a generalization of the **LIO** library. **LWeb** performs label-based policy checks and protects database values with dynamic labels, which can depend on the values stored in the database. We formalized **LWeb** (as λ_{LWeb}) and used Liquid Haskell to prove termination-insensitive noninterference. Our proof uncovered two noninterference violations in the implementation. We used **LWeb** to build the web site of the *Build it, Break it, Fix it* security-oriented programming contest, and found it could support rich policies and queries. Compared to manually checking security policies, **LWeb** impose a modest runtime overhead between 2% to 21% but reduces the trusted code base to 1% of the application code, and 21% overall (when counting **LWeb** too). With this minimal overhead cost, **LWeb** improves the security of database-backed applications by enforcing confidentiality and integrity policies.

```

eval :: Label l ⇒ Program l → Program l
eval (Pg lc (TBind t1 t2))
  | Pg lc' (TLIO t1') ← eval* (Pg lc t1) = Pg lc' (TApp t2 t1')

eval (Pg lc (TReturn t)) = Pg lc (TLIO t)

eval (Pg lc TGetLabel) = Pg lc (TReturn (TLabel lc))

eval (Pg lc (TLabel (TLabel l) t))
  | lc ⊆ l   = Pg lc (TReturn (TLabelled l t))
  | otherwise = Pg lc TException

eval (Pg lc (TUnlabel (TLabelled l t))) = Pg (l ⊔ lc) (TReturn t)

eval (Pg lc (TToLabeled (TLabel l) t))
  | Pg lc' (TLIO t') ← eval* (Pg lc t)
  , lc ⊆ l, lc' ⊆ l = Pg lc (TReturn (TLabelled l t'))
  | otherwise       = Pg lc (TReturn (TLabelled l TException))

eval (Pg lc t) = Pg lc (evalTerm t)

eval PgHole    = PgHole

evalTerm :: Label l ⇒ Term l → Term l
evalTerm (TLabelOf (TLabelled l _)) = TLabel l
evalTerm (TLabelOf t)                = TLabelOf (evalTerm t)
evalTerm (TApp (TLam x t) tx)        = subst (x,tx) t
evalTerm (TApp t tx)                 = TApp (evalTerm t) tx
evalTerm v                            = v

eval* :: Label l ⇒ Program l → Program l
eval* PgHole          = PgHole
eval* (Pg lc (TLIO t)) = Pg lc (TLIO t)
eval* p                = eval* (eval p)

subst :: Eq l ⇒ (Int, Term l)
      → Term l → Term l
subst = ...

```

Figure 3.6: Operational semantics of λ_{LIO} .

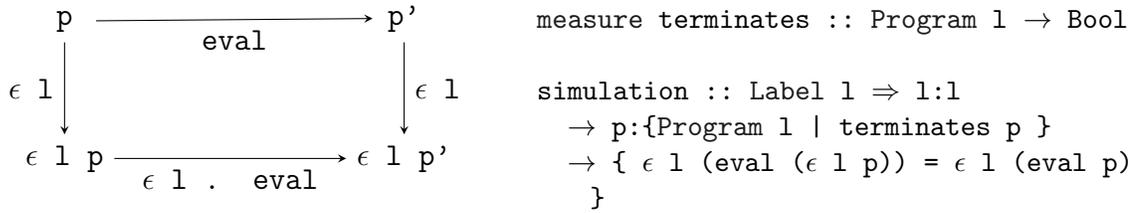


Figure 3.7: Simulation between `eval` and `ε l . eval`.

```

type DB l    = [(Name, Table l)]
type Name    = String
data Table l = Table {tpolicy :: TPolicy l, tRows :: [Row l]}
data Row l   = Row {rKey :: Term l, rVal1 :: DBTerm l, rVal2 :: DBTerm l}
type DBTerm l = {t:Term l | isDBValue t }
data TPolicy l = TPolicy { tpTableLabel  :: l , tpFresh :: Int
                          , tpLabelField1 :: {l1:l | l1 ⊆ tpTableLabel}
                          , tpLabelField2 :: Term l → l }
  
```

Figure 3.8: Definition of λ_{LWeb} database

```

data Program l =
  Pg { pLabel :: l, pDB :: DB l, pTerm :: Term l } | PgHole { pDB :: DB l }

data Term l = ...
  | TInsert Name (Term l) (Term l) | TSelect Name Pred
  | TDelete Name Pred              | TUpdate Name Pred (Term l) (Term l)
  
```

Figure 3.9: Extension of programs and terms with a database.

```

eval :: Label l ⇒ i:{Program l | ⊆ i && terminates i}
      → {o:Program l | ⊆ o }
eval (Pg l db (TInsert n t1 t2)
      | TLabeled l1 v1 ← t1, TLabeled l2 v2 ← t2, Just t ← db!!n
      , l1 ⊆ labelF1 t, l2 ⊆ labelF2 t v1, l ⊆ labelT t
      = let k = freshKey t in
      Pg (l ⊔ l1) (db += n (Row k v1 v2)) (TReturn k)

eval (Pg l db (TInsert n t1 t2))
      | TLabeled l1 v1 ← t1, TLabeled l2 v2 ← t2
      = Pg (l ⊔ l1) db (TReturn TException)

eval (Pg l db (TDelete n p))
      | Just t ← db!!n, l ⊔ labelPred p t ⊆ labelT t
      = let l' = l ⊔ labelRead p t in
      Pg l' (db -= n p) (TReturn TUnit)

eval (Pg l db (TDelete n p))
      | Just t ← db!!n
      = let l' = l ⊔ labelRead p t in
      Pg l' db (TReturn TException)
      | otherwise
      = Pg l db (TReturn TException)

eval (Pg l db (TSelect n p))
      | Just t ← db!!n
      = let l' = l ⊔ labelT t ⊔ labelPred p t in
      Pg l' db (TReturn (db = n p))

eval (Pg l db (TSelect n p))
      = Pg l db (TReturn TException)

eval (Pg l db (TUpdate n p t1 t2)
      | TLabeled l1 v1 ← t1, TLabeled l2 v2 ← t2, Just t ← db!!n
      , l ⊔ l1 ⊔ labelPred p t ⊆ labelF1 t
      , l ⊔ l2 ⊔ labelPred p t ⊆ labelF2 t v1
      = let l' = l ⊔ l1 ⊔ labelRead p t ⊔ labelT t in
      Pg l' (db := n p v1 v2) (TReturn TUnit)

eval (Pg l db (TUpdate n p t1 t2)
      | TLabeled l1 v1 ← t1, TLabeled l2 v2 ← t2, Just t ← db!!n
      = let l' = l ⊔ l1 ⊔ labelRead p t ⊔ labelT t in
      Pg l' db (TReturn TException)
      | otherwise
      = Pg l db (TReturn TException)
</pre

```

Figure 3.10: Evaluation of monadic database terms.

```

ε      :: (Label l) ⇒ l → Program l → Program l
εDB    :: (Label l) ⇒ l → DB l → DB l
εTable :: (Label l) ⇒ l → Table l → Table l
εRow   :: (Label l) ⇒ l → TPolicy l → Row l → Row l

ε l (PgHole db)                εTable l (Table tp rs)
  = PgHole (εDB l db)          | ¬ (tpTableLabel tp ⊆ l) = Table tp []
ε l (Pg lc db t)              εTable l (Table tp rs)
  | ¬ (lc ⊆ l)                 = Table tp (map (εRow l tp) rs)
  = PgHole (εDB l db)
  | otherwise                  εRow l tp (Row k v1 v2)
  = Pg lc (εDB l db) (εTerm l t) | ¬ (tpLabelField1 tp ⊆ l)
                                  = Row k THole THole
                                  | ¬ (tpLabelField2 tp v1 ⊆ l)
                                  = Row k (εTerm l v1) THole
                                  | otherwise
εDB l []                       = Row k (εTerm l v1) (εTerm l v2)
  = []
εDB l ((n,t):db)              = Row k (εTerm l v1) (εTerm l v2)
  = (n,εTable l):εDB l db

```

Figure 3.11: Erasure of programs and databases.

```

data Principal
  = PSys | PAdmin | PUser UserId
  | PTeam TeamId | PJudge JudgeId

type BBFLabel = DCLabel Principal

```

Figure 3.12: BIBIFI labels.

```

User
  account Text
  email Text   <Const Admin ⊑ Id, Id>
  admin Bool   <⊥, Const Admin>

```

Figure 3.13: Basic BIBIFI User table.

```

UserInfo
  user      UserId
  school    Text   <Const Admin ⊑ Field user, Field user>
  age       Int    <Const Admin ⊑ Field user, Field user>
  experience Int    <Const Admin ⊑ Field user, Field user>

```

Figure 3.14: Table UserInfo contains additional BIBIFI user information.

```

Announcement <⊥, Const Admin>      Team      TeamMember
  title  Text <⊥ Const Admin>      name  Text      team TeamId
  content Text <⊥, Const Admin>    contest ContestId  user UserId

```

Figure 3.15: Definition of Announcement, Team, and TeamMember tables and their policies.

```

BreakSubmission
  attacker TeamId <⊥, Const Sys>
  target   TeamId <⊥, Const Sys>
  result   Bool   <Const Admin ⊓ Field attacker ⊓ Field target, Const Sys>

```

Figure 3.16: Definition of BreakSubmission table and its policy.

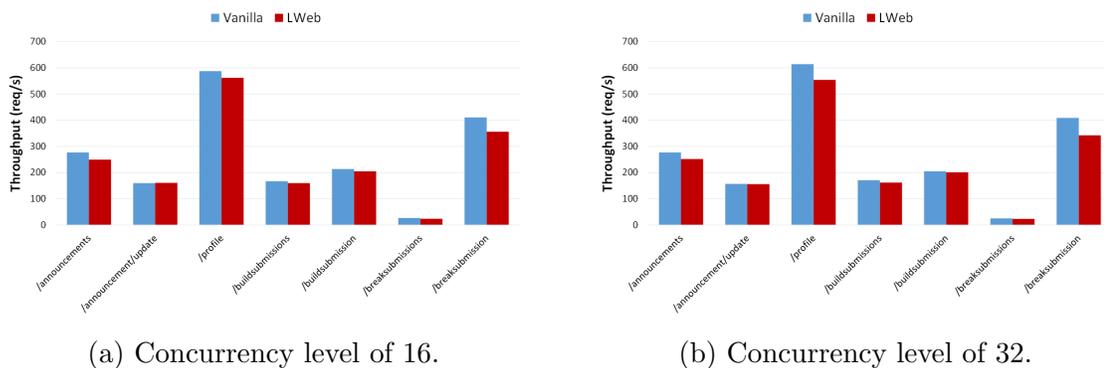


Figure 3.17: Throughput (req/s) of the **Vanilla** and **LWeb** versions of the BIBIFI application.

Chapter 4: Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell

4.1 Introduction

Ideally, we would like to verify that the implementation of `LWeb` is correct and satisfies noninterference. Unfortunately, Liquid Haskell lacked certain features to make this possible. One such feature was that Liquid Haskell could not verify properties of *typeclasses* [19]. To help develop this feature, we focused on the verification of distributed applications based on replicated data types (RDTs).

Typeclasses are used extensively throughout the Haskell ecosystem. A typeclass *definition* specifies a type constructor and a collection of method declarations over that type. A typeclass *instance* defines an implementation of that constructor and those methods. For example, the `Ord` typeclass from Haskell’s standard library declares that its instances `a` must have a method `(<=)` of type `a → a → bool`; numbers, strings, booleans, and many other types are instances of `Ord`. The standard `sort` function can only sort lists of types that are `Ord` instances, since it needs a comparison function; this requirement is expressed as a constraint on `sort`’s type, `Ord a ⇒ [a] → [a]`.

Typeclass refinements for Liquid Haskell. The primary contribution of this chapter is an extension to Liquid Haskell that supports stating and proving properties of typeclasses (§ 4.2). While it was previously possible in Liquid Haskell to prove properties of individual instances of a typeclass, it was not possible to give refinement types to a typeclass definition’s methods. As such, Liquid Haskell code and proofs could not then modularly *use* those types when invoking methods from functions whose arguments (like `sort`) have a typeclass constraint. Given the ubiquity of typeclasses in Haskell code, the ability to do this is key to being able to verify interesting properties of real-world Haskell applications.

Implementing typeclass refinements in Liquid Haskell was not straightforward. Its implementation works by verifying properties not of Haskell source code, but rather of *Core* expressions, which are the intermediate representation produced by the Glasgow Haskell Compiler (GHC) [28], the de facto Haskell standard. Doing so leverages functionality that GHC already provides (e.g., typechecking and elaboration) and allows Liquid Haskell to evolve semi-independently from GHC, since *Core*’s definition is relatively stable. But there is a problem: typeclasses are not *Core* expressions—during elaboration, GHC translates them to *dictionaries*, which are basically records of functions. Code that defines a typeclass instance is translated to create a dictionary, and code that expresses a typeclass constraint is translated to use a dictionary; e.g., `sort` will be translated to be passed an `Ord` dictionary, from which it invokes the `(<=)` method. To maintain the current separation between Liquid Haskell and GHC, our implementation (§ 4.3) transliterates typeclass methods’ refinement types to checked invariants over dictionaries, so refinement types

on typeclasses are verified when dictionaries are created, and those types can be used by client code. To do this modularly we had to expand the way Liquid Haskell interacts with GHC.

