Finding Security Bugs in Web Applications using Domain-Specific Static Analysis

by

Joseph P. Near

Submitted to the Department of Electrical Engineering and Computer Science

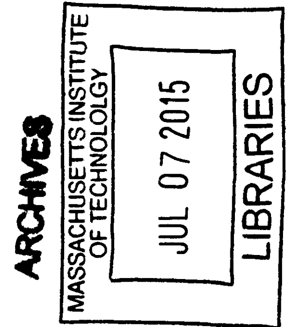
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015



© Massachusetts Institute of Technology 2015. All rights reserved.

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
May 20, 2015

Signature redacted

Certified by ....
Daniel Jackson
Professor
Thesis Supervisor

Signature redacted

Accepted by ....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Finding Security Bugs in Web Applications using Domain-Specific Static Analysis

by

Joseph P. Near

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis proposes new techniques for finding and eliminating application-specific bugs in web applications. We demonstrate three approaches to finding these bugs, each representing one position in the compromise between specificity and automation. All three are powered by a scalable symbolic execution specifically tailored to the structure of web application implementations, allowing analysis of even the largest real-world applications.

In contrast to existing general-purpose verification approaches, this work was inspired by the hypothesis that narrowing our focus might produce more effective tools. Our approach has been to take advantage of properties specific to application-specific security bugs in web applications in order to produce more effective tools. The results suggest that focusing on a particular class of applications (web applications) and on a particular class of bugs (missing security checks) we can build static analysis tools that are both significantly more scalable and more automated than general-purpose bug-finding tools.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

This research was supported in part by:

- National Science Foundation grant 0707612 (CRI: CRD - Development of Alloy Tools, Technology and Materials)
- National Science Foundation grant 0541183 (Deep and Scalable Analysis of Software)
- The Northrop Grumman Cybersecurity Research Consortium under the Secure and Dependable Systems by Design project

I am deeply grateful to the many people who made contributions to this work. Most important was Daniel Jackson's advising, which helped me navigate the murky waters of graduate school and learn how to do research effectively. I am also indebted to my thesis committee, Rob Miller and Nickolai Zeldovich, for their pointed and insightful comments.

Many other researchers have made intellectual contributions to this work; I would like to especially thank Eunsuk Kang, Rishabh Singh, and Aleksandar Milicevic, who made my research better and my life more enjoyable. I am also grateful for the support, insights, and fun conversations I have received from both other members of the Software Design Group—Santiago Perez de Rosso, Jonathan Edwards, Matt McCutchen, and Ivan Kuraj—and the wider community—especially Jean Yang and Kuat Yessenov. Finally, must I thank my wife, Marcie, for putting up with my low stipend and long hours; this research would not have been possible without her support.

Contents

1 **Introduction** **13**
1.1 Thesis Statement 14
1.2 Summary of Contributions 14
1.2.1 Limitations 15
1.3 Part I: Symbolic Execution 16
1.4 Part II: Verification 18
1.4.1 Exploring Data Exposures 18
1.4.2 Checking Code Against Security Patterns 19
1.4.3 Checking Code Against Specifications 19

I Symbolic Execution **21**

2 **Web Applications** **23**
2.1 Web Applications 23
2.1.1 HTTP and the Browser 23
2.1.2 Frameworks and REST 23
2.2 Rails 24
2.2.1 Model-View-Controller 24
2.2.2 Routing 25
2.2.3 Rendering 26
2.2.4 ActiveRecord 26
2.3 Representing Application Behavior for Security Analyses 27
2.3.1 Alloy Primer 28
2.3.2 Web Applications 28
2.3.3 Exposures 29
2.3.4 Analysis 30

3 **Symbolic Execution with an Existing Interpreter** **31**
3.1 Symbolic Execution without Side Effects 32
3.1.1 Proof of Equivalence to Standard Approach 33
3.2 Adding Side Effects 36
3.2.1 Proof of Equivalence to Standard Approach 38
3.3 Related Work 45

4 Implementation: A Symbolic Evaluator for Ruby on Rails **47**
4.1 A Simple Symbolic Evaluator 47
4.2 Conditionals 48
4.3 Side Effects 50
4.4 Stubbing Rails 54
4.5 Extracting Exposures 56
4.5.1 Rendering 56
4.5.2 Normalization 56
4.6 Challenges 57
4.7 Assumptions & Limitations 58
4.8 Soundness & Optimizations 58
4.9 Evaluation 62
4.9.1 Experimental Setup 62
4.9.2 Results 64
4.9.3 Discussion 64
4.10 Related Work 67

II Verification **69**

5 Exploring Exposures with Derailer **71**
5.1 Derailer: An Exposure Exploration Tool 72
5.2 Implementation 75
5.2.1 Exposure Generation 75
5.2.2 Exposure Visualization and Exploration 76
5.3 Evaluation 77
5.3.1 Results 78
5.3.2 Bugs Found 79
5.3.3 Comparison with Teaching Assistants 80
5.4 Related Work 80
5.4.1 Interactive Analyses 81
5.4.2 Automatic Anomaly Detection 81
5.4.3 Run-Time Approaches to Web Application Security 81

6 Checking Security Patterns with SPACE **83**
6.1 Formal Model of Access Control 84
6.1.1 Web Applications 84
6.1.2 Role-Based Access Control 85
6.1.3 Security Pattern Catalog 85
6.1.4 Mapping Application Resources to RBAC Objects 87
6.1.5 Finding Pattern Violations 88
6.2 SPACE: A Pattern-Based Bug Finder 89
6.2.1 Example Application: MediumClone 89
6.2.2 MediumClone's Security Bug 91
6.2.3 Finding the Bug Using SPACE 91

6.3 Implementation of SPACE 94
6.3.1 Extracting Exposures 94
6.3.2 Constraint Specializer 94
6.3.3 Pattern Library 96
6.3.4 Bounded Verification using the Alloy Analyzer 96
6.4 Evaluation 97
6.4.1 Bug-Finding and False Positives 98
6.4.2 Choice of Finite Bounds 101
6.5 Related Work 102

7 Full-functional Verification with Rubicon 103
7.1 Specifying Behavior 104
7.1.1 The RSpec Approach 104
7.1.2 Adding Quantifiers 105
7.1.3 The Power of Specification 108
7.1.4 Exploiting the Bug 110
7.2 Rubicon's Analysis 111
7.2.1 Rubicon's Semantics 112
7.2.2 Rubicon's Implementation 112
7.2.3 Preprocessing: Stubbing Objects 113
7.2.4 Postprocessing: Producing Alloy 114
7.2.5 An Example: Contact Permissions 115
7.3 Evaluation 117
7.3.1 Methodology 117
7.3.2 Results 118
7.3.3 Fat Free CRM Bug 121
7.4 Related Work 121

8 Conclusion 123
8.1 Discussion 123
8.2 Future Directions 125
8.2.1 Cyber-physical Systems 125
8.2.2 Access Control and APIs 126
8.2.3 Statistical Models of Behavior 126

List of Figures

1-1 Comparison of our Three Techniques, in terms of Expressive Power and Level of Automation . . . 16

2-1 Ruby on Rails Architecture . . . 25

3-1 Syntax and Reduction Rules for Untyped λ-calculus with Booleans and Natural Numbers . . . 34
3-2 Syntax and Reduction Rules for Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Booleans and Natural Numbers . . . 35
3-3 Implementation of Primitives to Achieve Mixed Concrete and Symbolic Execution of Untyped λ-calculus Under Standard Semantics . . . 35
3-4 Syntax and Reduction Rules for Untyped λ-calculus with Side Effects 39
3-5 Syntax and Reduction Rules for Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Side Effects (Part 1 of 2) . . . 40
3-6 Reduction Rules for Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Side Effects (Part 2 of 2) . . . 41
3-7 Implementation of Primitives to Achieve Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Side Effects Under Standard Semantics . . . 42

4-1 Helpers for "if" to Handle Side Effects . . . 51
4-2 Redefinition of "if" to Handle Side Effects . . . 52
4-3 Code to Stub Rails Database Accessor Methods . . . 55
4-4 Summary of Optimizations and their Effect on Soundness . . . 59
4-5 Scalability of Exposure Generation: Generation time vs application lines of code for 1000 applications . . . 63
4-6 Size of Diaspora's Codebase, by Language and Directory . . . 65
4-7 Occurrences of Language Features in Diaspora's Codebase . . . 65

5-1 Example Controller Code from Student Project . . . 72
5-2 Example Bug-finding Session using Derailer (Part 1) . . . 73
5-3 Example Bug-finding Session using Derailer (Part 2) . . . 74
5-4 Types of Bugs Found During Analysis of Student Projects . . . 77
5-5 Ratio of Exposures to Constraints in Student Projects . . . 78
5-6 Analysis Times for Student Projects . . . 79
5-7 Results of Analyzing 127 Student Projects . . . 79

6-1 Controller Code for MediumClone 90
6-2 SPACE Counterexample Showing MediumClone Security Bug: user can update another user's profile 92
6-3 SPACE Counterexample Showing a Second MediumClone Security Bug: unauthenticated user ("NoUser") can update any post 92
6-4 Summary of the Architecture of SPACE 95
6-5 Results of Running SPACE on the 50 Most-Popular Open-Source Rails Applications on Github 97
6-6 Classification of Bugs Found by Pattern Violated 99
6-7 Effect of Finite Bound on Verification Time 101

7-1 RSpec Tests for Displaying Users and Restricting Access to Private Contacts 105
7-2 Syntax of Rubicon Specifications 106
7-3 Rubicon Specifications for Displaying Users and Restricting Access to Private Contacts 107
7-4 Rubicon Specification for Displaying Contacts 109
7-5 RSpec Test for Contact Creation 110
7-6 Rubicon Specification for Contact Creation 111
7-7 Semantics of Rubicon Specifications; "b" is a Ruby Block, "p" a Predicate, "e" an Expression 112
7-8 Compiling Specifications to Abstract Syntax Trees 113
7-9 Comparison Between RSpec Execution and Rubicon Analysis 113
7-10 Compiling Abstract Syntax Trees to Alloy Specifications 114
7-11 Verification Condition and Corresponding Alloy Specification 116
7-12 Case-Study Summary: the Number of Tests, Average Runtime of Original Test and Corresponding Rubicon Specification, and Lines-of-Code Comparison Between Original Tests and Rubicon Specifications . . . 118
7-13 Range of Analysis Times for Fat Free CRM Rubicon Specifications: Average, Maximum, and Minimum Analysis Times for Each Test File 119
7-14 Effect of Finite Bound on Solving Time of Verification Conditions from Examples in Figures 3, 4, and 6 120

Chapter 1

Introduction

The web is fast becoming the most popular platform for application programming, but web applications continue to be prone to security bugs. Web apps are often implemented in dynamic languages, using relatively fragile frameworks based on metaprogramming. Most importantly, security policies themselves tend to be ad hoc, and many security bugs are the result of programmers simply forgetting to include vital security checks.

While programming frameworks and static analysis tools have begun to address those bugs—such as injection, cross-site scripting and overflow vulnerabilities—that violate generic, cross-application specifications, application-specific bugs (like missing security checks) have received less attention. Traditional solutions, such as verification and dynamic policy-enforcement techniques, ask the user to write a specification of the intended access control policy—a burdensome requirement—and have therefore seen little adoption in practice.

A report by the security research company Cenzic [1] suggests that, as of 2014, 96% of web applications contain security bugs, and nearly half of those bugs are application-specific. Moreover, a comparison of bugs found in 2013 and 2014 shows a decreasing number of cross-application bugs (like injection, cross-site scripting, and overflow vulnerabilities) over time, and an *increasing* number of application-specific bugs. This finding is in line with our experience with web frameworks and existing formal techniques, which are valuable in eliminating cross-application bugs but less useful for avoiding application-specific bugs.

This thesis proposes new techniques for finding and eliminating application-specific bugs in web applications. We demonstrate three approaches to finding these bugs, spanning the spectrum from general-purpose but burdensome to the user (meaning that the technique can find many different kinds of bugs, but requires relatively more work on the part of the user) to specific but less burdensome (meaning the technique finds only bugs from a smaller class, but requires less of the user). All three are powered by a scalable symbolic execution specifically tailored to the structure of web application implementations, allowing analysis of real-world applications.

[1] http://info.cenzic.com/rs/cenzic/images/Cenzic-Application-Vulnerability-Trends-Report-2013.pdf

1.1 Thesis Statement

Most verification approaches aim to be general-purpose, in an effort to be broadly applicable. This research, in contrast, has been successful precisely *because* it does not aspire to generality. Instead of producing general-purpose verification tools, our approach has been to take advantage of properties specific to application-specific security bugs in web applications in order to produce more effective tools. This thesis establishes the following:

> By focusing on a particular class of applications (web applications) and on a particular class of bugs (missing security checks) we can build static analysis tools that are both significantly more scalable and less burdensome to the user than general-purpose bug-finding tools.

Our hope is that this thesis will promote the discovery and application of similar domain properties in other areas, producing more effective static analysis tools that find a wide variety of bugs in many different kinds of programs.