While Liquid Haskell is not the first proof system with typeclass support—Coq, Isabelle, Idris, F*, Agda, and Lean have typeclasses or something like them—our approach represents an interesting point in the design space (see § 4.5.2). In particular, our modular approach reuses Haskell’s typeclass resolution procedure, which limits typeclass type parameters to normal Haskell types. But, Haskell’s resolution is *coherent* by default (it always chooses the same typeclass instance for a given type) [167] and this fact is very useful for some proofs. Our implementation introduces a *checked invariant* during elaboration to express coherence, which is sound even if coherent resolution is overridden by GHC pragma (in which case proof of the invariant could fail at instance creation time). Other systems may allow instance types to be more general, but the cost is a more involved resolution procedure which may be neither coherent nor terminating, complicating its use in programming and proofs.

Case study: Verifying standard typeclass laws. As a simple test of the utility of typeclass refinements, we carried out a small case study: We used Liquid Haskell to verify that instances of standard Haskell typeclasses satisfy the expected typeclass laws (§ 4.2.2). Significant prior work has focused on this application specifically, employing a variety of techniques, including random testing, term rewriting, contract verification, and conversion to Coq (see § 4.5.1). Liquid Haskell typeclass

refinements offer a natural, general-purpose approach. In particular, laws can be expressed as refinements to methods of a subclass of the target typeclass, and proofs of them are carried out in a subclass of each implementation of that target typeclass. This approach permits proofs of existing Haskell code without requiring that code be directly modified or annotated. We demonstrate this for several standard typeclasses, including `Semigroup`, `Monoid`, `Functor`, `Applicative`, and `Monad`, proving 34 instantiations satisfy their laws, in all (§ 4.2.2). Mostly, we find that the proofs are short (just a couple of lines), thanks to Liquid Haskell’s SMT automation, and proof checking time is fast (typically a few seconds).

Case study: A platform for programming with verified replicated data types. With the success of this case study, we set out to build a platform for programming distributed applications based on *replicated data types* (RDTs) [18, 168, 169, 170, 171, 172, 173] (§ 4.4). Replication is ubiquitous in distributed systems to guard against machine failures and keep data physically close to clients who need it, but it introduces the problem of keeping replicas consistent with one another in the face of network partitions and unpredictable message latency. RDTs are data structures whose operations must satisfy certain mathematical properties that can be leveraged to ensure *strong convergence* [18], meaning that replicas are guaranteed to have equivalent state given that they have received and applied the same *unordered* set of update operations.

Liquid Haskell typeclasses provide a natural, modular, and elegant way to implement and verify RDTs. We define a typeclass `VRDT` with a refinement type that

captures the necessary properties, and we use Liquid Haskell to prove that those properties hold for a several primitive instances. We also defined several larger `VRDT` instances by modularly combining both the code and proofs of smaller ones. We state and prove, in Liquid Haskell, the strong convergence property that `VRDT` instances enjoy. Pleasantly, our approach generalizes and relaxes the typical assumption of *causal message delivery*. Our `VRDT` instances are sufficiently expressive that with them we were able to build a shared calendar event planner, and also a collaborative text editor, though the latter relies on a `VRDT` we have not yet fully verified, but expect to. Each application is implemented using a few hundred lines of Haskell code (§ 4.4.5). Although there exists previous work on mechanized verification of RDTs (§ 4.5.3), our work is, to our knowledge, the first to use a solver-aided language (Liquid Haskell or otherwise) to implement verified RDTs. Because Liquid Haskell is an extension of standard Haskell, our applications are real, running Haskell applications, but now with mechanically verified RDT implementations.

Contributions. In summary, this work makes the following contributions:

- We present an extension to Liquid Haskell that supports stating and proving refinements of typeclass methods’ types. The engineering of this extension is an interesting interaction between GHC and Liquid Haskell’s core proof infrastructure, and our design sheds light on the interplay between coherent typeclass resolution and modular proofs (Sections 4.2 and 4.3).
- We use our extension to Liquid Haskell to modularly verify that 34 standard instances satisfy the laws of five widely-used Haskell typeclasses, the `Semigroup`,

`Monoid`, `Functor`, `Applicative`, and `Monad` typeclasses (Section 4.2.2).

- We further use our extension to Liquid Haskell to implement a platform for distributed applications based on replicated data types. We define a typeclass whose Liquid Haskell type captures the mathematical properties that must be true of RDTs, prove in Liquid Haskell that strong convergence does indeed hold if these properties are satisfied, and implement (and prove correct) several instances of our refined typeclass. Using these instances we implement two realistic applications: a shared calendar event planner and a (partially verified) collaborative text editor (Section 4.4).

We are working with the Liquid Haskell maintainers to integrate our extension into the main implementation, at which point it will be freely available.

4.1.1 Acknowledgements

This chapter presents work currently in submission to a peer-reviewed conference. It is joint work between Yiyun Liu, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Yiyun implemented the typeclass extension to Liquid Haskell while Niki and I provided mentorship. Patrick built the message delivery system for the CRDT applications. I designed and implemented the verified CRDTs and Yiyun helped with the mechanization of the related proofs.

<pre>class Semigroup a where (<>) :: a → a → a instance Semigroup [a] where (<>) = (++)</pre>	<pre>class Semigroup a ⇒ VSemigroup a where lawAssociativity :: x:a → y:a → z:a → {x <> (y <> z) == (x <> y) <> z} instance Semigroup [a] ⇒ VSemigroup [a] where lawAssociativity = lAssoc</pre>
--	---

(a) Standard `Semigroup` typeclass and the list instance of it (b) `VSemigroup` extends `Semigroup` with an associativity law, which its list instance satisfies via `lAssoc`

Figure 4.1: Typeclasses with Refinement Types

4.2 Typeclasses in Liquid Haskell

This section presents our extension to Liquid Haskell (§ 1.1), which permits annotating a typeclass definition’s methods with refinement types, thus allowing a typeclass’s clients to assume those richer types, while obligating a typeclass’s instances to implement them (§ 4.2.1). As a demonstration of the effectiveness of this approach, we verify that 34 instances of 5 standard typeclasses satisfy the expected laws (§ 4.2.2).

4.2.1 Refinement Types for Typeclasses

We have extended Liquid Haskell to allow typeclass methods to be annotated with refinement types. Doing so allows a developer to state properties that a typeclass’s methods should always satisfy. Clients of that typeclass can thus assume those properties in their own proofs. Of course, implementors of the typeclass’s instances must prove those properties hold for their instance.

Laws as Refinement Types. We illustrate the utility of our extension by showing how standard typeclass *laws* can be encoded as refinement types. Laws are properties that clients of a typeclass generally assume, and that implementors of a typeclass are supposed to ensure. Of course, without something like our extension, there is no guarantee that they do so.

Figure 4.1(a) shows the `Semigroup` typeclass, which defines a type `a` that is equipped with a single operator `<>`. One particular implementation of this typeclass for lists (`[a]`) is also shown, where `<>` corresponds to the `List` append operator. A key law of semigroups is that their operator is associative. Clients of `Semigroup` may assume this law holds of any instance they are given; they may break if it does not. Fortunately, as we proved in the previous subsection, `List` append is associative, so the `List` instance of `Semigroup` satisfies the law. How can we show this?

We extend the syntax of typeclasses to allow for refinement types on method declarations. Below is a version of `Semigroup` extended to capture the associativity typeclass law as a refinement type.

```
class VSemigroup a where
  (<>)          :: a → a → a
  lawAssociativity :: x:a → y:a → z:a → {x <> (y <> z) == (x <> y) <> z}
```

`VSemigroup` matches the definition of `Semigroup` from Figure 4.1(a) but adds typeclass method `lawAssociativity`, which (extrinsically) defines the associativity property. All `VSemigroup` instances are now required to define `lawAssociativity` and provide an explicit associativity proof. The lower portion of Figure 4.1(b) implements the list instance of `VSemigroup` by extending `Semigroup` list instance and

providing the associativity proof `1Assoc` from ??.

Using the Laws, Modularly. By allowing refinement types on typeclass definitions, we extend the modularity benefits of typeclasses from code to proofs. In particular, clients of a refined typeclass can take advantage of its stated refinement types when conducting their own proofs. For example, below we express and prove an extrinsic property that extends associativity to four elements.

```
assoc2 :: VSemigroup a => x:a -> y:a -> z:a -> w:a
        -> { x <> (y <> (z <> w)) == ((x <> y) <> z) <> w }
assoc2 x y z w = lawAssociativity x y (z <> w)
                'const' lawAssociativity (x <> y) z w
```

The proof is a consequence of `lawAssociativity`, which is applied twice, combined with Haskell's `constant` function. The proof is carried out once, independent of any `VSemigroup` instance, but the property holds for all of them.

The code of our VRDT case study (§ 4.4) is set up similarly. We define a `VRDT` typeclass with operations on data that enjoy particular properties. Relying on these properties, we can prove `strongConvergence` of all VRDTs; this property essentially states that two replicas that start in the same state will end up in the same state if they apply the same operations, in any order.

Refinements in Subclasses. For improved modularity, our extension allows typeclass method refinements to refer to superclass methods. For example, another way to write `VSemigroup` is shown at the top of Figure 4.1(b), which literally extends `Semigroup` with the added method. Defining properties in subclasses is particularly useful when not wanting to modify typeclasses in other packages (including those

in normal, not Liquid, Haskell). It can also be useful when not wanting to necessarily require implementations to prove all possible properties; different subsets of properties of interest can be defined in different subclasses.

Haskell typeclasses can have multiple superclasses, which allows defining a typeclass containing properties of data structures that implement multiple typeclasses. For example, consider the `Monoid` typeclass, which extends `Semigroup` to also include the `mempty` identity element. Since a particular data structure (like a list) can implement both typeclasses, we could define the verified typeclass `VMonoid` that extends `VSemigroup` and `Monoid` with two laws.

```
class (VSemigroup a, Monoid a) => VMonoid a where
  lawEmpty    :: x:a -> { x <> mempty == x && mempty <> x == x }
  lawMconcat  :: xs:[a] -> { mconcat xs == foldr (<>) mempty xs }
```

That `mempty` is an identity for `<>` is encoded in the `lawEmpty` method; it refers to `<>`, which is defined in the `VSemigroup` parent typeclass. The law `lawMconcat` guarantees that `mconcat`, defined by `Monoid`, is equivalent to folding over a non-empty list with `<>`.

We can also define verified components from other verified components, where proofs of the former's properties can depend on properties that hold of the latter. For example, in our VRDT case study, we define a VRDT `TwoPMap` in terms of any other VRDT; here is the beginning of the instance definition:

```
instance (Ord k, VRDT v) => VRDT (TwoPMap k v) where ...
```

The proofs of `TwoPMap k v`'s properties make use of the properties that hold for `Ord k` and `VRDT v`.

Coherence. There is an interesting twist in our `VMonoid` example. As mentioned, `Monoid` extends `Semigroup`; as such, proofs of properties in `VMonoid` may wish to assume that the `VSemigroup` instance resolved for `VMonoid` has the *same* parent superclass as that of the resolved `Monoid` instance. Indeed, this assumption is critical for these properties: we require that the `<>` operator in both `Monoid` and `VSemigroup` to be literally the same function. Such an assumption is reasonable because Haskell’s typeclass resolution procedure is *coherent* by default—there can always be only one possible typeclass instance at a particular type. While coherence solves the the “diamond problem” [174], it is possible for programmers to override coherence via the `INCOHERENT` GHC pragma. In this case, we must take care that proofs of or using refinements do not assume coherence holds. We say more in § 3.6 about how our system internally reasons about coherence to ensure soundness and precise reasoning.

Limitation: No Refined Instances. A limitation of our approach is that typeclass instances cannot be defined for refined types, only for base types. For example, we cannot have distinct semigroup instances for positive and negative numbers, i.e., `instance VSemigroup { v:Int | 0 < v }` and `instance VSemigroup { v:Int | v < 0 }`. But this limitation confers the benefit that we can reuse GHC’s typeclass resolution procedure in our implementation, and proofs can take advantage of the fact that resolution is coherent. We say more in § 4.3.3.

4.2.2 Verifying Laws of Standard Typeclass Instances

Before getting into the details of how we implemented typeclass refinements (in the next section) we present a case study demonstrating that the pattern we have shown for stating and verifying the laws of standard typeclass instances works well.

In our case study, we considered five standard typeclasses: `Semigroup`, `Monoid`, `Functor`, `Monad`, and `Applicative`. Then we defined subclasses (`VSemigroup`, `VMonoid`, etc.) that contain the parent’s expected typeclass laws. We have shown the definitions of `VMonoid` and `VSemigroup` already; `Functor`, `Monad`, and `Applicative` are shown in Figure 4.2 with their refined subclasses. We defined and verified instances of the above typeclasses for the `All`, `Any`, `Dual`, `Endo`, `Identity`, `List`, `Maybe`, `Peano`, `Either`, `Const`, `State`, `Reader`, and `Succs` datatypes. Because datatypes are instances of multiple subclasses, we performed 34 instance-verifications in total.

This effort was quite manageable. Table 4.1 tabulates the results, indicating the instance type in the first column, and the typeclasses it implements in the second. For each implementation we tabulate the lines of proof required to verify the stated laws. We also report the average (and standard deviation) of the time (in seconds) it took Liquid Haskell to verify each module.¹

For many of the proofs, Liquid Haskell is able to automatically verify the typeclass properties using PLE (Proof by Logical Evaluation) [25, 27]. As such,

¹All experiments of this work were carried out by the `criterion` Haskell package, which repeatedly reruns benchmarks until the error is small enough [175]. Typically, `criterion` ran up to 15 trials. The experiments were run on a machine with an Intel Xeon CPU with 64GB of RAM, running Ubuntu 16.04 with Z3 version 4.4.1.

most of the proofs are a couple of lines of code. In general, PLE reduces manual effort but increases verification time, but for most modules the proofs are checked within just a few seconds. There are some exceptions—the `List`, `Reader`, and `Succs` `Applicative` instances are more involved, with the `Succs` module taking hundreds of seconds to verify. Unlike the other proofs, which usually require no more than two or three lemmas, the proof of the composition law of `Succs` involves applying nine separate lemmas. The lemmas give more candidates for PLE to rewrite, but most of the rewritings do not lead to the correct solution, and just slow things down.

In sum, this case study shows that typeclass refinements constitute a natural and modular approach to stating typeclass laws and proving that they are satisfied by their instances. § 4.4 presents further evidence, in the form of our `VRDT` case study, of the utility of typeclass refinements.

4.3 Implementing Typeclass Refinements

Now we turn to the question of how we extended Liquid Haskell to implement typeclass refinements.

Liquid Haskell statically verifies Haskell programs by analyzing *Core* expressions. *Core* is a small, explicitly-typed variant of System F generated during compilation by GHC, the Glasgow Haskell Compiler. Liquid Haskell can thus ignore many of Haskell’s myriad source-level constructs, and focus on a smaller language. This implementation approach is also useful for managing Liquid Haskell as an independent codebase. Even as Haskell is actively modified with new or improved features,

```

class Functor f where
  fmap :: (a → b) → f a → f b
  (<$) :: a → f b → f a

class Functor m ⇒ VFunctor m where
  lawFunctorId :: x:m a → {fmap id x = id x}
  lawFunctorComposition :: f:(b → c) → g:(a → b) → x:m a
    → {fmap (f . g) x = (fmap f . fmap g) x}

class Functor f ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b
  liftA2 :: (a → b → c) → f a → f b → f c
  (<*>) :: f a → f b → f b
  (<*>) :: f a → f b → f a

class (VFunctor f, Applicative f) ⇒ VApplicative f where
  lawApplicativeId :: v:f a → {pure id <*> v = v}
  lawApplicativeComposition :: u:f (b → c) → v:f (a → b) → w:f a
    → {pure (.) <*> u <*> v <*> w = u <*> v
      <*> w}
  lawApplicativeHomomorphism :: g:(a → b) → x:a → {px:f a | px =
    pure x}
    → {pure g <*> px = pure (g x)}
  lawApplicativeInterchange :: u:f (a → b) → y:a
    → {u <*> pure y = pure ($ y) <*> u}

class Applicative m ⇒ Monad m where
  (>>=) :: m a → (a → m b) → m b
  (>>) :: m a → m b → m b
  return :: forall a. a → m a

class (VApplicative m, Monad m) ⇒ VMonad m where
  lawMonad1 :: x:a → f:(a → m b) → {f x = return x >>= f}
  lawMonad2 :: m:m a → {m >>= return = m }
  lawMonad3 :: m:m a → f:(a → m b) → g:(b → m c)
    → {h:(y:a → {v:m c | v = f y >>= g}) | True}
    → {(m >>= f) >>= g = m >>= h}
  lawMonadReturn :: x:a → y:m a → {(y = pure x) ⇔ (y = return x)}

```

Figure 4.2: Typeclass definitions for Functor, Applicative, and Monad and their associated laws.

Type	Typeclass	# Lines Proof	Verif. Time (Std. dev.)
All	Semigroup	2	1.233 (0.086)
	Monoid	2	
Any	Semigroup	2	1.211 (0.035)
	Monoid	2	
Dual	Semigroup	2	2.023 (0.086)
	Monoid	2	
Endo	Semigroup	2	1.198 (0.078)
	Monoid	2	
Identity	Semigroup	2	1.560 (0.142)
	Monoid	2	
List	Functor	2	7.801 (0.191)
	Applicative	4	
	Monad	4	
	Semigroup	3	
	Monoid	3	
Maybe	Functor	4	2.607 (0.179)
	Applicative	25	
	Monad	10	
	Semigroup	3	
	Monoid	3	
Peano	Functor	3	4.504 (0.161)
	Applicative	8	
	Monad	6	
	Semigroup	3	
	Monoid	3	
Either	Semigroup	3	1.291 (0.117)
	Monoid	3	
	Functor	3	
Const	Functor	3	4.084 (0.244)
	Applicative	8	
	Monad	6	
State	Functor	2	0.921 (0.169)
Reader	Functor	12	1.113 (0.156)
	Applicative	11	
Succs	Functor	21	2.184 (0.103)
	Applicative	2	
	Functor	2	341.730 (1.794)
	Applicative	18	

Table 4.1: Total lines of proofs for each typeclass instance and the average verification time in seconds. Each reported time covers the laws on its row and those on the following rows up to the next reported time.

Liquid Haskell needs no modification because those features are translated to Core.

The challenge with implementing typeclass refinements is that GHC removes typeclasses entirely during the translation to Core; each typeclass is replaced with a *dictionary* of its various operations. Thus, our extension to Liquid Haskell needs a way to connect the refinements the programmer writes on typeclass methods with the translated Core that comes back from GHC, and it needs to do so in a way that is robust to (at least some) future changes in GHC’s elaboration. This section explains how we do this by delegating as much work as possible to GHC. We also explain how we model the fact that typeclass elaboration is coherent by default, to simplify user proofs.

4.3.1 GHC Typeclass Elaboration

Haskell compilers, including GHC, translate typeclass definitions and instances to datatypes known as *dictionaries* [176]. As an example, the Semigroup typeclass definition from Figure 4.1(a) is translated to a dictionary as the following datatype, `Semigroup` (simplified for clarity).

```
data Semigroup a = CSemigroup { (<>) :: a → a → a }
```

The datatype `Semigroup a` has a single constructor `CSemigroup` and one field for the `<>` method. In general, one field is defined for each typeclass method.

Typeclass instances are translated into dictionary values. For example, the list Semigroup instance from Figure 4.1(a) generates a `Semigroup [a]` dictionary, which GHC names `$fSemigroup []`.

```

$fSemigroup[] :: Semigroup [a]
$fSemigroup[] = CSemigroup ($c<>[])
$c<>[]       = (++)

```

The dictionary's field is the list append method `(++)`, which is assigned to the generated variable `$c<>[]`. (Both the dictionary and field variables are prefixed with `$` to indicate they are internal variable names, and posfixed with `[]` to indicate the list instance.)

Elaboration. The translated dictionaries are inserted after each method call via a procedure known as *elaboration*. For example, the Haskell code `x <> y` that appends two list variables `x, y :: [a]` is elaborated to `(<>) \ $fSemigroup [] x y`, where now `(<>)` is the record selector of the `Semigroup` data type. Functions that explicitly mention the `Semigroup a` constraint, as in `f` below, are elaborated to take an explicit dictionary argument; `f` elaborates to `fElab`, on the right.

```

f :: Semigroup a => a -> a -> a
  -> a
f x y = x <> y

fElab :: Semigroup a -> a -> a
fElab d x y = (<>) d x y

```

Subclass Encoding and Coherence. In Core, subclass dictionaries store references to parent dictionaries as fields. For example, the dictionary of the `VMonoid` typeclass from § 4.2.1 has four fields, two for the class methods and two for the superclass dictionaries:

```

data VMonoid a = CVMonoid {
  p1VMVSemigroup :: VSemigroup a
, p2VMMonoid     :: Monoid a
, lawEmpty       :: a -> ()
, lawMconcat     :: [a] -> ()
}

```

Interestingly, `Semigroup` is a superclass of both `Monoid` and `VSemigroup`, which leads to the “diamond problem.” When the user writes `x <> y`, it is unclear if GHC’s elaboration will access `(<>)` via the `Monoid` or via the `VSemigroup` field. That is, GHC can elaborate the `coherence` code below to either `coherenceElab1` or `coherenceElab2`.

```
coherence :: VMonoid a => a -> a -> a
coherence x y = x <> y

coherenceElab1, coherenceElab2 :: VMonoid a -> a -> a -> a
coherenceElab1 d x y = (<>) (p1VSSemigroup (p1VMVSemigroup d)) x y
coherenceElab2 d x y = (<>) (p1MSemigroup (p2VMMonoid d)) x y
```

Here, `p1VSSemigroup` and `p1MSemigroup` access the semigroup dictionary from the `VSemigroup` and `Monoid`, respectively. Such nondeterminism of elaboration could lead to problems, as the runtime semantics of `coherence` could change with GHC’s elaboration decision. Fortunately, by default GHC’s elaboration is *coherent* [167], meaning that the dictionary for each typeclass instance at a given type is unique; as such, we know that the `Semigroup` dictionary is the same irrespective of how it is accessed, i.e., $(\text{p1VSSemigroup} \ . \ \text{p1VMVSemigroup}) = (\text{p1MSemigroup} \ . \ \text{p2VMMonoid})$. Such an equality may be needed in a proof, so our implementation reflects it (in a safe manner) in the proof state, as discussed in § 4.3.3.

4.3.2 Interaction with GHC

Now we explain how we modified Liquid Haskell’s interaction with GHC so that we can verify typeclass refinements.

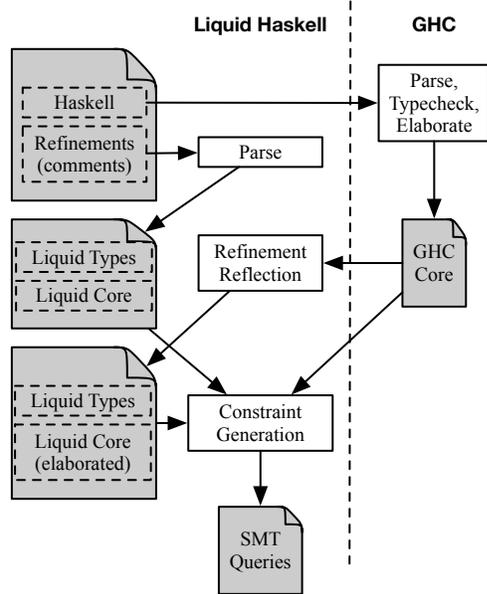
Refinements are ported to Dictionaries. The core intuition of typeclass verification is that refinements on typeclasses should be turned into refinements on the respective GHC-translated dictionaries. For example, the dictionary for the `VSemigroup` refined typeclass of Figure 4.1(b) should be refined to carry the associativity proof obligation (as a normal refinement):

```
data VSemigroup a = CSemigroup {
  p1VSSemigroup    :: Semigroup a
, lawAssociativity :: x:a → y:a → z:a → {x <> (y <> z) == (x <>
  y) <> z}
}
```

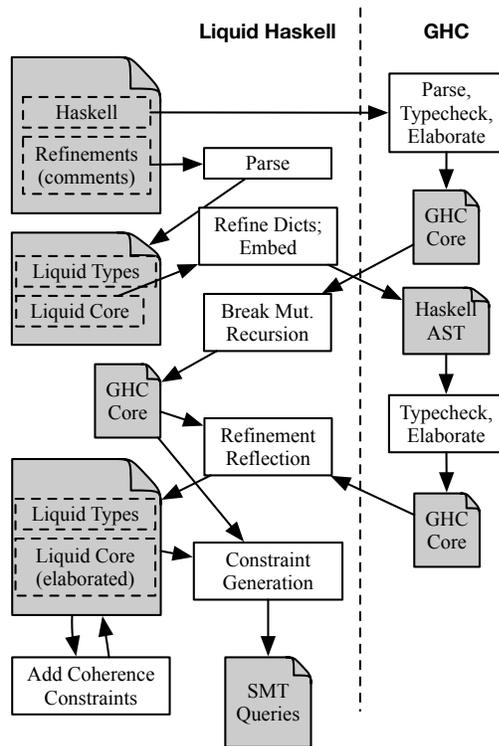
(Here, the first field of the dictionary is a link to the parent typeclass.) But of course we cannot literally do this because `lawAssociativity` is not well-formed Core. We make it so by expanding Liquid Haskell’s interaction with GHC, using it to parse, typecheck, and elaborate refinements.

Liquid Haskell’s Architecture. Figure 4.3 summarizes Liquid Haskell’s architecture and interaction with GHC API before and after support for refined typeclasses.

The first step is similar in both architectures: send Haskell source to GHC, which comes back as Core, and parse out refinement types appearing in comments. *Before* typeclass support (i.e., Figure 4.3a), the Core expressions were used by Liquid Haskell to strengthen the exact types for reflected functions, via refinement reflection, and to generate the verification constraints that were finally checked by an SMT. *After* typeclass support, the returned Core may include dictionary definitions, elaborated implementations of those dictionaries, and elaborated clients that use them, as explained in § 4.3.1. These dictionaries need to be connected to the



(a) Liquid Haskell's original architecture.



(b) Liquid Haskell's architecture with typeclass support.

Figure 4.3: Changes to Liquid Haskell's architecture.

refinement types retrieved from typeclass methods.