1.2 Summary of Contributions

This thesis makes three observations:

- *Web applications are different from regular programs in ways that can be exploited to improve the scalability of symbolic execution.* First, web applications are built from *independent actions*, each of which is typically fewer than 20 lines of code; these actions can be analyzed independently. Second, web developers tend to *embed program logic in database queries*, so applications have *few conditionals* and *few loops*. These properties minimize the exponential behavior of symbolic execution, enabling the technique to scale to real-world applications.
- *Web application security policies tend to be uniform.* Sensitive data is usually subject to the same security constraints everywhere it is used, so an access that is missing one of those checks is *likely to be a mistake.*
- *Web applications share common patterns of access control.* Programmers usually select one of a handful of common access control patterns for each data type in their application. While applications often mix and match different security patterns for different kinds of resources, they usually intend for a particular pattern to be applied uniformly to all uses of a given resource type, so a possible access outside the pattern is *likely to be a mistake.*

To take advantage of these observations, we have developed the following techniques:

- *A scalable symbolic execution framework for Ruby on Rails web applications.* Our symbolic execution framework leverages our observations about the structure of web applications to scale its analysis to *even the largest* real-world applications. To cope with the challenges of Ruby's dynamic environment and the

complexity of Rails, our technique implements symbolic execution *as a library*, hijacking the standard Ruby interpreter to perform symbolic execution.

- *Derailer, a tool for exploring data exposures to find bugs.* Derailer uses our symbolic execution framework to build a list of *data exposures*—ways the application can expose information from the database—and then interacts with the user to uncover the security policy already specified by the code itself. Derailer then finds gaps in the uniform application of this policy; these gaps tend to represent security bugs.

- SPACE, *a tool for comparing application code to a catalog of security patterns.* SPACE uses symbolic execution to discover data exposures, then checks that every exposure allowed by the code is also allowed by some security pattern in our catalog. When the application allows a data exposure *not* covered by a security pattern, we report that exposure as a security bug. This process requires only that the user provide a mapping of application resources to the basic types (such as user, permission, etc.) that occur in our access control patterns. From this information alone, application-specific security bugs are then identified *automatically*, based on the predefined catalog of patterns.

- *Rubicon, a tool for comparing application code to a user-provided formal specification.* Rubicon provides a specification language extending the Rails testing framework with quantifiers, allowing programmers to write complete specifications of behavior. Rubicon uses symbolic execution to run both the specification and the application code, obtaining verification conditions necessary for establishing that the code implements the specification. Then, Rubicon uses a bounded verifier to discharge the verification conditions automatically.

The first part of this thesis describes our symbolic execution framework, while the second part explains the techniques we have built upon it to support bug finding.

The tools based on these techniques all focus on finding missing security checks, but each one makes a different compromise between expressive power and level of automation. Rubicon, for example, can be used to perform full functional verification (so it is very expressive) but it requires the user to write a complete specification of the desired behavior. Derailer, in contrast, does not require a specification, while SPACE is an attempt to find the same kinds of bugs using a built-in set of patterns. This tradeoff space is summarized in Figure 1-1.

1.2.1 Limitations

Our contributions are focused *only* on detecting missing security checks. While our symbolic execution framework may be useful for finding other kinds of security problems (or even non-security-related bugs), the tools we have developed *will* miss bugs in the following categories:

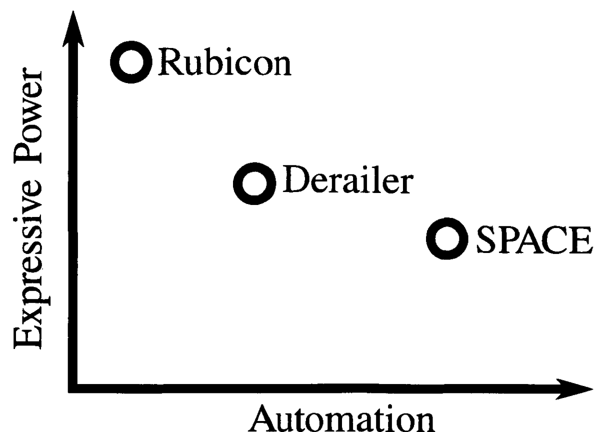


Figure 1-1: Comparison of our Three Techniques, in terms of Expressive Power and Level of Automation

- *Injection.* Injection attacks rely on a failure to sanitize inputs. Since static and dynamic techniques already exist for ensuring input sanitization, our tools do not address this problem.
- *CSRF and XSS.* Cross-site attacks represent a large portion of bugs, but can be addressed by general-purpose frameworks that automatically prevent them, and are therefore outside the scope of our tools.
- *Low-Level Bugs.* Bugs in software other than the web application itself—for example, the operating system, Ruby, Rails, or the client's browser—are also outside the scope of this work. Existing techniques tackle the problem of general software correctness, and widely-used pieces of software such as these are good targets for verification efforts.
- *Other Side-Channel Attacks.* We do not address other kinds of side-channel attacks—for example, denial of service, runtime and termination-based attacks, and client-side attacks—since there are simply too many for a single approach to reasonably detect all of them.

These limitations are, for the most part, solvable using general-purpose frameworks that automatically prevent security problems. Frameworks already exist to prevent injection, CSRF, and XSS; verification efforts have resulted in reliable operating systems, web servers, and so on. Our contributions instead target the *application-specific* security problems that existing techniques have so far failed to solve.

1.3 Part I: Symbolic Execution

Symbolic execution [20, 33] is one of the oldest strategies for reasoning about programs, and yet still forms the basis of many modern tools [11, 31, 39, 45, 44, 27].

The scalability of analyses based on symbolic execution remains a problem, however. In particular, symbolic execution of a conditional requires executing both branches of that conditional—creating the potential for an exponential number of execution paths through the program. This "path explosion" problem has limited the application of symbolic execution in practice.

Fortunately, web applications differ from traditional programs in ways that improve the scalability of symbolic execution. In particular, web applications typically use fewer loops and simpler branching structures than traditional programs, minimizing the exponential behavior of symbolic execution. Even more important, web applications are composed of independent *actions*, each of which acts like an individual program. By analyzing each action independently, our approach performs many small analyses rather than one large one—minimizing the effects of exponential behavior even when it does occur.

The second challenge of symbolic execution is building a complete evaluator that handles the entire target language and also handles concrete computation efficiently. Analysis tools based on symbolic execution, such as the symbolic extension for Java PathFinder [31, 39] and the CUTE concolic testing engine for C [45], are capable of analyzing real-world programs, but these tools are themselves large projects comprising hundreds of thousands of lines of code.

The similarity of symbolic execution and standard execution (indeed, the sharing of the very term "execution") suggests a simpler approach, in which the standard engine is used to propagate symbolic values, and to compute in the normal way with concrete values when available. The uniquely symbolic component is achieved by introducing a *library* written *in the target language itself.* Such a library comprises an encoding of symbolic values and new symbolic definitions for the primitive operations of the language, and effectively transforms the standard (concrete) implementation of the target language into a symbolic executor.

Implementing symbolic execution as a library means that concrete parts of the target program execute at full speed, just as they would during concrete execution. As a result, even large programs become amenable to symbolic execution if the number of symbolic inputs is small. At the same time, the library-based approach eliminates much of the burden of building a specialized symbolic execution engine, since a small number of primitive definitions often suffice to extend symbolic execution to the entire language.

The first part of this thesis describes a symbolic execution framework based on these two insights—that taking advantage of the implementation structure can improve scalability, and that implementing the evaluator in the language itself can improve compatibility and ease implementation. Using this approach, we have built a scalable symbolic execution system for Ruby on Rails web applications comprising fewer than 1000 lines of Ruby code. Despite its small size, this system scales to Rails applications with more than 45k lines of code, and has been used to build tools that have found previously unknown bugs in a number of open-source Rails applications.

1.4 Part II: Verification

1.4.1 Exploring Data Exposures

To find security bugs without requiring new frameworks or specifications, we built Derailer. Rather than verify an application's implementation against a specification, Derailer uses a combination of symbolic evaluation and user interaction to help the programmer discover mistakes.

This particular combination is motivated by two hypotheses. First, web applications differ from traditional programs in ways that improve the scalability of symbolic execution. In particular, web applications typically use fewer loops and simpler branching structures than traditional programs, minimizing the exponential behavior of symbolic execution. Second, security policies tend to be uniform: sensitive data is usually subject to security checks everywhere it is used, so an access that is *missing* one of those checks is likely to be a mistake.

Derailer is designed to be applied to web applications that accept requests and respond with sets of *resources* obtained by querying the database. Each response is characterized by the *path* through the database leading to the resource, and the control flow of the application's code imposes a set of *constraints* under which a particular resource is exposed to a client. We call the combination of a path and a set of constraints a *data exposure.*

An automatic strategy for finding security bugs might enforce that all exposures with the same path also share the same set of constraints; if a security check is forgotten, a constraint will be missing. But many constraints—like those used to filter sets of results for pagination—have nothing to do with security, and would cause an automatic strategy to report many false positives.

Derailer therefore asks the user to separate constraints into those representing security checks and those that are not security-related. In making this separation, the user effectively constructs a *specification* of the desired security policy—but by selecting examples, rather than writing a specification manually. Our tool makes this process easy, allowing the user to drag-and-drop constraints to build the policy. The tool then highlights exposures missing a constraint from the security policy—precisely those that might represent security bugs.

We evaluated Derailer on five open-source Rails applications and 127 student projects. The largest of the open-source applications, Diaspora, has more than 40k lines of code, and our analysis ran in 112 seconds. The student projects were taken from an access-control assignment in a web application design course at MIT. Derailer found bugs in over half of these projects; about half of those bugs were missed during manual grading. The bugs we found supported our hypothesis: most bugs were the result of either a failure to consider alternate access paths to sensitive data, or forgotten access control checks.

1.4.2 Checking Code Against Security Patterns

To further automate the discovery of missing security checks, we propose a technique for finding application-specific security bugs using a *catalog of access control patterns.* Each pattern in our catalog models a common access control use case in web applications. We built this catalog based on our experience with real-world web applications, which suggests that while applications often mix and match different security patterns for different kinds of resources, they usually intend for a particular pattern to be applied uniformly to all uses of a given resource type.

Our approach checks that for every kind of data exposure allowed by an application's code, some security pattern in our catalog also allows the exposure. When the application allows a data exposure *not* allowed by a security pattern, we report that exposure as a security bug. This process requires only that the user provide a mapping of application resources to the basic types (such as user, permission, etc.) that occur in our access control patterns. From this information alone, application-specific security bugs are then identified *automatically*, based on the predefined catalog of patterns.

We have built a prototype implementation of this technique, called SPACE (Security PAtern CheckEr). Our implementation uses symbolic execution to extract the set of all possible *data exposures* [37] from the source code of a Ruby on Rails application. The constraints associated with these exposures and the user-provided mapping are passed through a constraint specializer, which uses the mapping to re-cast the constraints in terms of the role-based access control model upon which our catalog of patterns is based. Then, SPACE translates the specialized constraints into the Alloy specification language, and uses the Alloy Analyzer to perform automatic bounded verification that each data exposure allowed by the application is also allowed by a security pattern in our catalog.

Of the 50 most popular open-source Rails applications on Github, 30 implement access control. We have used SPACE to find security bugs in nearly 1/3 of these—a total of 23 unique bugs. Both the symbolic execution and bounded verification steps of our technique scale well to applications as large as 45k lines of code—none of our analyses took longer than 64 seconds to finish.

1.4.3 Checking Code Against Specifications

To provide full-functional verification of web application code, we developed Rubicon, a bounded verifier for Ruby on Rails applications. Rubicon allows programmers to write specifications of the behavior of their web application and performs automatic bounded analysis to check those specifications against the implementation. Rubicon aims to be both powerful and easy to use: its specification language is expressive but based on a popular domain-specific testing language, and its analysis is implemented as a Ruby library.

Rubicon's specification language extends the RSpec testing language [16] with the quantifiers of first-order logic, allowing programmers to replace RSpec tests over a set of mock objects with general specifications over all objects. This compatibility with

the existing RSpec language allows converting test cases into specifications.

Rubicon's automated analysis comprises two parts. First, Rubicon uses our symbolic execution framework to generate verification conditions from the code and specifications; second, it invokes a constraint solver to check those conditions. To check the verification conditions, Rubicon compiles them into Alloy [30], a lightweight specification language whose analyzer is an automatic, bounded model finder for relational first-order logic. Alloy's logic is a good match because its semantics closely match those of relational databases, but the solving of the verification conditions is a separate problem, and in principle might be handled with a different technology (e.g. an SMT solver or theorem prover).

We evaluated Rubicon on five open-source web applications for which the original developers had already written RSpec tests. We converted a random sample of these tests into Rubicon specifications; in every case, Rubicon's analysis took no more than a few seconds per specification. In the largest of these applications, a customer relationship management system called Fat Free CRM, Rubicon's analysis uncovered a previously unknown security bug. The authors of Fat Free CRM have acknowledged and fixed this bug.