To connect typeclass method refinements to elaborated dictionaries, Liquid Haskell converts the parsed-out refinements into Haskell abstract syntax trees that make explicit reference to the relevant typeclasses. This occurs in the *Refine Dicts*; *Embed* step of the architecture diagram. As an example, for the `VSemigroup` method’s refinement of `lawAssociativity` (Figure 4.1), the following Haskell source-code AST expression is constructed:

```
(\x y z () → x <> (y <> z) = (x <> y) <> z) :: VSemigroup a ⇒
  a → a → a → () → Bool
```

GHC typechecks and elaborates the expression in the context of the `VSemigroup` definition it saw previously, and as such applies the appropriate dictionary arguments to typeclass methods. It returns the following:

```
(\d x y z () → x ‘<> (p1VSemigroup d)’ (y ‘<> (p1VSemigroup d)’ z)
 =
  (x ‘<> (p1VSemigroup d)’ y) ‘<> (p1VSemigroup d)’ z)
 :: VSemigroup a → a → a → a → () → Bool
```

Now the dictionary `d` is explicit in the elaborated Core expression. Liquid Haskell converts this into a refinement expression using refinement reflection, and then combines it with the Core code returned from GHC in the first step; for the example, the constructor and selectors for `VSemigroup` in Figure 4.1 are the following:

```
data VSemigroup a = CVSemigroup {
  p1VSSemigroup    :: Semigroup a
, lawAssociativity :: x:a → y:a → z:a
  → {x ‘<> p1VSSemigroup’ (y ‘<> p1VSSemigroup’ z)
    == (x ‘<> p1VSSemigroup’ y) ‘<> p1VSSemigroup’ z}
}

lawAssociativity :: d:VSemigroup a → x:a → y:a → z:a
```

```

→ {x '<>' (p1VSemigroup d)' (y '<>' (p1VSemigroup d)' z)
   == (x '<>' (p1VSemigroup d)' y) '<>' (p1VSemigroup d)' z}

```

First, notice that `lawAssociativity` basically matches the elaborated Core expression returned by GHC, but it has been converted to match Liquid Haskell’s refinement type syntax. Second, notice that the data constructor for `VSemigroup` also makes use of the returned Core, but has applied one additional step of transformation so that it can refer to the superclass’ (i.e., `Semigroup`) operator. Now, `VSemigroup` instances must satisfy the required properties since the dictionary datatype has been refined.

Typeclass methods do not only appear in the refinements of typeclasses. Functions with typeclass constraints may also contain typeclass methods in their refinements. We also elaborate the refinement expressions of these functions so that the appropriate dictionary arguments to typeclass methods are applied.

This whole process corresponds to the *refine dicts*; *embed*, *typecheck*, *elaborate*, and *refinement reflection* steps in the diagram. The *breaking mutual recursion* step inlines selector calls in the derived GHC dictionaries to break superficial mutual recursion since Liquid Haskell requires explicit proof of termination. The *adding coherence constraints* step is detailed in § 4.3.3.

Our implementation of all of this amounts to about 2000 lines of code, and is part of a fork Liquid Haskell’s codebase which is up to date with the main trunk as of May, 2020. Around 400 lines of code is used to define functions that communicate with the GHC API. The top-level driver function that orchestrates GHC’s *Typecheck*; *Elaborate* and Liquid Haskell’s *Refine Dicts*; *Embed* step takes another 700 lines of

code. This also includes the embedding functions from Liquid Haskell predicates to the Haskell AST. The rest of the code roughly corresponds to the *Refinement Reflection* step, where elaborated dictionaries are being converted into ordinary refined data types which Liquid already knows how to process and verify.

Limitation: Incompatibility with SMT interpreted Operators. One limitation with our implementation’s approach is that it is incompatible with the embedding of SMT theories. Liquid Haskell embeds operations from `Num`, `Ord` and `Eq` into the corresponding theories provided by the SMT solver. This allows Liquid Haskell to efficiently discharge theory-related proofs using existing decision procedures. However, if we elaborate an expression that uses the `+` operation, then `+` would no longer be treated a binary numerical operation, but rather as a data accessor that retrieves a binary operation from a `Num` dictionary. Currently, we simply add a special case which drops the dictionary if it corresponds to an instance of one of those three classes. By pretending those classes do not exist, we are still able to utilize the full power of the SMT solver, but we lose the ability to verify interesting instances, such as the `Num` instance of the free algebraic graph as defined Mokhov [177]. It would be interesting to explore how to get the best of both worlds: quickly discharging theory-related proofs and accessing the concrete definitions, as needed.

4.3.3 Reasoning About Coherence

In § 4.3.1 we mentioned that GHC’s elaboration is, by default, coherent. Various proofs on typeclass methods rely on coherence of elaboration (i.e., only hold

when instances are globally unique). Here, we give an example of such a proof and detail how coherence is automatically encoded and checked using refinement types.

lawMconcat for the VMonoid Dual Instance Requires Coherence. Given any binary operation $\langle \rangle$, we can always define a dual operation $\langle + \rangle$:

$$x \langle + \rangle y = y \langle \rangle x$$

Therefore, if we have a `Semigroup` instance for some type `a`, we can also define a `Dual` instance of a `Semigroup`. Indeed, we can define `Semigroups of Duals of any type a` once and for all:

```
newtype Dual a = Dual {getDual :: a}
instance Semigroup a => Semigroup (Dual a) where
  Dual (v :: a) <> Dual (v' :: a) = Dual (v' <> v)
```

Now, whenever we define a `Semigroup` instance for some type, GHC will automatically create its corresponding `Dual` instance. We do the same for `Monoid`, `VSemigroup`, and `VMonoid`.

```
instance Monoid a => Monoid (Dual a) where
  mempty      = Dual mempty
  mconcat xs = foldr (<>) mempty xs

instance VSemigroup a => VSemigroup (Dual a) where
  lawAssociative (Dual v) (Dual v') (Dual v'') = lawAssociative v''
    v' v

instance VMonoid a => VMonoid (Dual a) where
  lawEmpty (Dual v) = lawEmpty v
  -- lawMconcat :: VMonoid a => xs:_ -> {mconcat xs = foldr (<>)
    mempty xs}
  lawMconcat xs = () 'const' mconcat xs
```

The proof of `lawMconcat` proceeds by a simple unfolding of the `mconcat` definition, which is expressed as a call to that function (the full proof must be then cast to

unit via Haskell's `const ()`).

This proof requires coherence to hold. To see why, consider the proofs for the elaborated definitions. The elaborated equality `mconcat xs = foldr (<>) mempty xs` is as follows, for the dictionary `d :: VMonoid a`.

```
mconcat xs
= by elaboration
  mconcat ($fSemigroupDual (p2VMMonoid d)) xs
= by unfolding of mconcat
  foldr (<>) ($fSemigroupDual (p1SMonoid (p2VMMonoid d)))
    (mempty ($fMonoidDual (p2VMMonoid d)))
= by coherence
  foldr (<>) ($fSemigroupDual (p1VSSemigroup (p1VMVSemigroup d)))
    (mempty ($fMonoidDual (p2VMMonoid d)))
= by de-elaboration
  foldr (<>) mempty xs
```

The proof requires a coherence step to equate the two different ways that the semigroup operator `<>` is accessed. Concretely, the above equational proof only holds when `p1VSSemigroup (p1VMVSemigroup d)` equals `p1SMonoid (p2VMMonoid d)`, for all `d :: VMonoid a`. This equality cannot be asserted by the programmer, as dictionaries do not appear in the source.

Coherence as Dictionary Refinements. We represent the expected effect of coherent resolution as a refinement type on the datatype dictionary definitions for typeclasses. In particular, the equality of ancestor typeclasses is expressed as a refinement type on the fields of parent typeclasses. For example, the Core representation of `VMonoid` is as follows.

```
data VMonoid a = CVMonoid {
  p1VMVSemigroup :: VSemigroup a
, p2VMMonoid :: { v:Monoid a | p1VSSemigroup p1VMVSemigroup =
  p1MSemigroup v }
, ... -- as before
```

The added refinement on the second field states that the `Semigroup` from the `VSemigroup` parent (i.e., `p1VSSemigroup p1VMVSemigroup`) must be equal to the `Semigroup` from the `Monoid` parent (i.e., `p1MSemigroup v`). As a result, coherence becomes a *checked invariant* for each `VMonoid`—Liquid Haskell assumes the property for any `VMonoid` but checks it for instance declarations. This approach is sound even when GHC’s elaboration is potentially incoherent due to use of the `INCOHERENT` language pragma. Clients of dictionaries will still assume the invariant, but constructed dictionaries (i.e., typeclass instances) will induce an error from Liquid Haskell if the invariant cannot be proved.

Being able to take advantage of Haskell’s coherent typeclass resolution is the silver lining of our limitation that refined types cannot be distinct typeclass instances, as discussed at the end of § 4.2.1. If we allowed different instances of a class `TC` for both `{ v:int | v > 0 }` and `int`, say, then we could not use Haskell’s proved-coherent mechanism, and would have to develop our own and/or allow one

to be customized as part of proof search. There is no guarantee that the result would be coherent. In other dependently typed systems with typeclasses, e.g., Coq, the user has to explicitly encode and prove coherence requirements, case by case (see § 4.5).

4.4 Case Study: Verified Replicated Data Types

This section presents our platform for programming distributed applications based on *Conflict-free Replicated Data Types* (CRDTs) [18]. We define a Haskell typeclass `VRDT` to represent CRDTs and use type refinements to state the mathematical properties that CRDTs are expected to satisfy. We then prove in Liquid Haskell that `VRDT` instances (which must satisfy these properties) are sure to enjoy *strong convergence*. We have implemented a several primitive instances of the `VRDT` typeclass as well a mechanism for building compound `VRDT`s based on smaller components, where the necessary properties of the former automatically follow from the latter. With this infrastructure, as well as libraries for message delivery and user interaction we developed, we constructed two substantial applications, a shared event planner and a collaborative text editor (though the latter relies on a `VRDT` we have not fully verified, per § 4.4.3).

4.4.1 Background: Conflict-free Replicated Data Types (CRDTs)

A CRDT is a data structure with certain mathematical properties, designed for use in a distributed system that replicates its state. These properties enable

proving that such a system enjoys the *strong eventual consistency* (SEC) property, a key aspect of which is *strong convergence* (SC), which states that replicas of a CRDT that have received and applied the same set of updates will always have the same state, regardless of the order in which those updates are received and applied.

Shapiro et al. [18] describe two styles of CRDT specifications: *state-based*, in which replicas apply updates locally and periodically broadcast their local state to other replicas, which then merge the received state with their own, and *operation-based*, in which every state-updating operation is broadcast and applied at each replica. Shapiro et al. [18] prove that state-based and operation-based CRDTs are equivalent in the sense that each can be implemented in terms of the other (although practical implementation considerations may motivate the choice of one or the other in a particular application). In this work, we focus our attention on the operation-based style, which is suitable for implementing CRDTs such as ordered lists, which are key for applications such as collaborative text editing.

The key to proving convergence of (operation-based) CRDTs is to require that, under appropriate circumstances, the CRDT's operations commute. A replica that receives an operation from a client can update its own state and broadcast that operation to the other replicas. Since the operations commute, they can be applied in any order and produce the same final state, as required by SC. However, requiring operations to commute under all circumstances is too restrictive. Therefore, Shapiro et al. relax commutativity to exclude *causally ordered* operations. Such operations tend to affect the same parts of the state, and are usually issued in a strict order. For example, the operation of inserting a $k \rightarrow v$ pair in a map is causally ordered before

an operation that updates k 's mapped-to value. The assumption made by [Shapiro et al.](#) is that the underlying communication mechanism will ensure causal delivery to the CRDT, e.g., by employing vector clocks [178, 179] and buffering received operations until all operations that causally precede them have been applied [180]. They also assume that any preconditions that must be satisfied to enable an operation's execution (e.g., that a key must be present in a map if its value is to be updated) are ensured by causal delivery. Sometimes particular CRDTs make global assumptions for correctness, e.g., that generated keys are unique.

4.4.2 Verifying CRDTs with Typeclass Refinements

We define a CRDT as the Liquid Haskell typeclass `VRDT`, shown in Figure 4.4. Each `VRDT` has an associated `Op` type that specifies how the `VRDT`'s state is updated. The `apply` method takes a `VRDT` state `t` runs the given operation `Op t` on it, and returns the updated state.

Rather than formalize a general notion of causal delivery and additionally formalize any global correctness assumptions, we combine both together in a pair of predicates, `compat` and `compatS`. The former should return `True` for non-causally ordered operations (and perhaps others) that satisfy the global correctness assumptions. The latter will return `True` when the current state is compatible with the given operation, according to the global correctness assumptions. For example, in our `TwoPMap` key-value map implementation given below, we express the assumption of unique keys by deeming two insertion operations incompatible if they offer the

```

class VRDT t where
  type Op t
  apply :: t → Op t → t
  compat :: Op t → Op t → Bool
  compatS :: t → Op t → Bool

  lawCommut :: x:t → op1:Op t → op2:Op t
    → {(compat op1 op2 && compatS x op1 && compatS x op2)
       ⇒ (apply (apply x op1) op2 = apply (apply x op2) op1
          && compatS (apply x op1) op2)}

  lawCompatCommut :: op1:Op t → op2:Op t
    → {compat op1 op2 = compat op2 op1}

data Max a = Max a

instance Ord a ⇒ VRDT (Max a) where
  type Op (Max a) = Max a

  apply (Max a) (Max b) | a > b = Max a
  apply (Max a) (Max b)       = Max b

  compat op1 op2 = True
  compatS max op = True

  lawCommut max op1 op2 = ()
  lawCompatCommut op1 op2 = ()

```

Figure 4.4: Definition of the VRDT typeclass and its Max instance

same key, and deeming an insertion incompatible with a state that already contains the offered key. Our notion of compatibility is more flexible than a one-size-fits-all notion of causality. For example, while inserting a key-value pair is causally ordered with deleting that pair, these two operations can be deemed compatible (and are, in our `TwoPMap`) by internally buffering the latter until the former is delivered. It is up to the `VRDT` instance to decide, and reflect in its specifications of `compat` and `compatS`, what to do itself, and what to expect of the delivery mechanism.