Part I

Symbolic Execution

Chapter 2

Web Applications

2.1 Web Applications

Web applications are distributed applications consisting of a *server* (typically a web server) and a number of *clients* (each of which is typically a web browser). In modern web applications, the application developer writes code that executes both on the client (within the browser) and on the server. Clients communicate with the server via HTTP requests.

2.1.1 HTTP and the Browser

The HTTP architecture was designed to be stateless and synchronous. Clients would issue HTTP requests to the server, which would respond with HTML representing a single web page. Since an interactive distributed application requires both clients and servers to save some state between requests, a database is typically used as a persistent data store connected to the web server. This strategy allows the server to store session information in the database, so that the server can remember which user is logged in, what items are in the user's shopping cart, and so on. This basic architecture of a web server, database, and client web browser is shared by all web applications.

The web browser used by the client traditionally had predefined behavior (being able to issue requests only according to the HTTP standard), and was incapable of executing arbitrary code. Modern developments include sophisticated client-side programming capabilities, allowing the browser to execute arbitrary Javascript code issued by the server. This client-side code has the ability not only to modify the document presented to the user, but also to issue new HTTP requests to the server and act on the responses, enabling rich new applications like Gmail and Facebook.

2.1.2 Frameworks and REST

Modern web applications are typically written with the aid of a *framework*, which imposes a particular structure on the implementation of an application. A common

approach views the application as defining a set of *resources* and an API for performing operations on those resources. To implement the API, the application defines a *controller* for each resource type, and within each controller, an *action* for each API operation.

The most common model for defining resources and building an API is called REST (Representational State Transfer). REST specifies a structure for the URLs used to invoke the most common elements of resource APIs, and then defines the semantics of each HTTP verb on each of those URLs. To these standard API operations, the application adds its own composite operations to extend the API.

For example, for a blog post resource, REST defines a resource collection URL (e.g. http://example.com/posts) for listing all blog posts, and a resource element URL (e.g. http://example.com/posts/1) for accessing a particular post by its ID. REST also defines the semantics of the four HTTP verbs (GET, PUT, POST, and DELETE) on these URLs:

- GET displays the resource element or lists the resource collection
- PUT updates the resource element or replaces the collection with another
- POST makes a new resource element or adds to a collection
- DELETE deletes a resource element or collection

2.2 Rails

Rails[1] is a modern web programming framework implemented in the Ruby programming language. Rails uses a *model-view-controller* (MVC) strategy to enforce separation of concerns; it allows the programmer to define application resources via *models*; and it provides an *object-relational mapper* called ActiveRecord to allow these resources to be stored in and retrieved from a persistent database.

Figure 2-1 contains a summary of how a request is handled by the Rails architecture. Each request is sent to the router, which decides which controller action to invoke. The controller may call some methods of the model, which may in turn use the ActiveRecord API to perform database accesses. The controller also calls the rendering engine, specifying which view to render; the rendering engine loads the appropriate view template, runs the Ruby expressions embedded in it, and responds to the client with a rendered page.

2.2.1 Model-View-Controller

Model. Rails programmers define application resources using *model classes.* These are implemented as class definitions in Ruby, but represent composite conceptual resources. Each model class defines a set of relationships to other resources (e.g.

[1] http://rubyonrails.org/

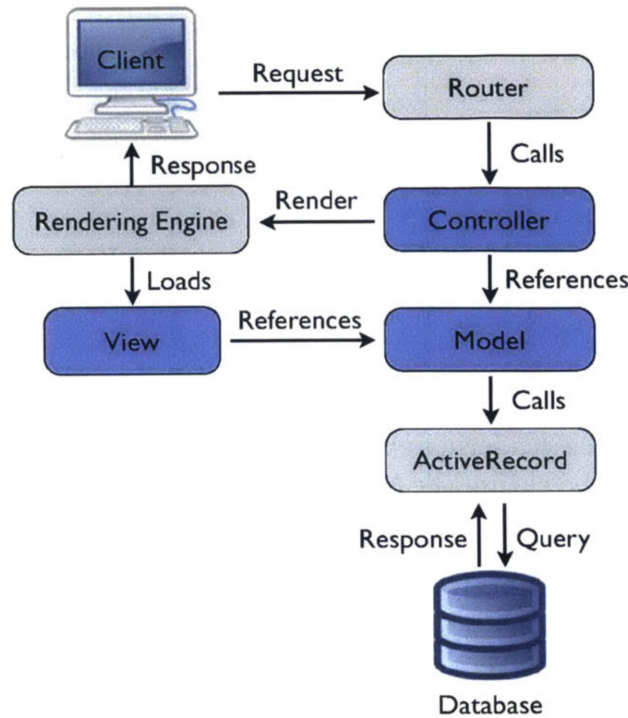


Figure 2-1: Ruby on Rails Architecture

a user *has many* blog posts), allowing the programmer to build complex resource structures and treat them in the same way as composite objects in traditional object-oriented programming.

Model classes also contain definitions of model operations, which are private operations exposed to the controller but not part of the public API. For example, a shopping cart might define a model operation to calculate the total price of the items in the cart.

View. The view component is defined using the Rails *templating* system. This system associates a template with the output of each action; these templates may contain Ruby expressions, and the rendering engine executes these expressions in the context of the action's results to produce a final rendered page to send to the client.

2.2.2 Routing

To assign requests to the controller actions that handle them, Rails provides a *router*. The routing system is configured via a set of rules that map URLs to controller actions. These rules are specified via a syntax that encourages the use of RESTful API practices for the application's resources: a complete set of URL routes for a particular resource can be specified using the form "resource :User." This specification results in mappings like the following:

- GET http://example.com/users → UsersController/index

- GET http://example.com/users/1 → UsersController/show(:id => 1)
- POST http://example.com/users/1/update → UsersController/update(:id => 1 ...)

The Rails router also allows one-off mappings like "get 'login', :to 'users#login'," which results in the following:

- GET http://example.com/login → UsersController/login

2.2.3 Rendering

The Rails rendering system transforms templates into HTML strings ready to be sent back to a client. The rendering engine is responsible for finding the appropriate template for rendering, executing the Ruby expressions that are part of the template, and integrating the resulting Ruby values into the rendered HTML.

The rendering engine picks a template to render based on the form of the request and the call to "render" in the controller action. The client may request a standard HTML response, or a response in XML or JSON format. The rendering engine picks the appropriate template based on what rendering formats are available.

The rendering engine then executes the Ruby expressions embedded in the template in the context of the instance variables defined by the controller action and the Rails environment corresponding to the current response. Some of these embedded expressions result in string values, and those are simply inserted into the resulting HTML. Others, however, may reference other templates or call special helpers (defined by the Rails API) for constructing HTML forms or links. These helper functions reference a part of the environment to produce HTML either by rendering another template (in the case of template references) or by constructing the HTML directly (in the case of form and link helpers).

2.2.4 ActiveRecord

ActiveRecord is the Rails *object-relational mapper*. It allows the programmer to pretend that the database contains Ruby objects: objects appear to be stored in the database directly, and queries result in lists of objects. ActiveRecord uses the application's set of model classes to define a database schema for storing objects of those classes in the database, and to map fields of those classes to database tables.

ActiveRecord provides an object-based query API, allowing the programmer to write queries like "User.find_by(:name => "Joe")." This query results in a list of User objects with the name "Joe." The query API is designed to replace SQL queries entirely, so that Rails programmers are never required to write SQL.

Through *associations*, ActiveRecord provides the illusion of composite objects. The programmer specifies the association in the model class definition using keywords like *belongs_to*; ActiveRecord defines new database columns with foreign keys linking to the referenced object, and adds a field referencing the other object. For example, given the following model classes:

```
class Customer < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

the programmer can find all of a customer's orders by writing:

```
@customer.orders
```

exactly as if the customer object had a field referencing an actual order object. ActiveRecord accomplishes this by adding a column "customer_id" to the *Orders* database table, and transforming the above expression into a database query that looks up orders with customer_id = @customer.id.

2.3 Representing Application Behavior for Security Analyses

Most web applications have two security goals: first, *data integrity*: that the information in the database remains uncorrupted by changes not allowed by the security policy; and second, *privacy*: that information in the database is only exposed to users of the application according to the security policy.

The distributed nature of web applications makes accomplishing these goals difficult. In contrast to desktop applications, web applications store all user data centrally, so a bug in enforcing the security policy could result in the exposure of every user's data at once.

Our model uses the idea of a *data exposure* to characterize the behavior of a web application with respect to data integrity and privacy. In this model,

- the database stores *resources*
- the application API *exposes* resources
- each *data exposure* characterizes the exposure to a user of an application resource

Our representation of data exposures includes:

- the *type* of resource exposed
- the *path* through the database (i.e. database query) to retrieve the resource
- the *constraints* the application enforces on exposure of the resource

This representation allows a simple characterization of the behavior of an application with respect to the security properties of data integrity and privacy, eliminating the state changes usually present in imperative implementations of web applications. The set of exposures characterizing an application is suitable for direct comparison against a declaratively-specified security policy, for example, to determine if the application allows exposing some data that the policy says should be private. Data exposures are the common intermediate format supporting all three of the tools explained in the second part of this thesis, and our implementation is specifically designed to produce the set of exposures efficiently.

In the rest of this section, we formalize the notion of application and exposure, as well as the desired semantics of the symbolic execution we will use to transform application code into a set of data exposures.

2.3.1 Alloy Primer

The model is given in Alloy [30]. For readers unfamiliar with Alloy, the following points may help. A signature (introduced by keyword sig) introduces a set of objects; each field of a signature introduces a relation whose first column is the set associated with the signature, and whose remaining columns are as declared. Thus the declaration

```
sig Request {params: Param → Value}
```

introduces a set "Request" (of request objects), and a ternary relation "params" on the sets *Request*, *Param* and *Value*; this relation can viewed as a table with three columns. A tuple *(r,p,v)* in this relation would indicate that in request *r*, parameter *p* has value *v*. Equivalently, the signature can be thought of as a class with the fields as instance variables; thus this field declaration introduces, for each request *r*, a mapping *r.params* from parameters to values.

Signature extension introduces subsets. Thus

```
sig ValueResource extends Resource {value: Value}
```

says that some resources are value resources, and introduces a relation called "value" from value resources to values. Equivalently, the subsignature can be viewed as if it were a subclass; thus a value resource *vr* has a value *vr.value.*

2.3.2 Web Applications

Clients issue requests that contain a binding of parameters to values, and a choice of action:

```
sig Request {
  params: Param → Value,
  action: Action
}
```

There is no need to distinguish clients or represent client-side state (such as cookies), since the analysis must be conservative and assume the worst (e.g. that a client

could manipulate a cookie). The model does not currently allow for access control through client-side certificates or reliance on signed cookies. The choice of HTTP method (eg, GET or POST) need not be modeled, nor whether the request is synchronous or asynchronous, since these factors do not impact what we seek to analyze (namely what data is released in response to a query). Nor do we need to distinguish how the parameters are passed (in a form, query string, or JSON object, eg); in Rails, and many other web frameworks, the request is accessed homogeneously through a single hashmap.

The response to a request is just a set of resources (to be elaborated shortly):

```
sig Response {resources: set Resource}
```

The internal state of the application is just a database mapping paths to resources:

```
sig Database {resources: DBPath → Resource}
```

A path is an abstraction of a general database query, representing a *navigation* through the database's tables using only the relational join. Such queries can be used to extract any resource the database contains, and filtered to contain only the desired results.

To represent these filters, we introduce constraints. A constraint has a left and right side, each of which may be a path, a parameter or a value, and a comparison operator:

```
sig Constraint {
  left, right: DBPath + Param + Value,
  operator: Operator
}
```

(In fact, constraints can have logical structure, and our implementation puts constraints into conjunctive normal form. This detail is not relevant, however, to understanding the essence of the approach.)

The behavior of an application can now be described in terms of two relations. Both involve a database (representing the pre-state, before execution of the action) and an incoming request. The first relates these to the resulting response, and the second to a database (representing the post-state, after execution of the action):

```
sig App {
  response: Database → Request → Response,
  update: Database → Request → Database
}
```

2.3.3 Exposures

An approximation to this behavior is inferred by static analysis of the code, and consists of a set of "exposures" of resources, with an exposure consisting of a path, an action, and a set of constraints:

```
sig Report {
  exposures: set Exposure
}
sig Exposure {
  path: DBPath,
  action: Action,
  constraints: set Constraint
}
```

The presence of an exposure in the report means that a set of resources might be exposed under the given constraints.

Example. The exposure with path User.notes.content, action update, and constraints User.notes. title = notetitle and User.notes.owner = session.user would represent the set of content strings that might be exposed when the update action is executed. The constraints limit the notes to those with a title matching the notetitle parameter and that are owned by the currently logged in user. The set of notes u.notes associated with a user u need not, of course, have user u as their owner; a constraint such as the one we have here would typically be used to ensure that while a user can read notes shared by others, she can only modify notes she owns.

2.3.4 Analysis

Our analysis uses the application's code to obtain a set of exposures. More precisely, it produces a superset of the exposures for which some concrete database and request exists such that the application produces the concrete results represented by the exposure.

```
fun symbolic_analysis[app: App]: set Exposure {
  {e: Exposure |
    some db: Database, request: Request {
      db.resources[e.path] in app.response[db,
        request].resources
      request.action = e.action
      e.constraints = {c: Constraint | holds[c, app]}
    }
}
```

This specification says that for each exposure, some concrete database and request exist such that (1) the application responds with the same resource as specified by the exposure, (2) the exposure's action matches that of the request, and (3) the constraints associated with the exposure are those enforced by the application's code. Our model does not define the *holds* predicate, since it depends on the semantics of the application's implementation language and on the particular representation of constraints. Symbolic execution satisfies this specification, since it uses the application code directly to build the set of exposed resources and the constraints associated with them.

Chapter 3

Symbolic Execution with an Existing Interpreter

Symbolic execution is the practice of executing programs with symbols, rather than values, provided as inputs. Executing a program symbolically in turn leads to outputs that are symbolic expressions rather than values. The traditional formulation of symbolic execution for languages with mutable state calls for a symbolic representation of the program's state during execution that includes both the possible values of the symbolic variables and a *path condition*—a boolean formula representing the conditions necessary for a particular variable to take a certain value.

Despite its status as one of the oldest strategies for reasoning about programs [20, 33], symbolic execution remains popular as the basis of many modern tools [11, 31, 39, 45, 44, 27].

Building a symbolic executor, however, is difficult, and building one that also handles concrete computation efficiently is more difficult still. Analysis tools based on symbolic execution, such as the symbolic extension for Java PathFinder [31, 39] and the CUTE concolic testing engine for C [45], are capable of analyzing real-world programs, but these tools are themselves large projects comprising hundreds of thousands of lines of code.

The similarity of symbolic execution and standard execution (indeed, the sharing of the very term "execution") suggests a simpler approach, in which the standard engine is used to propagate symbolic values, and to compute in the normal way with concrete values when available. The uniquely symbolic component is achieved by introducing a *library* written *in the target language itself.* Such a library comprises an encoding of symbolic values and new symbolic definitions for the primitive operations of the language, and effectively transforms the standard (concrete) implementation of the target language into a symbolic executor.

In this chapter, we formalize this technique and show that it results in a symbolic evaluator with equivalent semantics to the traditional strategy. We begin by formally defining symbolic execution for the side-effect-free untyped λ-calculus, and then show how the same results can be achieved without modifying the language's semantics. Next, we demonstrate the same progression in the more complicated setting of the untyped λ-calculus with side effects.

3.1 Symbolic Execution without Side Effects

In this section, we demonstrate our approach in the simplest context by applying it to a minimal language without side effects: the untyped λ-calculus. Our goal is to define both standard symbolic execution and our new approach formally, then prove their equivalence. This process involves 3 steps:

1. We define the target language formally.
2. We extend the set of values with symbolic ones and the set of inference rules to allow computing with those symbolic values, defining the standard approach to symbolic execution.
3. We define a set of operators *in the target language* to compute with symbols, and use them to perform symbolic execution.
4. We prove that the set of possible executions defined in step 2 (the standard symbolic execution semantics) is the same as the set defined in step 3 (our new approach).

The original semantics of the untyped λ-calculus are drawn from Pierce [40], and appear in Figure 3-1. We formalize the standard notion of symbolic execution (the second step) in Figure 3-2. This formalization adds symbolic values to the existing concrete ones, and adds evaluation rules for terms containing symbolic values. We use s to denote the class of symbolic values, which may be either symbolic variables or symbolic expressions containing an arbitrary number of symbolic or concrete values. The result of a symbolic execution under these semantics will be a concrete value if no symbolic values are involved, or a symbolic expression if computation using symbols is performed.

The key difference between the symbolic semantics and the concrete semantics is that the symbolic semantics must execute both branches of a conditional dependent on a symbolic value. This is accomplished by the rules IFSYMB1 and IFSYMB2 in Figure 3-2.

Because the target language omits side effects, no notion of symbolic state or a global path constraint is required. The resulting symbolic expression itself represents the condition necessary for the given term to yield a particular value. For example, consider the following term and its value under the symbolic semantics (where $\overset{*}{\rightarrow}$ represents zero or more reductions):

$$\begin{aligned} t &= \textsf{if iszero } s_1 \textsf{then 0 else succ 0} \\ &\overset{*}{\rightarrow} \textsf{Exp(if, Exp(iszero, } s_1\textsf{), 0, succ 0)} \end{aligned}$$

The resulting symbolic expression alone represents the possible results of executing t concretely: if the input variable is zero, then the result is zero; otherwise, it is zero's successor.

Figure 3-3 contains our approach to symbolic execution for the pure untyped λ-calculus (the third step). Our approach comprises a set of symbolic values—identical to the ones added in Figure 3-2—and a set of redefinitions for the primitive operations

of the language. Aside from the addition of symbolic values, this implementation can be executed directly under the standard semantics in Figure 3-1. This approach also works in languages without symbolic values, by encoding symbolic values in a form the language does support. Our implementation in Ruby, for example, uses a special set of classes to represent symbolic values.

Using our approach also requires the ability to selectively redefine language primitives, including "if." Many existing languages make this possible, and there are workarounds for some of those (like Ruby) that do not. When programs are side-effect free, invocations of these primitives can be either call-by-value or call-by-name; when side effects are introduced, call-by-name must be used.

Our approach produces the same results as the traditional symbolic semantics shown in Figure 3-2. The example given above, for example, evaluates as follows under the standard semantics with our redefinitions:

$$
\begin{array}{lll}
\mathsf{t} & = & \mathsf{if\ iszero}\ s_1\ \mathsf{then\ 0\ else\ succ\ 0} \\
& \overset{*}{\rightarrow}_\theta & \mathsf{if\ sym?\ Exp(iszero,}\ s_1\mathsf{)} \\
& & \quad \mathsf{then\ Exp(if,\ Exp(iszero,}\ s_1\mathsf{),\ 0,\ succ\ 0)} \\
& & \quad \mathsf{else\ if\ (iszero}\ s_1\mathsf{)\ then\ 0\ else\ succ\ 0} \\
& \rightarrow_\theta & \mathsf{if\ true} \\
& & \quad \mathsf{then\ Exp(if,\ Exp(iszero,}\ s_1\mathsf{),\ 0,\ succ\ 0)} \\
& & \quad \mathsf{else\ if\ (iszero}\ s_1\mathsf{)\ then\ 0\ else\ succ\ 0} \\
& \rightarrow_\theta & \mathsf{then\ Exp(if,\ Exp(iszero,}\ s_1\mathsf{),\ 0,\ succ\ 0)}
\end{array}
$$

3.1.1 Proof of Equivalence to Standard Approach

The proof that our approach corresponds to the symbolic semantics is straightforward, and is accomplished by induction on terms. A short version of this proof, considering only the relevant cases, follows.

Theorem 3.1.1 *Let $\rightarrow$ be the transition relation of the symbolic semantics described in Figure 3-2, and let $\rightarrow_\theta$ be the transition relation of the standard semantics plus redefinitions of primitive operations described in Figure 3-3. Then for all terms t, $t \overset{*}{\rightarrow} t' \iff t \overset{*}{\rightarrow}_\theta t'$.*

In other words, if the standard definition of symbolic execution ($\overset{*}{\rightarrow}$) allows reducing a term t to another term t', then our new approach ($\overset{*}{\rightarrow}_\theta$) allows the same reduction, and vice versa. The proof of this property considers two important classes of cases: first, conditionals, where we show that calling our redefined "if" yields the same "Exp" expression as the symbolic semantics; and second, built-in functions, where we show that calling our redefined versions also produces the same expression.

Proof By induction on t, considering the relevant cases.

- $t =$ if sv_1 then v_2 else v_3. We have:

$$
\begin{array}{rcl}
t & ::= & x \mid \lambda x.t \mid t\ t \\
  & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\
  & \mid & 0 \mid \mathsf{succ}\ t \mid \mathsf{pred}\ t \mid \mathsf{iszero}\ t \\
\\
v & ::= & \lambda x.t \\
  & \mid & \mathsf{true} \mid \mathsf{false} \\
  & \mid & nv \\
\\
nv & ::= & 0 \mid \mathsf{succ}\ nv
\end{array}
$$

$$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2} \quad (\textsc{App1}) \qquad \frac{t_2 \rightarrow t_2'}{v\ t_2 \rightarrow v\ t_2'} \quad (\textsc{App2})$$

$$(\lambda x.t)v \rightarrow [x \mapsto v]t \quad (\textsc{AppAbs})$$

$$\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \rightarrow t_2 \quad (\textsc{IfTrue})$$

$$\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \rightarrow t_3 \quad (\textsc{IfFalse})$$

$$\frac{t_1 \rightarrow t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3} \quad (\textsc{If})$$

$$\mathsf{pred}\ 0 \rightarrow 0 \quad (\textsc{PredZero})$$

$$\mathsf{iszero}\ 0 \rightarrow \mathsf{true} \quad (\textsc{IsZeroZero})$$

$$\mathsf{iszero}\ (\mathsf{succ}\ nv_1) \rightarrow \mathsf{false} \quad (\textsc{IsZeroSucc})$$

$$\mathsf{pred}\ (\mathsf{succ}\ nv_1) \rightarrow nv_1 \quad (\textsc{PredSucc})$$

$$\frac{t_1 \rightarrow t_1'}{\mathsf{succ}\ t_1 \rightarrow \mathsf{succ}\ t_1'} \quad (\textsc{Succ})$$

$$\frac{t_1 \rightarrow t_1'}{\mathsf{pred}\ t_1 \rightarrow \mathsf{pred}\ t_1'} \quad (\textsc{Pred})$$

Figure 3-1: Syntax and Reduction Rules for Untyped λ-calculus with Booleans and Natural Numbers

$$
\begin{array}{rcl}
v & ::= & \ldots \\
  & | & sv \\
\\
sv & ::= & sx \mid \mathsf{Exp}\ \ v^*
\end{array}
$$

$$\frac{t_2 \to t_2'}{\mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \to \mathsf{if}\ sv_1\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3} \quad (\textsc{IfSymb1})$$

$$\frac{t_3 \to t_3'}{\mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ t_3 \to \mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ t_3'} \quad (\textsc{IfSymb2})$$

$$\mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ v_3 \to \mathsf{Exp}(\mathsf{if},\ sv_1, v_2, v_3) \quad (\textsc{IfSymb})$$

$$\mathsf{pred}\ sv_1 \to \mathsf{Exp}(\mathsf{pred},\ sv_1) \quad (\textsc{PredSymb})$$

$$\mathsf{succ}\ sv_1 \to \mathsf{Exp}(\mathsf{succ},\ sv_1) \quad (\textsc{SuccSymb})$$

$$\mathsf{iszero}\ sv_1 \to \mathsf{Exp}(\mathsf{iszero},\ sv_1) \quad (\textsc{IsZeroSymb})$$

Figure 3-2: Syntax and Reduction Rules for Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Booleans and Natural Numbers

$$
\begin{array}{rcl}
v & ::= & \ldots \\
  & | & sv \\
\\
sv & ::= & sx \mid \mathsf{Exp}\ \ v^*
\end{array}
$$

$$\mathsf{sym?}\ sv_1 \to \mathsf{true} \quad (\textsc{Sym})$$

$$\begin{array}{c}\mathsf{sym?}\ v_1 \to \mathsf{false} \\ \text{where } v_1 \notin sv\end{array} \quad (\textsc{NotSym})$$

$$\frac{t_1 \to t_1'}{\mathsf{sym?}\ t_1 \to \mathsf{sym?}\ t_1'} \quad (\textsc{Sym2})$$