The required mathematical properties of a `VRDT` are specified extrinsically as methods `lawCommut` and `lawCompatCommut`. The former's type specifies the property that operations compatible with each other and the current state must commute. The latter's type expresses that the operation-compatibility predicate `compat` must also be commutative.

Primitive VRDTs. An example `VRDT` instance, `Max`, is given in Figure 4.4. It contains a polymorphic value with a defined ordering (specified with an `Ord` instance) and tracks the maximum value of that type. Its corresponding operation's type `Op` is itself. All pairs of operations are compatible, so `compat` and `compatS` always return `True`. The `apply` function updates `Max`'s state by taking the greatest value of the two arguments. Proofs of `lawCommut` and `lawCompatCommut` for `Max` are trivial and Liquid Haskell proves them automatically.

In addition to `Max` we have implemented and mechanically verified four more primitive `VRDTs`.

- `Min v` is the dual of `Max v`, and tracks the smallest value seen.

- `Sum v` is an implementation of Shapiro et al. [168]’s Counter. `Ops` are numbers and the state is their sum.
- `LWW t v` is an implementation of Shapiro et al.’s *Last Writer Wins* Register. When a node writes to a register, it attaches a (polymorphic) timestamp to the value. A receiving node only updates its value if the timestamp is greater than the current timestamp. `LWW` assumes that all timestamps are unique.
- `MultiSet v` maintains a collection of values, like a `Set`, but each member has an associated count; a non-positive count indicates logical non-membership. `Ops` include value insertion and removal, each with an associated count. Besides using Liquid Haskell to prove `MultiSet` is a proper `VRDT`, we also proved its semantics simulates the semantics of mathematical multisets; details are in § 4.4.4.

We have also implemented Grishchenko’s *causal trees* [2010], which maintain an ordered sequence of values, but only partially verified their correctness. In a `CausalTree`, each value is assumed to have a unique identifier, and each value knows the identifier of the previous value. The relationship to the previous value creates a tree data structure that can be traversed (in preorder) to recover a converging ordering. Causal trees, like other RDTs representing ordered sequences (e.g., Roh et al. [169], Oster et al. [170], Pregoica et al. [171], Weiss et al. [172]), are useful for implementing collaborative text editing, but their behavior is considered especially challenging to specify and verify [182, 183]. We are prevented from completing our proof by a bug in Liquid Haskell (see § 4.4.3), but hope to rectify the problem soon.

```

data TwoPMap k v = TwoPMap {
    twoPMap      :: Map k v
    , twoPMapTombstone :: Set k
    , twoPMapPending  :: Map k [Op v]
}

data TwoPMapOp k v =
    TwoPMapInsert k v
  | TwoPMapDelete k
  | TwoPMapApply k (Op v)

instance (VRDT v, Ord k, Ord (Op v)) => VRDT (TwoPMap k v) where
  type Op (TwoPMap k v) = TwoPMapOp k v

compat (TwoPMapInsert k v) (TwoPMapInsert k' v') | k == k' = False
compat (TwoPMapApply k op) (TwoPMapApply k' op') | k == k' = compat op
  op'
compat _ _ = True

compatS (TwoPMap m t p) (TwoPMapInsert k v) = Map.lookup k m == Nothing
compatS (TwoPMap m t p) (TwoPMapApply k o) | Just v <- Map.lookup k m =
  compatS v o
compatS _ _ = True

apply (TwoPMap m p t) (TwoPMapInsert k v) | Set.member k t = TwoPMap m p
  t
apply (TwoPMap m p t) (TwoPMapInsert k v) =
  -- Apply pending operations.
  let (opsM, p') = Map.updateLookupWithKey (const (const Nothing)) k p
  in
  let v' = maybe v (foldr (\op v -> apply v op) v) opsM in
  let m' = Map.insert k v' m in
  TwoPMap m' p' t

apply (TwoPMap m p t) (TwoPMapApply k op) | Set.member k t = TwoPMap m p
  t
apply (TwoPMap m p t) (TwoPMapApply k op) =
  let (updatedM, m') = Map.updateLookupWithKey (\_ v -> Just (apply v op)
  ) k m in
  -- Add to pending if not inserted.
  let p' = if isJust updatedM then p else insertPending k op p in
  TwoPMap m' p' t

apply (TwoPMap m p t) (
  TwoPMapDelete k) =
  let m' = Map.delete k m in
  let p' = Map.delete k p in
  let t' = Set.insert k t in
  TwoPMap m' p' t'
  lawCommut _ _ _ = ...
  lawCompatCommut _ _ = ...

```

Figure 4.5: Implementation of TwoPMap

A Compound VRDT: Two-phase Map. We can also define VRDTs by reusing other VRDTs. In doing so, proofs of a compound VRDT’s required properties can be proved (in part) by using the properties of the VRDTs it is building on. As an example compound VRDT, we have implemented a *two-phase map*, shown in Figure 4.5. `TwoPMap` implements a map from keys to values, where values are themselves VRDTs. A `TwoPMap`’s operations are given by the datatype `TwoPMapOp`. Operation `TwoPMapInsert k v` inserts a $k \rightarrow v$ mapping; operation `TwoPMapDelete k` deletes a key; and `TwoPMapApply k (Op v)` applies a VRDT operation `Op` on `k`’s value `v`. An important restriction on a `TwoPMap`’s operation is that a key can only be used once in the map; even once a key is deleted it can never be re-added. This restriction is expressed in the definition of `compat` and `compatS`, which also lifts the requirement that the value VRDT’s operations are compatible. A few additional cases are omitted from the compatible predicates for brevity.

A two-phase map would naturally require causal delivery because a happens-before relationship exists between inserting and updating (or inserting and deleting) a value in the map. `TwoPMap` avoids the need for causal delivery (and thus does not specify it via `compat` or `compatS`) by buffering pending operations on a given key. The `apply` code for `TwoPMapApply` stores operations on the value of `k` in a separate operations buffer if `k` does not yet exist in the map. The `apply` code for `TwoPMapInsert` checks `k`’s operation buffer and applies any operations on the given value before inserting it with the mapping. The `apply` code for `TwoPMapDelete` clears `k` from the map and operations buffer, and adds it to the *tombstone*; future attempted insertions of `k` will be ignored due to its presence there. In general

```

data Event = Event {
    eventTitle :: LWW Timestamp Text
  , eventDescription :: LWW Timestamp Text
  , eventStartTime :: LWW Timestamp UTCTime
  , eventRSVPs :: MultiSet Text
}

```

Figure 4.6: Data type for a calendar event that is made up of VRDTs.

we assume that, like `TwoPMap`, instances of `VRDT` do not require causal delivery. Like `TwoPMap`, this requirement is easily satisfied by pushing a buffer of pending operations into the data type itself.

Automatically Deriving Compound VRDTs. Generally speaking, we might like to collect together several VRDTs to create an aggregate whose operations delegate to the operations of the components. For example, the shared event planner we discuss in § 4.4.5 represents a calendar event as a record with a title, description, time, and guest RSVP tally. To implement such a record as a VRDT, we can represent the first three fields as LWW registers and the last as a `MultiSet`, as shown in Figure 4.6. However, just collecting separate VRDTs together is not enough to show that the result is a VRDT: we need to define a corresponding `Op` data type and a `VRDT` instance for `Event`. Fortunately, since the fields of `Event` are VRDT instances, it is possible to derive the `Event` operation and `VRDT` instance automatically. We use Template Haskell [184] to, at compile time, generate operations and `VRDT` instances for data types that are composed of other VRDTs. Liquid Haskell can automatically verify that the generated code satisfies the `VRDT` properties.

4.4.3 Proofs

We have proved, in Liquid Haskell, that Strong Convergence, the key safety property required by Strong Eventual Consistency [18], holds for VRDT instances.

```
strongConvergence ::
  (Eq (Op a), VRDT a) =>
  s0:a → ops1:[Op a] → ops2:[Op a] →
  { (isPermutation ops1 ops2 && allCompatible ops1 &&
    allCompatibleState s0 ops1)
    => (applyAll s0 ops1 = applyAll s0 ops2)}
```

The theorem states that if two lists of operations are permutations of one another, then applying either one to the same input VRDT state will produce the same output state, assuming the list contains mutually compatible operations, and that all of these operations are compatible with the initial state. The proof is by induction over the operation lists and makes use of `lawCommut` and `lawCompatCommut` laws of VRDT. Importantly, the proof is independent of any *particular* VRDT instance, and thus applies to all of them.

Table 4.3 summarizes the lines of proof and verification time for the VRDTs we built. The development totals 2092 lines of code. These also include duplicate definition of Haskell functions in a way amenable to verification. For example, `Data.Map` was redefined to prove it satisfies the sortedness invariant, while common list functions were redefined to be reflected, as required by extrinsic proofs. As expected, Liquid Haskell’s PLE and SMT automation over intrinsic properties (e.g., sortedness invariant on `Data.Map`) aided proof generation. That said, there are still some issues to iron out. For example, there are difficulties proving properties of code

that makes use of typeclasses that have SMT-interpreted theories in Liquid Haskell, e.g., set theory used by the verified `Data.Map`. In fact, an existing limitation of this combination is blocking the verification of the `CausalTree` instance that we will resume once the Liquid Haskell limitation is addressed. The proof of `TwoPMap` also ran very slowly; because of the large search space (9 case splits between the 3 operations), the verification took more than 90 minutes. The long verification time can be attributed to PLE’s expansion and the discharging of verification conditions by the SMT solver. The bloated verification conditions consume a significant amount of memory space as well; when verifying the insert/apply case of `TwoPMap`, Liquid Haskell exhausted the 16 GiB physical memory and consumed no less than 1 GiB of the swap space.

In short, the verification effort was strenuous, which was expected as the first, real-world case study of refined typeclasses. Nevertheless, this case study increases our confidence that Liquid Haskell’s automation reduces proof effort and, since most of the implementation limitations we faced are already addressed, refined typeclasses in Liquid Haskell can actually be used to verify sophisticated properties of real-world applications.

4.4.4 Verifying CRDT Semantics

Just because a data type satisfies the required `VRDT` typeclass laws does not mean its implementation is correct. Fortunately, since the data type is defined in Liquid Haskell, it is possible to verify that its implementation matches an ideal se-

VRDT	Property	# Lines Proof	Verif. Time (Std. dev.)
-	strongConvergence	320	122.043 (2.415)
Max	lawCommut	1	0.544 (0.050)
	lawCompatCommut	1	
Min	lawCommut	1	0.565 (0.037)
	lawCompatCommut	1	
Sum	lawCommut	1	0.473 (0.028)
	lawCompatCommut	1	
LWW	lawCommut	1	0.835 (0.048)
	lawCompatCommut	1	
Multiset	lawCommut	315	48.555 (0.943)
	lawCompatCommut	1	
	simulation	72	45.705 (4.473)
TwoPMap	lawCommut	1253	5666.866 (56.797)
	lawCompatCommut	3	

Table 4.3: Total lines of proofs for each typeclass instance and the average verification time in seconds. Verifications times for `lawCommut` and `lawCompatCommut` are combined.

manics. To demonstrate this, we prove that the behavior of our `Multiset` VRDT (introduced in § 4.4.2) simulates the mathematical (denotational) semantics of multisets.

Our implementation of `Multiset` maintains a *positive* map `p` and a *negative* map `n`; the former contains members with positive counts while the latter contains members with non-positive counts. The `Ops` are `MultiSetOpAdd` and `MultiSetOpRemove`; they shift a value between maps as its count crosses 0. We define the denotation of a `MultiSet` to be a function from an element of the `Multiset` to the number of copies of that element. This is represented by the type alias `DMultiSet` in Figure 4.7. The `toDenotation` function is a straightforward mapping from a `MultiSet` to a `DMultiSet` that looks up the element in the positive and

```

type DMultiSet a = (a → Integer)

toDenotation :: Ord a ⇒ MultiSet a → DMultiSet a
toDenotation (MultiSet p n) t | Just v ← Map.lookup t p = v
toDenotation (MultiSet p n) t | Just v ← Map.lookup t n = v
toDenotation _ _ _ = 0

dApply :: Eq a ⇒ DMultiSet a → MultiSetOp a → DMultiSet a
dApply f (MultiSetOpAdd v c) t = if t == v then f t + c else f t
dApply f (MultiSetOpRemove v c) t = if t == v then f t - c else f t

simulation :: x:MultiSet a → op:{MultiSetOp a | enabled x op} → t:a
→ {toDenotation (apply x op) t == dApply (toDenotation x) op t}

```

Figure 4.7: Denotational semantics of `Multiset`.

negative `Map`'s of the `MultiSet`. `dApply` defines how to run a `MultiSet` operation on a `DMultiSet` by adding the number of new copies to the existing count. The `DMultiSet` denotation serves as a simple specification of how we expect `Multiset` to operate. We prove that `Multiset` and `DMultiSet` have the same behavior: The `simulation` theorem states that for all `MultiSet`'s and enabled operations on that `MultiSet`, looking up an element when you apply the operation on the `MultiSet` and then convert it to its denotation returns the same result as when you first convert it to a `DMultiSet` and run the operation on the denotation.

4.4.5 Applications

We built two realistic applications that are backed by `VRDT` instances: a *shared event planner* and a *collaborative text editor*. We close out this section by briefly describing these applications and some of the other infrastructure we built beyond `VRDT`s to put them together.

Message delivery and UI components. Both of our applications build on Haskell libraries we developed for message delivery and user interfaces. In particular, we developed a message delivery client and server to broadcast un-ordered messages from each client to all other clients. We also developed an application programming interface (API) to the client which transparently handles network disconnections by buffering and re-sending outgoing messages. Applications provide to the client API a function to receive messages, and the client API produces a function with which the application may send messages.

Our user interface library is based around *functional reactive programming* (FRP), a programming paradigm that models values that change over time [185]. FRP values are either continuous, called *behaviors*, or discrete, called *events*. We can treat replicated data types as FRP values whose state changes as a result of actions by the local user or update messages from a remote replica. We use Reflex ², a Haskell FRP library, to integrate FRP applications with our message delivery system. Any VRDT instance whose operations can be marshalled and sent over a network, e.g., as JSON, can be used as the state of these distributed applications. We provide the following library function, which internally calls the client API to connect a FRP application client to the server.

```
connectToStore :: (VRDT a, Serialize (Op a), MonadIO m)
  => ServerSettings -> a -> Event (Op a) -> m (Dynamic a)
```

`connectToStore` takes the settings of the server to connect to and an initial state. It also receives an `Event` of `Ops`. Any time the FRP client performs an operation, the

²<https://reflex-frp.org/>

event fires and this function sends the operation to the server. `Dynamic` is a special Reflex type that is both an event and a behavior. Whenever its value changes, an event fires as well. Since `connectToStore` returns a `Dynamic` of the current state, the FRP application automatically updates its interface whenever an operation is received and applied to the `VRDT` state.

Event planner. Our shared event planner application allows multiple users to create and manage calendar events and RSVP to event invitations. The planner's state (`TwoPMap UniqueId Event`) is a two-phase map where elements are the `VRDT` automatically derived from the `Event` type described in Figure 4.6. `UniqueId` is a pair of `ClientId` and an integer that is always incremented locally by the client application. It is used to ensure that the keys are unique as required by `TwoPMap`. The event planner has a terminal interface that supports viewing the list of events, creating events, updating events, and displaying event details. Since the application's state is a `VRDT` instance, updates are quickly displayed on all clients once they receive the corresponding operations. In this application, 12 lines of code define the types associated with the application's state, and one line of code invokes Template Haskell to generate the operation type for `Event` and its `VRDT` instance. The rest of the 400 lines of code in the application implement the user interface. The small amount of code necessary for managing replicated data highlights how `VRDTs` make it easy to build a distributed application.

Collaborative text editor. Our collaborative text editor represents the state of the text document being edited as a `CausalTree`. The majority of the code in the text editor (278 lines, out of roughly 350) is the `causalTreeInput` function, which has the following signature:

```
causalTreeInput :: Dynamic (CausalTree id Char)
                → Widget (Event (Op (CausalTree id Char)))
```

`causalTreeInput` creates a `Reflex widget` that builds a text box in the terminal interface that displays the contents of the `CausalTree`, handles scrolling, and processes keystrokes by the user. It takes a `Dynamic` of the `CausalTree` as input so that the view is updated when operations from the network are received. It returns an `Event` of `CausalTree` operations that fires whenever keystrokes update the state of the document.

4.5 Related Work

4.5.1 Verification of Haskell's Typeclass Laws

Verification of inductive properties, including per-instance typeclass laws, is possible in Haskell using dependently typed features [186, 187, 188, 189]. In work closely related to ours, Scott and Newton [190] verify algebraic laws of typeclasses using a singletons encoding of dependent types [191], and they employ generic programming to greatly reduce the proof burden. Even though their generic boilerplate technique is very effective for verifying typeclass instances, it is unclear how the encoding of typeclass laws interacts with the rest of the Haskell code that uses those

instances. In our approach, typeclass laws are expressed as refinement types and smoothly co-exist with refinement type specifications of clients of typeclass methods. In fact, [Scott and Newton](#) initially attempted to use Liquid Haskell, but it was impossible to do so at the time since Liquid Haskell did not yet support refinement types for typeclasses.

Haskell researchers have developed various techniques outside of Haskell itself to increase their confidence that typeclass laws actually hold. For example, Jeuring et al. [192] and Claessen and Hughes [193] used QuickCheck, the property-based random testing tool, to falsify typeclass laws. Zeno [194] and HERMIT [195] generate typeclass law proofs by term rewriting while HALO [196] uses an axiomatic encoding to verify Haskell contracts. HipSpec [197, 198] reduces typeclass laws to an external, automated-over-induction theorem prover. `hs-to-coq` [199] converts Haskell typeclasses and instances to equivalent ones in Coq which can then be proved to satisfy the respective laws.

Compared to these approaches, our technique has three main advantages. First, our proofs are Haskell programs, highly automated by SMT and PLE; unlike the other approaches, when proof automation fails, the user does not need to debug the external solver. Second, our proofs co-exist and interact with non-typeclass-specific Haskell code, so Haskell functions can use class laws to prove further properties (as in the `assoc2` example of § 4.2.1). Finally, our within-Haskell verification approach gives the developer the ability to distinguish between verified and original (i.e., non-verified) typeclasses (as in the `Semigroup` example of § 4.2.1) and the flexibility to only use verified methods on critical code fragments, thus

saving verification time.

4.5.2 Type System Expressiveness vs. Coherence of Elaboration

Typeclasses, introduced by Haskell [200], have been adopted by PureScript [201] and have inspired related abstractions in many programming languages, including Scala’s implicits [202] and Rust’s traits [203]. These languages (like vanilla Haskell) are not designed for proving rich logical properties, e.g., by making use of dependent types. But such simpler type systems make it possible to implement coherent typeclass resolution; for Haskell in particular, Bottu et al. [167] prove coherence of GHC’s elaboration by showing global uniqueness of dictionary creations. Coherence means that decisions made by typeclass elaboration cannot change the runtime semantics of the program, making it easier to reason about.

Fully dependently-typed languages such as Coq [204], Isabelle [205], Agda [206], Lean [14], and F* [207] permit proofs of rich logical properties, and also support typeclasses. However, to maximize expressiveness, their typeclass resolution procedures can end up being divergent or incoherent. For example, in Coq’s typeclasses [204], instantiation can diverge and is not guaranteed to be coherent since it is not always possible to decide whether two instances overlap [208].

In our work, we attempt to strike a balance between these two extremes. We use Liquid Haskell’s expressiveness to prove typeclass properties, while we use GHC’s less expressive type system to perform resolution. This design reduces our flexibility, as we cannot have distinct typeclasses for two refined types that have the

same base type. But, in turn, we gain two nice benefits. First, we reuse GHC’s mature elaboration implementation. More importantly, using elaboration on the coherent [167], less expressive type system of Haskell, we break the dilemma between expressiveness of the type system and coherence of elaboration.

4.5.3 Verifying Replicated Data Types

No verification can take place without a specification, and precisely *specifying* the behavior of replicated data types is a significant challenge in itself. Most work proposing new designs and implementations of replicated data structures (e.g., [18, 168, 169]) does not provide formal specifications. An exception is [Attiya et al.’s work \[2016\]](#), which precisely specifies a replicated list object and gives a (non-mechanized) proof that an implementation satisfies the specification.

[Burckhardt et al. \[209\]](#) proposed a comprehensive framework for formally specifying and verifying the correctness of RDTs, using an approach inspired by axiomatic specifications of weak shared-memory models. Although it is not obvious how to automate [Burckhardt et al.’s](#) verification approach, the [Quelea \[210\]](#) programming model uses the [Burckhardt et al.](#) specification framework as a contract language embedded in Haskell that allows programmers to attach axiomatic contracts to RDT operations; an SMT solver analyzes these contracts and determines the weakest consistency level at which an operation can be executed while satisfying its contract.³

[Gotsman et al. \[211\]](#) develop an SMT-automatable proof rule that can establish

³Implementation-wise, in contrast to our approach which uses Liquid Haskell’s solver-aided type system, [Quelea](#) is implemented in Haskell by directly querying the underlying SMT solver through the [Z3 Haskell bindings](#) at compile time, via [Template Haskell](#).

whether a particular choice of consistency guarantees for operations on RDTs is enough to ensure preservation of a given application-level data integrity invariant. This approach is implemented in Najafzadeh et al.’s CISE tool. Houshmand and Lesani’s Hamsaz system [2019] improves on CISE by automatically synthesizing a conflict relation that specifies which operations conflict with each other (whereas this conflict relation has to be provided as input to CISE).

Unlike our approach, tools like Quelea, CISE, and Hamsaz do not, in and of themselves, prove correctness properties of RDT implementations, e.g., strong convergence of replicas. Rather, they determine whether or not it is safe to execute a given RDT operation under the assumption that that replicas satisfy a given consistency policy (in the case of Quelea), or whether or not an application-level invariant will be satisfied, given the consistency policies satisfied by individual operations (in the case of CISE and Hamsaz). The goals of these lines of work are therefore complementary to ours: we *prove* a property of RDT implementations (strong convergence) that such tools could then leverage as an assumption to prove *application-level* properties, e.g., that a replicated bank account never has a negative balance. Verification of these application-level properties is important because CRDT correctness alone is not enough to ensure application correctness. (Of course, it would also be possible to prove such application-level properties directly in Liquid Haskell as well.)

Other works [183, 214, 215, 216] directly address proving the correctness of RDT implementations. Zeller et al. [214] specify and prove SC and SEC for a variety of state-based counter, register, and set CRDTs using the Isabelle/HOL

proof assistant. Nair et al. [215] present an automatic, SMT-based verification tool for specifying state-based CRDTs and verifying application-level properties of them. Neither Zeller et al. nor Nair et al. consider operation-based CRDTs, the focus of this work.

Gomes et al. [183] also use Isabelle/HOL to prove SC and SEC; like us, they focus on operation-based CRDTs. In addition to proving that RDT operations commute for three operation-based CRDTs—Shapiro et al.’s counters and observed-remove sets, and Roh et al.’s replicated growable arrays [2011]—Gomes et al. formalize in Isabelle/HOL a network model in which messages may be lost and replicas may crash, and prove that SC and SEC hold (under any behavior of the network model). Although it is possible to extract executable implementations from Isabelle definitions, our semi-automated Liquid Haskell-based approach has the advantage that the programmer can write, and use, mechanically verified RDT implementations without ever leaving Haskell. Gomes et al. bake causal delivery of updates into their network model (following Shapiro et al. [18], who assume causal delivery of updates in their proof of SEC for operation-based CRDTs); however, we observe that causal delivery is neither necessary nor sufficient to guarantee strong convergence [216].

Nagar and Jagannathan [216] address the question of automatically verifying strong convergence of various operation-based CRDTs (sets, lists, graphs) under different consistency policies provided by the underlying data store. They develop an SMT-automatable proof rule to show that all pairs of operations either commute or are guaranteed by the consistency policy to be applied in a given order. Given

a CRDT specification, their framework will determine which consistency policy is required for that CRDT. Their CRDT specifications are written in an abstract specification language designed to correspond to the first-order logic theories that SMT solvers support, whereas our verified RDTs are running Haskell code, directly usable in real applications.

4.6 Conclusion

We have presented an extension of Liquid Haskell to allow refinement types on typeclasses. Clients of a typeclass may assume its methods' refinement predicates hold, while instances of the typeclass are obligated to prove that they do. Implementing this extension was challenged by the fact that Liquid Haskell verifies properties of *Core*, the intermediate representation of the Glasgow Haskell Compiler, but typeclasses are replaced with dictionaries (records of functions) during translation to *Core*. Our implementation expands the interaction between Liquid Haskell and GHC to carry over refinements to those dictionaries during verification, and does so in a way that takes advantage of Haskell's typeclass resolution procedure being coherent. We have carried out two case studies to demonstrate the utility of our extension. First, we have used typeclass refinements to encode the algebraic laws for the `Semigroup`, `Monoid`, `Functor`, `Applicative`, and `Monad` standard typeclasses, and verified these properties hold of many of their instances. Second, we have used our extension to construct a platform for distributed applications based on replicated data types. We define a typeclass whose Liquid Haskell

type captures the mathematical properties of RDTs needed to prove the property of strong convergence; implement several instances of this typeclass; and use them to build two substantial applications.

Chapter 5: Conclusion

This dissertation studied the factors that influence software development and presented techniques for ensuring that software is correct and secure. Specifically, we have proven the thesis of this dissertation that *programming language-based techniques can be used to improve the correctness and security of programs*.

The BIBIFI competition served as a novel educational environment where participants learned to build secure programs and identify vulnerabilities in code written by other teams. As a quasi-controlled experiment, we ran quantitative and qualitative analysis on the results of the contest. C/C++ build-it submissions experienced better performance, while statically-typed language were less likely to have a security bug.

We built `LWeb` to protect the confidentiality and integrity of data stored in database-backed web applications. `LWeb` was implemented as a Haskell library that integrates with the Yesod web programming framework. Users specify expressive, label-based information flow control policies by annotating their database schema. `LWeb` uses a dynamic enforcement mechanism that halts execution if there is a flow of information that violates the user defined policy. The system imposed low runtime overhead according to benchmarks on the BIBIFI codebase and required minimal

developer effort for adoption. To reason about its correctness, we formalized LWeb’s metatheory with the λ_{LWeb} calculus and mechanically proved that it satisfies *non-interference* using Liquid Haskell.