```
if     = λ t,c,a. if sym? t then Exp(if, t, c, a)
                  else if t then c else a
pred   = λ v. if sym? v then Exp(pred, v) else pred v
succ   = λ v. if sym? v then Exp(succ, v) else succ v
iszero = λ v. if sym? v then Exp(iszero, v) else iszero v
```

Figure 3-3: Implementation of Primitives to Achieve Mixed Concrete and Symbolic Execution of Untyped λ-calculus Under Standard Semantics

$$t \xrightarrow{*} \mathsf{Exp}(\mathsf{if}, sv_1, v_2, v_3)$$

and:

$$\begin{aligned} t &\rightarrow_\theta \mathsf{if\ sym?}\ sv_1\ \mathsf{then\ Exp(if,}\ sv_1, v_2, v_3) \\ &\quad \mathsf{else\ if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ v_3 \\ &\xrightarrow{*}_\theta \mathsf{Exp}(\mathsf{if}, sv_1, v_2, v_3) \end{aligned}$$

which are equivalent.

- $t = \mathsf{pred}\ sv_1$. We have:

$$t \xrightarrow{*} \mathsf{Exp}(\mathsf{pred}, sv_1)$$

and:

$$\begin{aligned} t &\rightarrow_\theta \mathsf{if\ sym?}\ sv_1\ \mathsf{then\ Exp(pred,}\ sv_1)\ \mathsf{else\ pred}\ sv_1 \\ &\xrightarrow{*}_\theta \mathsf{Exp}(\mathsf{pred}, sv_1) \end{aligned}$$

which are equivalent.

- Similarly for succ and iszero. ∎

3.2 Adding Side Effects

We now turn our attention to languages with side effects. We present the standard semantics for the untyped λ-calculus with side effects in Figure 3-4. This formalization of mutable state introduces the set l of labels and the store μ to hold a mapping from labels to values. The corresponding reduction rules update the store as a given term is reduced, and the final result of a program is represented by both the fully-reduced value of the term and the final value of the store.

This new style of execution makes symbolic execution more difficult. Given symbolic inputs, a program with side effects should produce both a symbolic value and a store—but the value of the store depends on the path taken through the program. To perform symbolic execution, we introduce a new *symbolic store* σ that represents all the possible values a symbolic variable *could* take, and also records the conditions necessary for the variable to take each of those values.

Each condition recorded in the symbolic store represents a single path through the program, and is therefore called a *path constraint.* Symbolic execution keeps track of the current path constraint during execution, and uses that path constraint when updating the symbolic store.

We formalize the symbolic semantics with side effects in Figures 3-5 and 3-6. We call the current path constraint ϕ, and the symbolic store is σ. The majority of the reduction rules correspond to those of the standard semantics in Figure 3-4, except that the new rules propagate the values of ϕ and σ.

The rules for handling assignment and conditionals have changed significantly to deal with symbolic values. Intuitively, the rule for "if" must execute both branches of the conditional, constructing the appropriate path constraint for each branch. Rules

IFSYMB1 and IFSYMB2 perform this task, using the condition's value to extend the path constraint. When both branches have evaluated to values, IFSYMB transforms the conditional into a symbolic expression.

The rules for assignment are responsible for extending and merging symbolic states. The ASSIGN rule handles the entirely concrete case, and operates just as before. ASSIGNSYMB1 handles situations in which the variable being assigned to is not symbolic, but its new value is dependent on a symbolic value, as in the following program:

```
x := 5;
if sv then x := 6
```

In this case, "x" must take a symbolic value, even though it is only assigned concrete values, since its value is dependent on the symbolic value "sv." ASSIGNSYMB1 constructs a new symbolic variable for this purpose, assigns that symbolic variable to the given location, and adds both possible values for the variable to the symbolic state.

The final case, handled by ASSIGNSYMB2, is the situation in which the target of an assignment is already symbolic. Consider the following program, for example:

```
x := 5;
if sv then x := 6
else x := 7
```

After executing the first assignment, the symbolic state for "x" will be:

$$(\mathsf{sv} \Rightarrow 6), (\mathsf{true} \wedge \neg\mathsf{sv} \Rightarrow 5)$$

For the second assignment, since "x" already has a symbolic value, we take its symbolic state and duplicate it. One copy of the symbolic state has its path conditions conjoined with the current path condition, and its values replaced with the value being assigned (this part of the symbolic state represents all possible paths through the program *that end up going through the current path*). The other copy has its path conditions conjoined with the *negation* of the current path condition, and its values remain unchanged (this part of the symbolic state represents the possible paths through the program that do *not* end up going through the current path). After the second assignment, then, the symbolic state for "x" is:

$$(\mathsf{sv} \wedge \neg\mathsf{sv} \Rightarrow 7), (\mathsf{true} \wedge \neg\mathsf{sv} \wedge \neg\mathsf{sv} \Rightarrow 7),$$
$$(\mathsf{sv} \wedge \mathsf{sv} \Rightarrow 6), (\mathsf{true} \wedge \neg\mathsf{sv} \wedge \mathsf{sv} \Rightarrow 5)$$

Of these possible outcomes, the first and the last are impossible, reflecting the fact that there is no way to take more than one path through the program simultaneously, and making sure that some path is taken (as a result, it is impossible for "x" to have the final value 5). The rule ASSIGNSYMB2 implements this duplication and updating process on the symbolic state, computing a σ' that correctly merges the possible symbolic states.

The end result of executing a program with symbolic inputs is a symbolic value, a store μ mapping labels to symbolic variables or concrete values, and a symbolic store σ mapping symbolic variables to sets of values paired with path conditions.

Figure 3-7 contains our redefinitions of operations that produce the same results as the symbolic semantics. Like the symbolic semantics, these redefinitions keep track of the current path constraint and symbolic state; in the absence of semantic constructs, however, these elements are encoded in the target language. The path constraint is stored in a global variable "pc," while the symbolic state is encoded as a data structure tagged with the unique value we label "SYM_TAG" and is treated symbolically by the redefined primitives.

Just as in the symbolic semantics, the major changes occur in the definitions of assignment and conditionals. The definition of assignment performs the same additions to the symbolic state as the rules of the symbolic semantics do, but the redefinition places these changes in tagged data structures inside the store, rather than in σ. Similarly, the redefinition of conditionals modifies the path constraint by updating its value in the store before executing the first branch of the conditional, updates the path constraint again before executing the second branch, and resets it before returning.

3.2.1 Proof of Equivalence to Standard Approach

Theorem 3.2.1 *Let $\rightarrow$ be the transition relation of the symbolic semantics described in Figure 3-5, let $\rightarrow_\theta$ be the transition relation of the standard semantics plus redefinitions of primitive operations described in Figure 3-7, and let γ be a concretization function encoding the contents of the store and symbolic state such that $\gamma(\mu,\sigma)(l) = (\mathsf{SYM_TAG}, \sigma(\mu(l)))$ if $\mu(l) \in sx$, and $\gamma(\mu,\sigma)(l) = \mu(l)$ otherwise. Then for all terms t, $\phi \vdash t|\mu,\sigma \xrightarrow{*} t'|\mu',\sigma' \iff t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \xrightarrow{*}_\theta t'|(\gamma(\mu',\sigma'), \mathsf{pc} = \phi')$.*

In other words, given a function γ that turns a symbolic state into the concrete representation that our redefinitions use, reducing a term t to a new term t' (plus new values μ' and σ' for the store and symbolic state) using the standard approach ($\xrightarrow{*}$) is equivalent to performing the same reduction using our approach ($\xrightarrow{*}_\theta$) and then applying γ. The proof considers four important cases: one involving conditionals and three involving side effects.

1. We show that our redefinition of "if":
    - Uses recursive calls to execute both branches; by the inductive hypothesis, we determine that these recursive calls produce the same results as the symbolic semantics.
    - Updates the symbolic correctly (as defined by γ).
    - Updates the current path constraint correctly.
2. We show that when the path constraint is "true," updating a concrete variable causes an update in the regular (non-symbolic) store in both the symbolic semantics and our approach—in other words, everything remains concrete.

$$
\begin{array}{lcl}
t & ::= & \ldots \\
 & | & \textsf{ref}\ t \ \mid\ !\ t \ \mid\ t := t \ \mid\ l \\
v & ::= & \ldots \\
 & | & \textsf{unit} \ \mid\ l \\
\mu & ::= & \emptyset \ \mid\ \mu, l = v
\end{array}
$$

$$\frac{t_1|\mu \to t_1'|\mu'}{t_1 t_2|\mu \to t_1' t_2|\mu'} \quad (\textsc{App1}) \qquad \frac{t_2|\mu \to t_2'|\mu'}{v\ t_2|\mu \to v\ t_2'|\mu'} \quad (\textsc{App2})$$

$$(\lambda x.t)v|\mu \to [x \mapsto v]t|\mu \quad (\textsc{AppAbs})$$

$$\textsf{if true then } t_2 \textsf{ else } t_3|\mu \to t_2|\mu \quad (\textsc{IfTrue})$$

$$\textsf{if false then } t_2 \textsf{ else } t_3|\mu \to t_3|\mu \quad (\textsc{IfFalse})$$

$$\frac{t_1|\mu \to t_1'|\mu'}{\textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3|\mu \to \textsf{if } t_1' \textsf{ then } t_2 \textsf{ else } t_3|\mu'} \quad (\textsc{If})$$

$$\textsf{pred } 0|\mu \to 0|\mu \quad (\textsc{PredZero})$$

$$\textsf{iszero } 0|\mu \to \textsf{true}|\mu \quad (\textsc{IsZeroZero})$$

$$\textsf{iszero } (\textsf{succ } nv_1)|\mu \to \textsf{false}|\mu \quad (\textsc{IsZeroSucc})$$

$$\textsf{pred } (\textsf{succ } nv_1)|\mu \to nv_1|\mu \quad (\textsc{PredSucc})$$

$$\frac{t_1|\mu \to t_1'|\mu'}{\textsf{succ } t_1|\mu \to \textsf{succ } t_1'|\mu'} \quad (\textsc{Succ})$$

$$\frac{t_1|\mu \to t_1'|\mu'}{\textsf{pred } t_1|\mu \to \textsf{pred } t_1'|\mu'} \quad (\textsc{Pred})$$

$$\frac{l \notin dom(\mu)}{\textsf{ref } v_1|\mu \to l|(\mu, l \mapsto v_1)} \quad (\textsc{RefV})$$

$$\frac{t_1|\mu \to t_1'|\mu'}{\textsf{ref } t_1|\mu \to \textsf{ref } t_1'|\mu'} \quad (\textsc{Ref})$$

$$\frac{\mu(l) = v}{!l|\mu \to v|\mu} \quad (\textsc{DerefLoc})$$

$$\frac{t_1|\mu \to t_1'|\mu'}{!t_1|\mu \to !t_1'|\mu'} \quad (\textsc{Deref})$$

$$l{:=}v_2|\mu \to \textsf{unit}|[l \mapsto v_2]\mu \quad (\textsc{Assign})$$

$$\frac{t_1|\mu \to t_1'|\mu'}{t_1{:=}t_2|\mu \to t_1'{:=}t_2|\mu'} \quad (\textsc{Assign1})$$

$$\frac{t_2|\mu \to t_2'|\mu'}{v_1{:=}t_2|\mu \to v_1{:=}t_2'|\mu'} \quad (\textsc{Assign2})$$

Figure 3-4: Syntax and Reduction Rules for Untyped λ-calculus with Side Effects

$$
\begin{array}{lcl}
v & ::= & \dots \\
 & | & sv \\
sv & ::= & sx \mid \mathsf{Exp}\ v^* \\
\sigma & ::= & \emptyset \mid \sigma, sx = \{\phi, v\}
\end{array}
$$

$$\frac{\phi \vdash t_1|\mu,\sigma \rightarrow t_1'|\mu',\sigma'}{\phi \vdash t_1 t_2|\mu,\sigma \rightarrow t_1' t_2|\mu',\sigma'} \quad \text{(App1)}$$

$$\frac{\phi \vdash t_2|\mu,\sigma \rightarrow t_2'|\mu',\sigma'}{\phi \vdash v\ t_2|\mu,\sigma \rightarrow v\ t_2'|\mu',\sigma'} \quad \text{(App2)}$$

$$\phi \vdash (\lambda x.t)v|\mu,\sigma \rightarrow [x \mapsto v]t|\mu,\sigma \quad \text{(AppAbs)}$$

$$\phi \vdash \mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma \rightarrow t_2|\mu,\sigma \quad \text{(IfTrue)}$$

$$\phi \vdash \mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma \rightarrow t_3|\mu,\sigma \quad \text{(IfFalse)}$$

$$\frac{\phi \vdash t_1|\mu,\sigma \rightarrow t_1'|\mu',\sigma'}{\begin{array}{c}\phi \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma \rightarrow \\ \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu',\sigma'\end{array}} \quad \text{(If)}$$

$$\phi \vdash \mathsf{pred}\ 0|\mu,\sigma \rightarrow 0|\mu,\sigma \quad \text{(PredZero)}$$

$$\phi \vdash \mathsf{iszero}\ 0|\mu,\sigma \rightarrow \mathsf{true}|\mu,\sigma \quad \text{(IsZeroZero)}$$

$$\phi \vdash \mathsf{iszero}\ (\mathsf{succ}\ nv_1)|\mu,\sigma \rightarrow \mathsf{false}|\mu,\sigma \quad \text{(IsZeroSucc)}$$

$$\phi \vdash \mathsf{pred}\ (\mathsf{succ}\ nv_1)|\mu,\sigma \rightarrow nv_1|\mu,\sigma \quad \text{(PredSucc)}$$

$$\frac{\phi \vdash t_1|\mu,\sigma \rightarrow t_1'|\mu',\sigma'}{\phi \vdash \mathsf{succ}\ t_1|\mu,\sigma \rightarrow \mathsf{succ}\ t_1'|\mu',\sigma'} \quad \text{(Succ)}$$

$$\frac{\phi \vdash t_1|\mu,\sigma \rightarrow t_1'|\mu',\sigma'}{\phi \vdash \mathsf{pred}\ t_1|\mu,\sigma \rightarrow \mathsf{pred}\ t_1'|\mu',\sigma'} \quad \text{(Pred)}$$

$$\frac{l \notin dom(\mu)}{\phi \vdash \mathsf{ref}\ v_1|\mu,\sigma \rightarrow l|(\mu, l \mapsto v_1),\sigma} \quad \text{(RefV)}$$

$$\frac{\phi \vdash t_1|\mu,\sigma \rightarrow t_1'|\mu',\sigma'}{\phi \vdash \mathsf{ref}\ t_1|\mu,\sigma \rightarrow \mathsf{ref}\ t_1'|\mu',\sigma'} \quad \text{(Ref)}$$

Figure 3-5: Syntax and Reduction Rules for Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Side Effects (Part 1 of 2)

$$\frac{\mu(l) = v}{\phi \vdash\ !l|\mu, \sigma \rightarrow v|\mu, \sigma} \quad (\textsc{DerefLoc})$$

$$\frac{\phi \vdash t_1|\mu, \sigma \rightarrow t_1'|\mu', \sigma'}{\phi \vdash\ !t_1|\mu, \sigma \rightarrow\ !t_1'|\mu', \sigma'} \quad (\textsc{Deref})$$

$$\frac{\phi \vdash t_1|\mu, \sigma \rightarrow t_1'|\mu', \sigma'}{\phi \vdash t_1 {:=} t_2|\mu, \sigma \rightarrow t_1' {:=} t_2|\mu', \sigma'} \quad (\textsc{Assign1})$$

$$\frac{\phi \vdash t_2|\mu, \sigma \rightarrow t_2'|\mu', \sigma'}{\phi \vdash v_1 {:=} t_2|\mu, \sigma \rightarrow v_1 {:=} t_2'|\mu', \sigma'} \quad (\textsc{Assign2})$$

$$\frac{\mu(l) \notin sx}{\mathsf{true} \vdash l {:=} v_2|\mu, \sigma \rightarrow \mathsf{unit}|[l \mapsto v_2]\mu, \sigma} \quad (\textsc{Assign})$$

$$\frac{\mu(l) \notin sx \qquad \phi \neq \mathsf{true} \qquad sx_1 \notin dom(\sigma)}{\begin{array}{c}\phi \vdash l {:=} v_2|\mu, \sigma \rightarrow \\ \mathsf{unit}|[l \mapsto sx_1]\mu, (\sigma, sx_1 \mapsto \{(\phi, v_2), (\neg\phi, \mu(l))\})\end{array}} \quad (\textsc{AssignSymb1})$$

$$\frac{\mu(l) = sx_1}{\begin{array}{c}\phi \vdash l {:=} v_2|\mu, \sigma \rightarrow \mathsf{unit}|\mu, \sigma' \\ \text{where } \sigma' = [sx_1 \mapsto \{(\phi \wedge \phi', v_2)|(\phi', v) \in \sigma(sx_1)\} \cup \\ \{(\neg\phi \wedge \phi', v)|(\phi', v) \in \sigma(sx_1)\}]\sigma\end{array}} \quad (\textsc{AssignSymb2})$$

$$\phi \vdash \mathsf{pred}\ sv_1|\mu, \sigma \rightarrow \mathsf{Exp}(\mathsf{pred}, sv_1)|\mu, \sigma \quad (\textsc{PredSymb})$$

$$\phi \vdash \mathsf{succ}\ sv_1|\mu, \sigma \rightarrow \mathsf{Exp}(\mathsf{succ}, sv_1)|\mu, \sigma \quad (\textsc{SuccSymb})$$

$$\phi \vdash \mathsf{iszero}\ sv_1|\mu, \sigma \rightarrow \mathsf{Exp}(\mathsf{iszero}, sv_1)|\mu, \sigma \quad (\textsc{IsZeroSymb})$$

$$\frac{\phi \wedge sv_1 \vdash t_2|\mu, \sigma \rightarrow t_2'|\mu', \sigma'}{\begin{array}{c}\phi \vdash \mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu, \sigma \rightarrow \\ \mathsf{if}\ sv_1\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3|\mu', \sigma'\end{array}} \quad (\textsc{IfSymb1})$$

$$\frac{\phi \wedge \neg sv_1 \vdash t_3|\mu, \sigma \rightarrow t_3'|\mu', \sigma'}{\begin{array}{c}\phi \vdash \mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu, \sigma \rightarrow \\ \mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3'|\mu', \sigma'\end{array}} \quad (\textsc{IfSymb2})$$

$$\begin{array}{c}\phi \vdash \mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ v_3|\mu, \sigma \rightarrow \\ \mathsf{Exp}(\mathsf{if}, sv_1, v_2, v_3)|\mu, \sigma\end{array} \quad (\textsc{IfSymb})$$

Figure 3-6: Reduction Rules for Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Side Effects (Part 2 of 2)

$$
\begin{array}{rcl}
v & ::= & \dots \\
  & | & sv \mid \mu \\
  &   & \\
sv & ::= & sx \mid \mathsf{Exp}\ v^*
\end{array}
$$