Typeclasses are ubiquitous in the Haskell ecosystem, however Liquid Haskell was previously unable to reason about them. To remedy this situation, we extended Liquid Haskell with typeclass support. Users can now define refinement types on typeclasses and Liquid Haskell will ensure that their typeclass instances satisfy the required refinements. We used this new functionality to formally define the laws of existing typeclasses in the Haskell ecosystem and verify that their instances satisfy the laws. In addition, we formalized the mathematical properties of replicated data types, implemented several instances, and proved that the instance satisfy the required RDT properties. We proved that all replicated data types satisfy *strong convergence* and implemented two distributed applications using RDTs.

Ideally, the work in this dissertation is one step towards a future where software deployed in the real world is secure and bug-free. Based on my experience, I strongly believe that techniques used in this dissertation can eliminate classes of bugs and lead to more reliable programs.

5.1 Future Work

I envision many potential avenues for future work on the topics presented in this dissertation. Perhaps I may work on some of these ideas or others will take up the mantle. There are still many research questions that can be answered by

continuing to run BIBIFI contests. Since contestants produce multiple implementations solving the same problem and offer information about their demographic backgrounds, one could create a public dataset with this information. This would serve as a valuable resource for other researchers in areas including software engineering, programming languages, cybersecurity, and machine learning. For example, there are numerous metrics on code that attempt to reason about the quality of that code. A team’s performance in a public BIBIFI dataset would set a baseline for code quality and allow one to measure the efficacy of different metrics. If researchers develop bug and vulnerability detection tools using techniques like fuzzing and static analysis, the tools can be evaluated by how they perform on the BIBIFI dataset. BIBIFI could also be adopted to other domains. Formal method tools have the potential to drastically improve the quality of software, however they typically require user expertise and suffer from poor usability. On POPLmark’s retrospective panel [217], Benjamin Pierce suggested running BIBIFI where participants need to use formal methods tools to verify properties, encouraging these tools to improve in performance, accessibility, and usability.

There is room for future development on `LWeb`. One research idea is to verify a shallow embedding of `LWeb` to prove that the actual library implementation satisfies noninterference. Lack of typeclass support was one limitation that prevented this verification. This motivated the work on typeclasses and RDTs, so typeclasses are now supported in Liquid Haskell. Another limitation is that verifying a shallow embedding requires mechanizing monadic, effectful code to reason about `LWeb`’s monad transformer. Recent work on Interaction Trees [218] may be a helpful tool in

accomplishing this. Another potential improvement is to implement a static version of `LWeb`, so that users receive the IFC guarantees without any runtime overhead.

With additional engineering effort, the work on VRDTs could be used to build real distributed applications. Integration with public key infrastructure would allow messages to be encrypted and authenticated, enabling federated or decentralized applications.

Liquid Haskell is a promising verification tool, but there is still potential for improvement. The proofs of this work took hours to run fully, which hinders usability during proof development. Improving runtime performance of Liquid Haskell and adding sound incremental verification could improve the developer's experience. Creating an interactive proof environment that integrates with a language server protocol (LSP) server would also aid the developer by showing what proof goals remain.

Bibliography

- [1] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
- [2] MITRE. Common vulnerabilities and exposures. <https://cve.mitre.org/>, 2020.
- [3] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement Types for Haskell. In *ICFP*, 2014.
- [4] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.
- [5] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
- [6] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.
- [7] Danel Ahman, Cătălin Hrițcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 515–529, 2017.
- [8] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.
- [9] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2005.

- [10] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
- [11] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552, 2013.
- [12] Ulf Norell. Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer, 2008.
- [13] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda—a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.
- [14] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.
- [15] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1), 2003. ISSN 0733-8716.
- [16] Deian Stefan, Alejandro Russo, John Mitchell, and David Mazieres. Flexible dynamic information flow control in haskell. In *ACM SIGPLAN Haskell Symposium*, 2011.
- [17] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5), 1976. ISSN 0001-0782.
- [18] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24550-3.
- [19] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.
- [20] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, page 159–169, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375602. URL <https://doi.org/10.1145/1375581.1375602>.
- [21] John Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

- [22] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, 1998.
- [23] Robert L Constable and Scott Fraser Smith. Partial objects in constructive type theory. Technical report, Cornell University, 1987.
- [24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [25] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158141. URL <https://doi.org/10.1145/3158141>.
- [26] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: equational reasoning in liquid haskell (functional pearl). In Nicolas Wu, editor, *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, 2018.
- [27] K Rustan M Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *International Conference on Computer Aided Verification*, pages 361–381. Springer, 2016.
- [28] GHC. GHC: The Glasgow Haskell compiler. <https://www.haskell.org/ghc/>, 2020.
- [29] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.
- [30] Federal Business Council. Maryland cyber challenge & competition. <http://www.fbcinc.com/e/cybermdconference/competitorinfo.aspx>, 2012.
- [31] National Collegiate Cyber Defense Competition. <http://www.nationalccdc.org>, 2019.
- [32] DEF CON Communications. Capture the flag archive. <https://www.defcon.org/html/links/dc-ctf.html>, 2018.
- [33] Polytechnic Institute of New York University. Csaw - cybersecurity competition 2012. <http://www.poly.edu/csaw2012/csaw-CTF>, 2012.
- [34] dragostech.com inc. Cansecwest applied security conference. <http://cansecwest.com>, 2020.
- [35] TOPCODER. Top coder competitions. <http://apps.topcoder.com/wiki/display/tc/Algorithm+Overview>, 2020.

- [36] ICPC Foundation. ACM-ICPC International Collegiate Programming Contest. <http://icpc.baylor.edu>, 2018.
- [37] ICFP Programming Contest. <http://icfpcontest.org>, 2019.
- [38] Queena Kim. Want to learn cybersecurity? head to def con. <http://www.marketplace.org/2014/08/25/tech/want-learn-cybersecurity-head-def-con>, 2014.
- [39] Positive Technologies. ATM logic attacks: scenarios, 2018. <https://www.ptsecurity.com/upload/corporate/ww-en/analytics/ATM-Vulnerabilities-2018-eng.pdf>, November 2018.
- [40] James Parker, Michael Hicks, Andrew Ruef, Michelle L Mazurek, Dave Levin, Daniel Votipka, Piotr Mardziel, and Kelsey R Fulton. Build it, break it, fix it: Contesting secure development. *ACM Transactions on Privacy and Security (TOPS)*, 23(2):1–36, 2020.
- [41] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.
- [42] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *CCS*, 2016. ISBN 978-1-4503-4139-4.
- [43] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Atif Memon, Jandelyn Plane, and Piotr Mardziel. Build it break it: Measuring and comparing development security. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, 2015.
- [44] Git. Git – distributed version control management system. <http://git-scm.com>, 2020.
- [45] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4:1–4:40, 2009. ISSN 1094-9224.
- [46] Úlfar Erlingsson. personal communication stating that CFI was not deployed at Microsoft due to its overhead exceeding 10%, 2012.
- [47] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Atif Memon, Jandelyn Plane, and Piotr Mardziel. Build it break it: Measuring and comparing development security. In *CSET*, 2015.
- [48] Michael Snoyman. Yesod web framework for haskell. <http://www.yesodweb.com>, 2020.

- [49] The PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. <http://www.postgresql.org>, 2020.
- [50] James Parker, Niki Vazou, and Michael Hicks. Lweb: Information flow security for multi-tier web applications. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.
- [51] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.
- [52] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, pages 159–176. Springer, 2012.
- [53] OWASP. Secure coding practices - quick reference guide, 2010. URL https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf.
- [54] Kenneth P Burnham, David R Anderson, and Kathryn P Huyvaert. AIC model selection and multimodel inference in behavioral ecology: some background, observations, and comparisons. *Behavioral Ecology and Sociobiology*, 65(1):23–35, 2011.
- [55] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.
- [56] SeongIl Wi, Jaeseung Choi, and Sang Kil Cha. Git-based CTF: A simple and effective approach to organizing in-course attack-and-defense security competition. In *ASE 18*, 2018.
- [57] N J D Nagelkerke. A note on a general definition of the coefficient of determination. *Biometrika*, 78(3):691–692, 09 1991.
- [58] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. API blindspots: Why experienced developers write vulnerable code. In *Proc. SOUPS*, 2018.
- [59] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *Proc. IEEE S&P*, 2018.
- [60] DEF CON Communications. Def con hacking conference. <http://www.defcon.org>, 2010.

- [61] Nicholas Childers, Bryce Boe, Lorenzo Cavallaro, Ludovico Cavedon, Marco Cova, Manuel Egele, and Giovanni Vigna. Organizing large scale hacking competitions. In *DIMVA*, 2010.
- [62] Adam Doupé, Manuel Egele, Benjamin Caillat, Gianluca Stringhini, Gorkem Yakin, Ali Zand, Ludovico Cavedon, and Giovanni Vigna. Hit 'em where it hurts: A live security exercise on cyber situational awareness. In *ACSAC*, 2011.
- [63] Erik Trickel, Francesco Disperati, Eric Gustafson, Faezeh Kalantari, Mike Mabey, Naveen Tiwari, Yeganeh Safaei, Adam Doupé, and Giovanni Vigna. Shell we play a game? ctf-as-a-service for security education. In *ASE 17*, 2017.
- [64] Daniele Antonioli, Hamid Reza Ghaeini, Sridhar Adepu, Martin Ochoa, and Nils Ole Tippenhauer. Gamifying ics security training and research: Design, implementation, and results of s3. In *CPS 2017, CPS '17*, 2017. ISBN 978-1-4503-5394-6.
- [65] Kevin Bock, George Hughey, and Dave Levin. King of the hill: A novel cybersecurity competition for teaching penetration testing. In *2018 USENIX Workshop on Advances in Security Education (ASE 18)*, 2018.
- [66] Peter Chapman, Jonathan Burket, and David Brumley. PicoCTF: A game-based computer security competition for high school students. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*, 2014.
- [67] Gregory Conti, Thomas Babbitt, and John Nelson. Hacking competitions and their untapped potential for security education. *Security & Privacy*, 9(3): 56–59, 2011.
- [68] Chris Eagle. Computer security competitions: Expanding educational outcomes. *Security & Privacy*, 11(4), 2013.
- [69] Lance J. Hoffman, Tim Rosenberg, and Ronald Dodge. Exploring a national cybersecurity exercise for universities. *Security & Privacy*, 3(5):27–33, 2005.
- [70] Art Conklin. Cyber defense competitions and information security education: An active learning solution for a capstone course. In *HICSS*, 2006.
- [71] Art Conklin. The use of a collegiate cyber defense competition in information security education. In *InfoSecCD*, 2005.
- [72] Muhammad Mudassar Yamin, Basel Katt, Espen Torseth, Vasileios Gkioulos, and Stewart James Kowalski. Make it and break it: An iot smart home testbed case study. In *Proceedings of the 2Nd International Symposium on Computer Science and Intelligent Control, ISCSIC '18*, 2018.
- [73] Google. Google code jam. <http://code.google.com/codejam>, 2020.

- [74] BSIMM. Building security in maturity model (bsimm). <http://bsimm.com>, 2020.
- [75] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.
- [76] Gary McGraw. *Software Security: Building Security In*. Software Security Series. Addison-Wesley, 2006.
- [77] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Professional Computing Series. Addison-Wesley, 2001.
- [78] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2003.
- [79] Robert C. Seacord. *Secure Coding in C and C++*. Professional Computing Series. Addison-Wesley, 2013.
- [80] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Professional Computing Series. Addison-Wesley, 2007.
- [81] Paul E. Black, Lee Badger, Barbara Guttman, and Elizabeth Fong. Dramatically reducing software vulnerabilities: Report to the white house office of science and technology policy. Technical Report Draft NISTIR 8151, National Institute of Standards and Technology, 2016. http://csrc.nist.gov/publications/drafts/nistir-8151/nistir8151_draft.pdf.
- [82] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L. Mazurek, and Sascha Fahl. Developers need support too: A survey of security advice for software developers. In *IEEE SecDev 2017*.
- [83] Matthew Finifter and David Wagner. Exploring the relationship between web application development tools and security. In *USENIX Conference on Web Application Development (WebApps)*, 2011.
- [84] L. Prechelt. Plat_forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Transactions on Software Engineering*, 37(1):95–108, 2011.
- [85] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An empirical study on the effectiveness of security code review. In *ESSOS 2013*, 2013.
- [86] Riccardo Scandariato, James Walden, and Wouter Joosen. Static analysis versus penetration testing: A controlled experiment. In *ISSRE 2013*, 2013.
- [87] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *IEEE International Symposium on Software Reliability Engineering*, 2014.

- [88] Joonseok Yang, Duksan Ryu, and Jongmoon Baik. Improving vulnerability prediction accuracy with secure coding standard violation measures. In *International Conference on Big Data and Smart Computing (BigComp)*, 2016.
- [89] Keith Harrison and Gregory White. An empirical study on the effectiveness of common security measures. In *Hawaii International Conference on System Sciences (HICSS)*, 2010.
- [90] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In *Proc. ACM CCS*, 2013.
- [91] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *CCS '12: Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, October 2012.
- [92] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *CCS 2013*, pages 73–84. ACM, 2013.
- [93] J Xie, H R Lipford, and B Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2011. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6070393.
- [94] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. It’s the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer’s blind spots. In *ACSAC*, 2014.
- [95] Glenn Wurster and P C van Oorschot. The developer is the enemy. In *NSPW*, page 89, 2008.
- [96] Charles Weir, Awais Rashid, and James Noble. I’d like to have an argument, please: Using dialectic for effective app security. In *2nd European Workshop on Usable Security (Euro USEC 2017)*. Internet Society, 2017.
- [97] Ingolf Becker, Simon Parkin, and M. Angela Sasse. Finding security champions in blends of security culture. In *2nd European Workshop on Usable Security (Euro USEC 2017)*. Internet Society, 2017.
- [98] Christian Stransky, Yasemin Acar, Duc Cuong Nguyen, Dominik Wermke, Doowon Kim, Elissa M. Redmiles, Michael Backes, Simson Garfinkel, Michelle L. Mazurek, and Sascha Fahl. Lessons learned from using an on-line platform to conduct large-scale, online controlled security experiments with software developers. In *CSET 17*, 2017.

- [99] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.
- [100] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. Security developer studies with github users: Exploring a convenience sample. In *SOUPS 2017*, 2017.
- [101] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. API blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, 2018.
- [102] Christopher Thompson and David Wagner. A large-scale study of modern code review and security in open source projects. In *PROMISE 17*, 2017.
- [103] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, 2009.
- [104] American Fuzzing Lop (AFL). <http://lcamtuf.coredump.cx/af1/>, 2018.
- [105] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *CCS '18*, 2018.
- [106] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2018.
- [107] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. Bucketing failing tests via symbolic analysis. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2017.
- [108] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, 1999. ISBN 1-58113-095-3.
- [109] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1), 2003. ISSN 0164-0925.
- [110] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security Symposium*, 2007. ISBN 111-333-5555-77-9.
- [111] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. Selinq: Tracking information across application-database boundaries. In *ICFP*, 2014. ISBN 978-1-4503-2873-9.

- [112] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *SOSP*, 2007. ISBN 978-1-59593-591-5.
- [113] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6), 2009. ISSN 1615-5262.
- [114] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *PLDI*, 2015.
- [115] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.
- [116] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In R. Sekar and Arun K. Pujari, editors, *Proceedings of the International Conference on Information Systems Security (ICISS)*, volume 5352 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2008. URL <http://www.cs.umd.edu/~mwh/papers/implicitflows.pdf>.
- [117] Andrey Chudnov and David A. Naumann. Inlined information flow monitoring for javascript. In *CCS*, 2015. ISBN 978-1-4503-3832-5.
- [118] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *PLDI*, 2009. ISBN 978-1-60558-392-1.
- [119] Eran Tromer and Roei Schuster. Droiddisintegrator: Intra-application information flow control in android apps. In *ASIA CCS*, 2016. ISBN 978-1-4503-4233-9.
- [120] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *PLDI*, 2016.
- [121] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012. ISBN 978-1-4503-1083-3.
- [122] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *NordSec*, 2012. ISBN 978-3-642-29614-7.
- [123] Esqueleto. Esqueleto: Type-safe edsl for sql queries on persistent backends. https://github.com/bitemyapp/esqueleto/blob/master/docs/blog_post_2012_08_06, 2018.

- [124] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell Workshop*, 2002.
- [125] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012. ISBN 978-1-931971-96-6.
- [126] Daniel Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. *Journal of Computer Security*, 25(4), 2017.
- [127] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 269–282, 2009. URL <http://www.cs.umd.edu/~mwh/papers/selinks.pdf>.
- [128] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI*, 2010.
- [129] David Schultz and Barbara Liskov. Ifdb: Decentralized information flow control for databases. In *EuroSys*, 2013. ISBN 978-1-4503-1994-2.
- [130] James Parker, Niki Vazou, and Michael Hicks. Lweb: Information flow security for multi-tier web applications. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2019.
- [131] James Parker. LMonad: Information flow control for haskell web applications. Master’s thesis, Dept of Computer Science, the University of Maryland, 2014.
- [132] Joseph A. Goguen and José Meseguer. Security policies and security models. In *S&P*, 1982.
- [133] Michael Snoyman. Yesod web framework for haskell. <http://www.yesodweb.com/>, 2018.
- [134] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with SMT. *PACMPL*, 2(POPL), 2018.
- [135] Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. In *Haskell Symposium*, 2017.
- [136] Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19), 2010. ISSN 0304-3975.
- [137] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Haskell Symposium*, 2008.

- [138] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F^* . In *POPL*, 2016.
- [139] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.
- [140] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27(E5), 2017.
- [141] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5), 2009. ISSN 0926-227X.
- [142] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *J. Comput. Secur.*, 17(5):655–701, October 2009. ISSN 0926-227X. URL <http://dl.acm.org/citation.cfm?id=1662658.1662660>.
- [143] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 114–128, June 2011. doi: 10.1109/CSF.2011.15.
- [144] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973. URL <http://doi.acm.org/10.1145/512950.512973>.
- [145] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.
- [146] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.
- [147] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.
- [148] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. *MITRE Technical Report 2547*, 1, 1973.

- [149] Luísa Lourenço and Luís Caires. Dependent information flow types. In *POPL*, 2015. ISBN 978-1-4503-3300-9.
- [150] Luísa Lourenço and Luís Caires. Information flow analysis for valued-indexed data security compartments. In *International Symposium on Trustworthy Global Computing - Volume 8358*, 2014. ISBN 978-3-319-05118-5.
- [151] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security*, 24(2), 2016.
- [152] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP*, 2005. ISBN 1-59593-079-5.
- [153] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI*, 2006.
- [154] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP*, 2007. ISBN 978-1-59593-591-5.
- [155] Alejandro Russo. Functional pearl: Two can keep a secret, if one of them uses haskell. In *ICFP*, 2015. ISBN 978-1-4503-3669-7.
- [156] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ICFP*, 2015. ISBN 978-1-4503-3669-7.
- [157] Lucas Waye, Pablo Buiras, Owen Arden, Alejandro Russo, and Stephen Chong. Cryptographically secure information flow control on key-value stores. In *CCS*, 2017. ISBN 978-1-4503-4946-8.
- [158] Pablo Buiras, Joachim Breitner, and Alejandro Russo. Securing concurrent lazy programs against information leakage. In *CSF*, 2017.
- [159] Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. Ifc inside: Retrofitting languages with dynamic information flow control. In *POST*, 2015. ISBN 978-3-662-46665-0.
- [160] Marco Vassena and Alejandro Russo. On formalizing information-flow control libraries. In *PLAS*, 2016. ISBN 978-1-4503-4574-3.
- [161] Divya Muthukumaran, Dan O’Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch. Flowwatcher: Defending against data disclosure vulnerabilities in web applications. In *CCS*, 2015. ISBN 978-1-4503-3832-5.

- [162] Nikhil Swamy, Brian Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 369–383, 2008. URL <http://www.cs.umd.edu/~mwh/papers/fable.pdf>.
- [163] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In *USENIX Security Symposium*, 2017. ISBN 978-1-931971-40-9.
- [164] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, 2012.
- [165] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In *PLAS*, 2013.
- [166] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. In *Haskell Symposium*, 2012.
- [167] Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. Coherence of type class resolution. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.
- [168] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.
- [169] Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, 2011. doi: 10.1016/j.jpdc.2010.12.006. URL <https://doi.org/10.1016/j.jpdc.2010.12.006>.
- [170] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work, CSCW '06*, page 259–268, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595932496. doi: 10.1145/1180875.1180916. URL <https://doi.org/10.1145/1180875.1180916>.
- [171] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems, ICDCS '09*, page 395–403, USA, 2009. IEEE Computer Society. ISBN 9780769536590. doi: 10.1109/ICDCS.2009.20. URL <https://doi.org/10.1109/ICDCS.2009.20>.
- [172] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems, ICDCS '09*, page 404–412, USA, 2009. IEEE Computer Society. ISBN

9780769536590. doi: 10.1109/ICDCS.2009.75. URL <https://doi.org/10.1109/ICDCS.2009.75>.
- [173] M. Kleppmann and A. R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.
- [174] Bjarne Stroustrup. Multiple inheritance for c++. *Computing Systems*, 2(4):367–395, 1989.
- [175] Bryan O’Sullivan. criterion: Robust, reliable performance measurement and analysis. <https://hackage.haskell.org/package/criterion>, 2020.
- [176] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, 2007.
- [177] Andrey Mokhov. Algebraic graphs with class (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, 2017.
- [178] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988. URL <http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf>.
- [179] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).
- [180] Ken Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9:272–, 08 1991. doi: 10.1145/128738.128742.
- [181] Victor Grishchenko. Deep hypertext with embedded revision control implemented in regular expressions. In *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, WikiSym ’10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300568. doi: 10.1145/1832772.1832777. URL <https://doi.org/10.1145/1832772.1832777>.
- [182] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC ’16, pages 259–268, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3964-3. doi: 10.1145/2933057.2933090. URL <http://doi.acm.org/10.1145/2933057.2933090>.

- [183] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133933. URL <https://doi.org/10.1145/3133933>.
- [184] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002. ISSN 0362-1340. doi: 10.1145/636517.636528. URL <https://doi.org/10.1145/636517.636528>.
- [185] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997. URL <http://conal.net/papers/icfp97/>.
- [186] Richard A. Eisenberg. Dependent types in haskell: Theory and practice. *CoRR*, abs/1610.07978, 2016. URL <http://arxiv.org/abs/1610.07978>.
- [187] Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. A role for dependent types in haskell. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341705. URL <https://doi.org/10.1145/3341705>.
- [188] Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. Kind inference for datatypes. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371121. URL <https://doi.org/10.1145/3371121>.
- [189] Conor McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002.
- [190] Ryan G. Scott and Ryan R. Newton. Generic and flexible defaults for verified, law-abiding type-class instances. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 15–29, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi: 10.1145/3331545.3342591. URL <https://doi.org/10.1145/3331545.3342591>.
- [191] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *SIGPLAN Not.*, 47(12):117–130, September 2012. ISSN 0362-1340. doi: 10.1145/2430532.2364522. URL <https://doi.org/10.1145/2430532.2364522>.
- [192] Johan Jeuring, Patrik Jansson, and Cláudio Amaral. Testing type class laws. In *Proceedings of the 2012 Haskell Symposium*, pages 49–60, 2012.
- [193] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.
- [194] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In Cormac Flanagan

and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–421, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28756-5.

- [195] Andrew Farmer, Neil Sculthorpe, and Andy Gill. Reasoning with the hermit: Tool support for equational reasoning on ghc core programs. *SIGPLAN Not.*, 50(12):23–34, August 2015. ISSN 0362-1340. doi: 10.1145/2887747.2804303. URL <https://doi.org/10.1145/2887747.2804303>.
- [196] Dimitrios Vytiniotis, Simon L. Peyton Jones, Koen Claessen, and Dan Rosén. Halo: haskell to logic through denotational semantics. In *POPL*, 2013.
- [197] Andreas Arvidsson, Moa Johansson, and Robin Touche. Proving type class laws for haskell. In *International Symposium on Trends in Functional Programming*, pages 61–74. Springer, 2016.
- [198] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Hipspec: Automating inductive proofs of program properties. In *ATx/WInG at IJCAR*, pages 16–25, 2012.
- [199] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total haskell is reasonable coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, page 14–27, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355865. doi: 10.1145/3167092. URL <https://doi.org/10.1145/3167092>.
- [200] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*, volume 13. 01 2003.
- [201] Phil Freeman. *PureScript by Example*. 2017. URL <https://doi.org/10.1145/2887747.2804303>.
- [202] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: Foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158130. URL <https://doi.org/10.1145/3158130>.
- [203] rust. The rust programming language. <http://www.rust-lang.org/>.
- [204] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-71067-7.
- [205] Florian Haftmann and Makarius Wenzel. Constructive type classes in isabelle. In *International Workshop on Types for Proofs and Programs*, pages 160–174. Springer, 2006.

- [206] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. *ACM SIGPLAN Notices*, 46(9):143–155, 2011.
- [207] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hrițcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, et al. Meta-f*: Proof automation with smt, tactics, and metaprograms. In *European Symposium on Programming*, pages 30–59, 2019.
- [208] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. 2018.
- [209] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 271–284, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325448. doi: 10.1145/2535838.2535848. URL <https://doi.org/10.1145/2535838.2535848>.
- [210] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 413–424, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737981. URL <https://doi.org/10.1145/2737924.2737981>.
- [211] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 371–384, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837625. URL <https://doi.org/10.1145/2837614.2837625>.
- [212] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The cise tool: Proving weakly-consistent applications correct. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342964. doi: 10.1145/2911151.2911160. URL <https://doi.org/10.1145/2911151.2911160>.
- [213] Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290387. URL <https://doi.org/10.1145/3290387>.
- [214] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of crdts. In Erika Ábrahám and Catuscia Palamidessi,

editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43613-4.

- [215] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In Peter Müller, editor, *Programming Languages and Systems*, pages 544–571, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44914-8.
- [216] Kartik Nagar and Suresh Jagannathan. Automated parameterized verification of crdts. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 459–477, Cham, 2019. Springer International Publishing. ISBN 978-3-030-25543-5.
- [217] Peter Sewell Xavier Leroy Robby Findler Scott Owens Brigitte Pientka Michael Hicks Talia Ringer, Benjamin C. Pierce. POPLmark 15 year retrospective panel. <https://popl20.sigplan.org/track/POPL-2020-poplmark-15-year-retrospective-panel#About>, 2020.
- [218] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.