```
:=      =  λ a, b.
             if sym? !a then
               a:=sym({(!pc ∧ pc', b)|(pc',v) ∈ !a} ∪
                         {(¬!pc ∧ pc', v)|(pc',v) ∈ !a})
             else if !pc != true then
               a:=sym({(pc, b), (¬pc, !a)})
             else a := b

if      =  λ c, t, e.
             if sym? c then
                let old_pc = !pc in
                  pc := old_pc ∧ c;
                  let v1 = t.call in
                  pc := old_pc ∧ ¬c;
                  let v2 = e.call in
                  pc := old_pc;
                  Exp(if, c, v1, v2)
             else if c then t.call else e.call

sym     =  λ vals. (SYM_TAG, vals)
sym?    =  λ v. v = (SYM_TAG, vals) or sym? v

pred    =  λ v. if sym? v then Exp(pred, v) else pred v
succ    =  λ v. if sym? v then Exp(succ, v) else succ v
iszero  =  λ v. if sym? v then Exp(iszero, v) else iszero v
```

Figure 3-7: Implementation of Primitives to Achieve Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Side Effects Under Standard Semantics

3. We show that when the path constraint is *not* "true," updating a concrete variable causes the same update in the symbolic state in both the symbolic semantics and our approach—in other words, the variable is concrete, but its value depends on a symbolic value.
4. We show that updating a symbolic variable causes the same update in the symbolic state in both the symbolic semantics and our approach—in this case, the variable itself is symbolic, so the symbolic state is updated to store the new value for the current path condition and the old value with its negation.

Proof By induction on t, considering the "if" and assignment cases involving symbolic values.

Case 1 $t = \mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu, \sigma$

By IFSYMB1, IFSYMB2, and IFSYMB, if:

$$\phi \wedge sv_1 \vdash t_2|\mu, \sigma \xrightarrow{*} v_2|\mu', \sigma'$$

$$\phi \wedge \neg sv_1 \vdash t_3|\mu', \sigma' \xrightarrow{*} v_3|\mu'', \sigma''$$

Then we have that:

$$\phi \vdash t|\mu, \sigma \xrightarrow{*} \mathsf{Exp}(sv_1, v_2, v_3)|\mu'', \sigma''$$

By $\rightarrow_\theta$, we have that:

$t|(\gamma(\mu, \sigma), \mathsf{pc} \mapsto \phi)$
$\xrightarrow{*}_\theta$ $\mathsf{if\ sym?}\ sv_1\ \mathsf{then}\ ...\ \mathsf{else}\ ...|(\gamma(\mu, \sigma), \mathsf{pc} \mapsto \phi)$
$\xrightarrow{*}_\theta$ $\mathsf{pc} := \phi \wedge sv_1;\ ...|(\gamma(\mu, \sigma), \mathsf{pc} \mapsto \phi)$
$\xrightarrow{*}_\theta$ $\mathsf{let}\ \mathsf{v}_1 = t_2.\mathsf{call\ in}\ ...|(\gamma(\mu, \sigma), \mathsf{pc} \mapsto \phi \wedge sv_1)$
$\xrightarrow{*}_\theta$ $\mathsf{let}\ \mathsf{v}_1 = v_2\ \mathsf{in}\ ...|(\gamma(\mu', \sigma'), \mathsf{pc} \mapsto \phi \wedge sv_1)$
(by inductive hypothesis)
$\xrightarrow{*}_\theta$ $\mathsf{pc} := \phi \wedge \neg sv_1;\ ...|(\gamma(\mu', \sigma'), \mathsf{pc} \mapsto \phi \wedge sv_1)$
$\xrightarrow{*}_\theta$ $\mathsf{let}\ \mathsf{v}_2 = t_3.\mathsf{call\ in}\ ...|(\gamma(\mu', \sigma'), \mathsf{pc} \mapsto \phi \wedge \neg sv_1)$
$\xrightarrow{*}_\theta$ $\mathsf{let}\ \mathsf{v}_2 = v_3\ \mathsf{in}\ ...|(\gamma(\mu'', \sigma''), \mathsf{pc} \mapsto \phi \wedge \neg sv_1)$
(by inductive hypothesis)
$\xrightarrow{*}_\theta$ $\mathsf{pc} := \phi;\ ...|(\gamma(\mu'', \sigma''), \mathsf{pc} \mapsto \phi \wedge \neg sv_1)$
$\xrightarrow{*}_\theta$ $\mathsf{Exp}(sv_1,\ v_2,\ v_3)|(\gamma(\mu'', \sigma''), \mathsf{pc} \mapsto \phi)$

Case 2 $t = l{:=}v_2|\mu, \sigma$ *where* $\mu(l) \notin sx$ *and* $\phi = \mathsf{true}$.

By ASSIGN, we have:

$$\mathsf{true} \vdash t|\mu, \sigma \xrightarrow{*} \mathsf{unit}|[l \mapsto v_2]\mu, \sigma$$

By $\rightarrow_\theta$:

$$
\begin{array}{ll}
& t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\stackrel{*}{\rightarrow}_\theta & l := v_2|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\stackrel{*}{\rightarrow}_\theta & \mathsf{unit}|[l \mapsto v_2](\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
= & \mathsf{unit}|(\gamma([l \mapsto v_2]\mu,\sigma), \mathsf{pc} \mapsto \phi)
\end{array}
$$

Case 3 $t = l{:=}v_2|\mu,\sigma$ *where* $\mu(l) \notin sx$ *and* $\phi \neq \mathsf{true}$.

By AssignSymb1, we have:

$$\phi \vdash t|\mu,\sigma \stackrel{*}{\rightarrow} \mathsf{unit}|[l \mapsto sx_1]\mu, (\sigma, sx_1 \mapsto \{(\phi, v_2), (\neg\phi, \mu(l))\})$$

By $\rightarrow_\theta$:

$$
\begin{array}{ll}
& t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\stackrel{*}{\rightarrow}_\theta & l := \mathsf{sym}(\{(\phi,\ v_2),\ (\neg\phi,\ !l)\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\stackrel{*}{\rightarrow}_\theta & l := \mathsf{sym}(\{(\phi,\ v_2),\ (\neg\phi,\ \mu(l))\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
& \text{(because } \mu(l) \notin sx\text{)} \\
\stackrel{*}{\rightarrow}_\theta & l := (\mathsf{SYM_TAG},\ \{(\phi,\ v_2),\ (\neg\phi,\ \mu(l))\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\stackrel{*}{\rightarrow}_\theta & \mathsf{unit}|[l \mapsto (\mathsf{SYM_TAG}, \{(\phi, v_2), (\neg\phi, \mu(l))\})](\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
= & \mathsf{unit}|(\gamma([l \mapsto sx_1]\mu, (\sigma, sx_1 \mapsto \{(\phi, v_2), (\neg\phi, \mu(l))\})), \mathsf{pc} \mapsto \phi) \\
& \text{(by definition of } \gamma\text{)}
\end{array}
$$

Case 4 $t = l{:=}v_2|\mu,\sigma$ *where* $\mu(l) = sx_1$.

By AssignSymb2, we have:

$$
\begin{array}{r}
\phi \vdash t|\mu,\sigma \stackrel{*}{\rightarrow} \mathsf{unit}|\mu,\sigma' \\
\text{where } \sigma' = [sx_1 \mapsto \{(\phi \wedge \phi', v_2)|(\phi', v) \in \sigma(sx_1)\} \cup \\
\{(\neg\phi \wedge \phi', v)|(\phi', v) \in \sigma(sx_1)\}]\sigma
\end{array}
$$

By $\rightarrow_\theta$:

$$
\begin{array}{ll}
& t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\stackrel{*}{\rightarrow}_\theta & l{:=}\mathsf{sym}(\{(\phi \wedge \mathsf{pc'},\ v_2)|(\mathsf{pc'},\mathsf{v}) \in\ !l\} \cup \\
& \qquad \{(\neg\phi \wedge \mathsf{pc'},\ \mathsf{v})|(\mathsf{pc'},\mathsf{v}) \in\ !l\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\stackrel{*}{\rightarrow}_\theta & l{:=}\mathsf{sym}(\{(\phi \wedge \mathsf{pc'},\ v_2)|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\} \cup \\
& \qquad \{(\neg\phi \wedge \mathsf{pc'},\ \mathsf{v})|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
& \text{(since } \mu(l) = sx_1,\ \gamma(\mu,\sigma)(l) = \sigma(sx_1)\text{)} \\
\stackrel{*}{\rightarrow}_\theta & l{:=}(\mathsf{SYM_TAG},\ \{(\phi \wedge \mathsf{pc'},\ v_2)|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\} \cup \\
& \qquad \{(\neg\phi \wedge \mathsf{pc'},\ \mathsf{v})|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\stackrel{*}{\rightarrow}_\theta & \mathsf{unit}|[l \mapsto (\mathsf{SYM_TAG}, S)](\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
& \text{where } S = \{(\phi \wedge \text{pc'},\ v_2)|(\text{pc'},\text{v}) \in \sigma(sx_1)\} \cup \\
& \qquad \{(\neg\phi \wedge \mathsf{pc'},\ \mathsf{v})|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\}) \\
= & \mathsf{unit}|(\gamma(\mu,\sigma'), \mathsf{pc} \mapsto \phi) \\
& \text{(by definition of } \gamma\text{)}
\end{array}
$$

The remaining cases are straightforward. ∎

3.3 Related Work

Research on symbolic execution has a long history, with the first systems due to King [33] and to Clarke [20], both in 1976. Interest in symbolic execution has continued, and new developments have greatly increased the scalability of symbolic execution engines [11, 31, 39, 45, 44, 27].

Two notable examples of modern symbolic execution systems are the symbolic extension of Java PathFinder [31, 39], which has been used to analyze Java code used by NASA, and CUTE [45], a "concolic" testing tool for C that interleaves invocations of a symbolic and concrete execution.

The recent popularity of dynamic languages has lead to a corresponding interest in symbolic execution for these languages. Saxena et. al [44] perform symbolic execution on Javascript programs, for example, to discover malware; Rozzle [22] is a similar effort that uses symbolic execution along with other techniques to detect malicious Javascript. Apollo [4] is a symbolic evaluator for PHP, and finds run-time errors, while Ardilla [32] uses this evaluator to additionally detect SQL injection and cross-site scripting attacks. CutiePy [43] is a standalone concolic evaluator for Python, but it has not been applied to web applications. NICE-PySE [12] and Commuter [21] implement symbolic execution as a library, as we do, to analyze programs written in two domain-specific languages embedded in Python. Unlike our approach, these tools do not attempt to cover the entire host language, and do not apply to web applications.

Chef [9] produces symbolic evaluators for interpreted languages by symbolically executing *the standard interpreter itself* on the target program, allowing a single native-code symbolic evaluator to be converted into a symbolic evaluator with minimal effort. Like our approach, Chef results in a system that is 100% compatible with the standard interpreter; it improves on our approach by directly executing calls to native code, too (our approach requires specifying these methods). On the other hand, the indirection of symbolically executing the interpreter incurs significant overhead (at least 5x over NICE-PySE [12], which is implemented in the same way as our approach) even when all program inputs are concrete—whereas our approach allows the interpreter to run at full speed on concrete inputs. Despite its lower performance, Chef may indeed produce a symbolic evaluator fast enough to analyze web applications; had it been available when we developed our symbolic evaluator, we likely would have tested it before investing the time to build our own system.

As in our approach, Yang et al. [53] and Köskal et al. [34] both embed symbolic values in the host language—in this case, Scala—to enforce security policies and perform constraint programming, respectively. Both require that symbolic values interact only with a short list of "symbolic" library functions, however, and do not allow symbolic values to flow through arbitrary program code. Rosette [51] also uses our approach, but in the context of Racket, to perform both verification and program synthesis; Rosette does not allow symbolic execution of arbitrary Racket programs.

Austin et. al [5] propose *virtual values*, and allow the programmer to provide definitions for primitive operations over these values. Such a mechanism provides the perfect platform on which to build library-based alternative execution models like our approach to symbolic execution, but has not yet been applied to real-world programming languages.

Chapter 4

Implementation: A Symbolic Evaluator for Ruby on Rails

In this chapter, we demonstrate the implementation of a symbolic evaluator for Ruby on Rails programs as a library. This library transforms the standard Ruby interpreter into a symbolic evaluator, capable of computing both with concrete values and with symbolic objects. Ruby's flexibility aids us in this task: its metaprogramming features allow most of the evaluator to be implemented using standard Ruby features, with only a small amount of code rewriting required.

Implementing symbolic execution as a library means that concrete parts of the target program execute at full speed, just as they would during concrete execution. As a result, even large programs become amenable to symbolic execution if the number of symbolic inputs is small. At the same time, the library-based approach eliminates much of the burden of building a specialized symbolic execution engine, since a small number of primitive definitions often suffice to extend symbolic execution to the entire language.

4.1 A Simple Symbolic Evaluator

To explain how our implementation performs the analysis described by our formal model, we begin by illustrating the basics of symbolic execution as a library. We construct a simple symbolic evaluator for side-effect free programs. We first introduce a class to represent symbolic values, and a descendant of that class to represent symbolic expressions:

```
class SymbolicObject
  def method_missing(meth, *args)
    Exp.new(meth, [self] + args)
  end

  def ==(other)
    Exp.new(:equals, [self, other])
  end
```

```
end

class Exp < SymbolicObject
  def initialize(rator, rands)
    @rator = rator
    @rands = rands
  end
end
```

An instance of "SymbolicObject" represents a symbolic variable. The class defines the "method_missing" method so that an arbitrary method invocation on a symbolic object yields a symbolic expression representing that invocation. For example, the following program produces a symbolic expression:

```
x = SymbolicObject.new
y = SymbolicObject.new
x.foo(y)

 ⇒

Exp(foo, [x, y])
```

Ruby only invokes the "method_missing" method if the receiver is missing the called method; since it is defined on the Object class, every object has the "==" method. We therefore must redefine "==" specifically:

```
x = SymbolicObject.new
y = SymbolicObject.new
x + y == y + x

 ⇒

Exp(==, [Exp(+, [x, y]), Exp(+, [y, x])])
```

4.2 Conditionals

Handling conditionals is a key part of symbolic execution, since the system must execute both branches of conditional that depends on a symbolic value. In the ideal implementation of Ruby, we could write the following definition of "**if**" as a call-by-name function:

```
def if(condition, then_do, else_do)
  c = condition.call
  if c.is_a? SymbolicObject then
    Exp.new(:if, [c, then_do.call, else_do.call])
  else
    if c then then_do.call else else_do.call end
  end
end
```

This definition would enable the user to write code with conditionals and get symbolic results:

```
x = SymbolicObject.new
if x.even? then
  (x+1).odd?
end

  =>

Exp(if, [Exp(even?, [x]), Exp(odd?, [Exp(+, [x, 1])])])
```

While Ruby makes it easy to redefine most primitive operations, there are a select few that have been hard-coded. These include "**if**," "**and**," "**or**," "**not**," "**while**," and "**until**." These operators cannot be redefined using Ruby alone, meaning that we cannot use our metaprogramming approach to implement symbolic versions of them.

To solve this problem, we used a Ruby library called VIRTUAL_KEYWORDS developed with the motivation of handling conditionals in Rubicon. The library allows programmers to redefine the hard-coded keywords in Ruby, passing a block representing the new definition. It works by performing code transformation: the library intercepts method calls, replaces calls to the affected keywords with calls to the redefined versions, and then executes the transformed code.

Rubicon uses VIRTUAL_KEYWORDS to redefine the hard-coded keywords as follows:

```
virtualizer = VirtualKeywords::Virtualizer.new
  :for_subclasses_of => [ActionController::Base,
                         RSpec::Core::ExampleGroup]

virtualizer.virtual_if do |condition, then_do, else_do|
  c = condition.call
  if c.is_a? SymbolicObject then
    Exp.new(:if, [c, then_do.call, else_do.call])
  else
    if c then then_do.call else else_do.call end
  end
end

virtualizer.virtual_and do |a, b|
  a = a.call
  if a.is_a? SymbolicObject then
    Exp.new(:and, [a, b.call])
  elsif not a then
    a
  else
    b = b.call
    if b.is_a? SymbolicObject then
      Exp.new(:and, [a, b])
    else
```

```
      b
    end
  end
end

 ...
```

Lines 1-3 construct a virtualizer object, which is the interface through which keywords are redefined; the restriction in lines 2-3 means that the redefinitions will take effect only in Rails controllers (which hold the implementations of Rails applications, and which are descendents of ActionController::Base) and in Rubicon specifications (which are descendents of RSpec::Core::ExampleGroup).

Lines 5-12 redefine "**if**." The redefined version first invokes the block containing the condition; if the condition turns out to be symbolic, then the redefinition produces a symbolic conditional based on invoking both branches (line 8). If the condition is concrete, the redefinition falls back to the standard definition. This version does not handle side effects, but we will address them in the next section.

Lines 14-28 redefine "**and**." The redefinition executes the first conjunct first (line 15); if it is symbolic, then the redefinition returns a symbolic expression containing both that value and the value of the other conjunct (lines 16-17). If the first conjunct is false, the redefinition returns **false** (lines 18-19), preserving Ruby's short-circuiting behavior for "**and**." Next, the redefinition executes the second conjunct; if it is symbolic, then a symbolic expression containing both conjuncts is returned (lines 21-23). Otherwise, the redefinition returns the value of the second conjunct.

4.3 Side Effects

Supporting side effects in the presence of symbolic values requires a significant change to the way conditionals are handled. Since both branches of the conditional may contain updates to the same variable, it becomes necessary to save both values, along with the *path condition* under which the variable takes a particular value.

We begin by adding a representation of symbolic state, which we store in symbolic objects themselves. We add a method to symbolic objects that adds a new possible value, along with the associated path condition, to the symbolic state:

```
class SymbolicObject
  def initialize
    @vals = []
  end

  def add_val(cond, val)
    @vals << [cond,val]
  end
end
```

To handle side effects properly, the new definition of "**if**" must save the current

```
def get_state(binding)
  Hash[eval("local_variables", binding).
       map{|var| [var, eval(var, binding)]}]
end

def save_state(binding)
  state = get_state(binding)

  state.each_pair do |var, val|
    eval(var + "_old = " + var, binding)
  end

  get_state(binding)
end


def update_state(state, state1, binding)
  state1.each_pair do |var, val|
    if val.equal? state[var] then
      # no change
    else
      if state[var].is_a? SymbolicObject then
        eval(var + " = " + var + "_old", binding)
        state[var].add_val($path_condition, val)
      else
        eval(var + " = SymbolicObject.new", binding)
        new_obj = eval(var, binding)
        new_obj.add_val(true, state[var]) if state[var]
        new_obj.add_val($path_condition, val)
      end
    end
  end
end
```

Figure 4-1: Helpers for "**if**" to Handle Side Effects

```
def if(condition, then_do, else_do)
  c = condition.call
  if c.is_a? SymbolicObject then
    pc = $path_condition

    state = save_state(then_do.binding)
    $path_condition = Exp.new(:and, [c, pc])
    v1 = then_do.call
    state1 = get_state(then_do.binding)
    update_state(state, state1, then_do.binding)

    state = save_state(else_do.binding)
    $path_condition = Exp.new(:and, [Exp.new(:not, [c]), pc])
    v2 = else_do.call
    state2 = get_state(else_do.binding)
    update_state(state, state2, else_do.binding)

    $path_condition = pc
    Exp.new(:if, [c, v1, v2])
  else
    if c then then_do.call else else_do.call end
  end
end
```

Figure 4-2: Redefinition of "**if**" to Handle Side Effects

state, execute the conditional's first branch, update the symbolic state based on the updates made during that execution, and repeat the process for the second branch. In addition, the path condition must be set appropriately for the execution of each branch, and used in updating the symbolic state.

Figures 4-1 and 4-2 contains a definition of "**if**" that handles side effects. It works by saving the current state, updating the path condition based on the branch being executed, executing the branch, and updating the symbolic state based on the updates to the concrete state. Because it is impossible to set the value of a variable stored in a Ruby Binding object directly, the "save_state" procedure saves a copy of each variable with the suffix "_old" and the "update_state" procedure uses these copies to retrieve previous values of updated variables.

The procedure "get_state" (lines 1-4) is equivalent to the "get_state" operation from Section 4, and "update_state" (lines 17-33) corresponds to the "update_symb" of Section 4. The "save_state" procedure (lines 6-14) combines "get_state" with an implementation trick to save the old values of all variables so that they can be recovered after a branch is executed. Lines 35-57 are the definition of "**if**."

The redefinition first executes the condition (line 36) to determine whether or not it is symbolic. If not, the redefinition falls back to the standard definition (line 55). If the condition is symbolic, the redefinition gets the current path condition (line 38) and saves the current state (line 39). The "save_state" procedure (lines 6-14) gets the current state (line 7), saves a copy of the value of each variable in a new variable with the suffix "-old," (lines 9-11) and gets the current state again (line 13), so that the returned version of the state contains the saved "-old" values. This implementation trick is required because Ruby does not allow the value of variables in an environment (in Ruby, an instance of the class "Binding") to be updated directly. Instead, we update these variables using "eval," as in line 10.

Returning to the redefinition of "**if**," lines 41-44 updates the path condition, executes the first branch of the conditional, gets the updated state, and calls "update_state." That procedure, which corresponds to "update_symb" from Section 4, examines each member of the state (line 18). If there has been no change, the procedure does nothing (lines 19-20). If the variable's value has changed, and the variable was previously symbolic (line 22), then the procedure resets the variable's value to the previous symbolic object (line 23) and adds the new value to the symbolic state of that object. If the variable's value was not previously symbolic, the procedure constructs a new symbolic object (line 26) and adds both the previous value of the variable, if it exists, (line 28) and the new value (line 29) to the symbolic state of the new object.

The other branch of the conditional is treated the same way (lines 46-50), but with the path condition modified to contain the negation of the condition. Finally, the path condition is reset (line 52) and a symbolic expression containing the resulting values is returned (line 53).

This redefinition allows the programmer to write code like the following:

```
x = SymbolicObject.new
y = SymbolicObject.new
if x.even? then
```

```
  y = true
else
  y = false
end
```

Executing this program gives the variable "y" the following symbolic state:

```
Exp(even?, [x]) => true,
Exp(not, [Exp(even?, [x])]) => false
```

This symbolic state represents the two possibilities for "y:" if "x" is even, then the value of "y" will be **true**; otherwise, it will be **false**.

4.4 Stubbing Rails

Rails web applications interact with a persistent database through ActiveRecord, an object-relational mapper. In general, the properties of Rails applications that Rubicon checks should be true for all configurations of the database. As a result, Rubicon must treat the database as symbolic data when checking properties.

Fortunately, Rails enforces the use of the ActiveRecord interface for accessing the database, so we can simply provide a new implementation of that interface which returns symbolic values instead of actual database records. In a Rails application, objects to be stored in the database extend ActiveRecord, and the ActiveRecord class provides methods such as "find" and " all" to query the database for records representing objects of the receiver's type. For example, given the following User class:

```
class User < ActiveRecord::Base
end
```

A Rails application could find all users with the name "Joe" using the following expression, which evaluates to a list of records with the given property:

```
User.find :name => "Joe"
```

Our goal is for expressions like these to evaluate to symbolic expressions representing the database query itself. The obvious way to accomplish this is to use Ruby's open classes to redefine "find" and the other querying methods of ActiveRecord. Unfortunately, Rails prevents this approach by defining the methods on ActiveRecord objects dynamically. At runtime, then, our redefinitions would be overwritten with the originals as defined by Rails. Rails defines methods dynamically so that each ActiveRecord object responds to a set of methods representing the fields of the corresponding database records. Given a User object "u," for example, the expression u.name evaluates to the name field of the database record corresponding to the user "u."

The solution is to employ dynamic redefinition ourselves. We redefine each instance method corresponding to a database field so as to return a symbolic expression, and we redefine the class methods for constructing database queries to return

```
class TypedSymbolicObject < SymbolicObject
  def initialize(type)
    @type = type
  end
end

klasses = ActiveRecord::Base.descendants

klasses.each do |klass|
  metaklass = class << klass; self; end
  metaklass.send(:define_method, :new, lambda
     {TypedSymbolicObject.new(self)})
  metaklass.send(:define_method, :my, lambda
     {TypedSymbolicObject.new(self)})
  metaklass.send(:define_method, :all, lambda
     {TypedSymbolicObject.new(self)})
  ...

  klass.column_names.each do |name|
    klass.send(:define_method, name.to_sym, lambda {
       Exp.new(:field_get, [self, name.to_sym]) })
    klass.send(:define_method, (name + "=").to_sym, lambda
       {|arg| Exp.new(:field_set, [self, name.to_sym, arg]) })
  end

  klass.reflect_on_all_associations.each do |assoc|
    klass.send(:define_method, assoc.name, lambda {
       Exp.new(:field_get, [self, assoc.name]) })
  end
end
```

Figure 4-3: Code to Stub Rails Database Accessor Methods

symbolic queries.

Figure 4-3 contains the code used to perform this step, along with a definition of typed symbolic objects. The code works by redefining both the class methods and instance methods of ActiveRecord's descendents. Lines 10-13 show how some of the database query methods are redefined. Lines 16-19 redefine the getters and setters for the database field methods of the class, and lines 21-23 do the same for the *associations*—relationships with other objects through a separate database table—belonging to the class.

With this redefinition, we can symbolically execute code like the following definition of the "show" method of the "User" controller of Fat Free CRM, which method starts by fetching the user associated with the provided ID:

```
class UsersController < ApplicationController
  def show
    @user = User.my.find(params[:id])
    ...
  end
end
```

Given a symbolic ID, the "@user" variable will get the value:

```
Exp(:find, [Exp(:query, [User]), SymbolicObject1])
```

4.5 Extracting Exposures

4.5.1 Rendering

A Rails action serves a request in two steps: first, the code defined in the controller populates a set of instance variables; then, Rails evaluates an appropriate *template*—which may reference those instance variables—to produce an HTML string. We wrap the Rails renderer to extract the set of symbolic values that appear on each rendered page. These values, along with the constraints attached to them, contribute to the set of exposures resulting from the action being executed.

4.5.2 Normalization

Since two constraints can be logically equivalent but syntactically different, we attempt to normalize the set of constraints so that whenever possible, two logically equivalent constraints will also be syntactically equal.

We uses two basic methods to accomplish this normalization. First, calls to the ActiveRecord API are rewritten in terms of the find method, and queries are merged when possible. For example, User.find (:name ⇒ 'Joe').filter (:role ⇒ 'admin') becomes User.find (:name ⇒ 'Joe', :role ⇒ 'admin'). Second, we convert all constraints to conjunctive normal form, eliminating issues like double negation.

4.6 Challenges

Both the dynamic nature of Ruby and the size of the Rails library pose significant challenges to standard symbolic execution strategies. These challenges motivated our unique solution.

Rails is large and complicated. The Rails framework is notoriously complicated—for many years, it was compatible only with the standard MRI Ruby interpreter due to its use of undocumented features. This provided the strongest motivation for implementing our symbolic evaluator as a Ruby library. It allows some code, such as Rails's configuration code, to be run concretely, and at full speed. In addition, using the standard Ruby interpreter gives us confidence that our symbolic evaluator is faithful to the semantics of Ruby and Rails, since it runs the actual implementations of both.

Ruby does not allow the redefinition of conditionals. Ruby provides facilities for metaprogramming, but they are limited. When a conditional expression depends on a symbolic value, symbolic execution requires that we execute both branches, but Ruby does not allow the programmer to attach special behavior to conditionals. We rewrite the application's code, transforming **if** expressions into calls to our code that runs both branches. This can result in the exponential blowup characteristic of symbolic execution, so we execute only the appropriate branch when a conditional's condition is concrete.

Rails plugins use metaprogramming. Rails plugins are extra libraries that can be included in applications to provide additional functionality. The CanCan plugin, for example, provides user authentication and access control—features not built into Rails. Unfortunately, the use of metaprogramming in plugins often conflicts with our own use of the same technique. CanCan, for example, replaces many of ActiveRecord's query methods with versions that perform security checks. Since our versions of the same methods are essentially specifications of the default Rails behavior, our replacement methods eliminate the extra security checks introduced by CanCan.

Our solution is to allow our specifications to be extended to match the functionality added by plugins. Using this technique, adding CanCan's security checks is accomplished in just a few lines of Ruby code.

Rendering makes it difficult to extract the set of symbolic values the user will actually see. Rails's rendering mechanism is complicated, so we prefer to run its implementation rather than specify its semantics manually; since the output of rendering is a string containing HTML, however, it is also difficult to reconstruct the set of symbolic values from the renderer's output.

Our solution assumes that the set of objects receiving the to_s method—which converts an object into a string—during rendering is exactly the set of objects appearing on the resulting page. This is a conservative assumption, since while a template may convert an object into a string and then discard it (a situation we have yet to encounter in practice), Rails always calls to_s on objects appearing in templates. Under this assumption, we modify the to_s method of symbolic values so that each value keeps track of whether or not it has been converted into a string, and run Rails's ren-

derer unmodified. After rendering has finished, we collect the set of symbolic values that have been converted to strings, and return them as the set of results.

4.7 Assumptions & Limitations

Our strategy for symbolic execution relies on several assumptions. While we consider these assumptions reasonable, some of them do imply corresponding limitations of our analysis.

We assume that to_s is called on exposed objects. But since the Rails rendering engine calls to_s automatically on every object that appears in a template, we consider this a reasonable assumption. An application may use a conditional to decide whether or not to display a particular string; in this case, the result could leak some information about the symbolic values present in the condition without calling to_s.

Our system does not handle symbolic string manipulation. We record manipulations performed on symbolic strings, but cannot solve them. Fortunately, Rails applications tend to perform few string manipulations, especially on objects drawn from the database, and almost never base control flow on the results of those manipulations. We have therefore not found the inability to solve string manipulation constraints to be a problem in practice.

We assume that all actions are reachable. Rails uses *routes* to define the mapping between URLs and actions. We ignore routes and simply analyze all actions defined by the application. This strategy is an over-approximation: it is possible for an action to exist without a corresponding route, making the action dead code. But the converse is *not* true: it is impossible to miss an exposure by ignoring some code that is actually live—we simply analyze *all* the code.

We assume that the application under analysis uses ActiveRecord. Our system wraps the ActiveRecord API to make database queries symbolic, but an application that uses a different method to make database queries may bypass this wrapping. The application may therefore have access to *concrete* database data during analysis, meaning the results will not generalize to all database values. But since our analyses are intended for use by an application's developer, we assume that he or she will know whether or not the application uses a database API other than ActiveRecord—and if so, we provide a simple mechanism to specify that API.

4.8 Soundness & Optimizations

To guarantee soundness in a dynamic language such as Ruby is a difficult task. Because Ruby's modularity constructs are permeable, a module's semantics depends on the environment in which it executes. It is therefore impossible to provide a modular analysis for Ruby that is sound, since loading the module in a different context could change its semantics. Our solution is to instruct users to perform the analysis in the environment they will use in production; in practice, this is sufficient to produce correct results.

Program Element	Optimization
Functional maps	Run map body once with symbolic representation of array element (sound).
For-each loops with side effects	Special path constraint for handling side effects within loops (**unsound**).
Other loops	Unroll loop a finite number of times (**unsound**).
Array & hash operations	Replace array and hash methods for symbolic manipulation of arrays (**unsound**).
Database queries	Replace query methods to return symbolic representations of query (sound).
Model class calls	Except for user-defined methods, replace *all* methods on model classes with symbolic versions (sound).

Figure 4-4: Summary of Optimizations and their Effect on Soundness

We also use a number of optimizations that, while unsound in theory, produce correct results in practice. In addition to aiding in performance of the symbolic analysis, these optimizations can actually produce more precise symbolic results. Chief among these is the ability to treat certain kinds of loops as functional maps, allowing their behavior to be characterized precisely using only a single execution of the loop body. In addition, all of our loop-handling strategies are guaranteed to terminate, meaning that our analysis also always terminates. These optimizations are summarized in Figure 4-4 and detailed below.

The general strategy is to attempt to convert looping constructs into set comprehensions (eliminating side effects). For functional maps, this is easy, since they already contain no side effects. For loops that do perform side effects, we construct a special symbolic expression representing a universally quantified index into the target collection and run the loop body on that expression; we then construct a functional map by examining the results of the side effects performed while running the loop to find the quantified expression we constructed. This result is then converted to a set comprehension.

Functional maps. Instead of side-effecting loops, many Rails applications make extensive use of functional maps. These can be characterized precisely by running the mapped function on a single symbolic value, and the resulting symbolic value can be represented as a set comprehension. For example, this code from Diaspora finds a post, accesses the list of "likes" attached to the post, and then derives a list of the people who did the "liking" using a functional map. In our system, it evaluates as follows:

```
@likes = Post.find(params[:post_id]).likes
@people = @likes.map(&:author)
```

```
⇒
Exp(:map, Exp(Post, Exp(Post, :find, :params[:post_id]),
    :likes), :author)
```

The resulting symbolic expression simply records the fact that a map is performed on the collection. For tools that perform constraint solving, this expression can be transformed into a set comprehension that applies the appropriate field access to each element of the collection:

```
{ u: User | some p: Post | p.id = post_id and u in
    p.likes.author }
```

Loops with side effects. Some for-each loops in Rails applications perform side effects on objects constructed outside the loop. Running the loop body just once does not adequately capture the behavior of such a loop, which is commonly used to fill up an array for later use. The following code, for example, constructs a hash of conversation authors using a side effect:

```
@authors = []
@conversations = current_user.conversations
@conversations.each { |c| @authors << c.last_author }
```

Such loops are equivalent to a functional map with a filter. Our system detects this condition and transforms the resulting symbolic expression into the equivalent functional map in two steps. First, the loop body is run once with a special copy of the collection being looped over tagged with the type "foreach_var," which specifies that this expression represents an arbitrary index into the collection it contains. Second, the system transforms the resulting values into functional maps by checking for the tagged value; when it is found, the transformation treats the enclosing value as if it is defined by a functional map over the original collection.

For the code above, our strategy results in setting the "@authors" variable to the following:

```
[Exp(:foreach_var, Exp(:Conversation, :current_user,
    :conversations),
:last_author)]
```

which is an array of length one, containing a "foreach_var" expression representing the invocation of the "last_author" method on the original list of conversations. This value is transformed into the following functional map:

```
Exp(:map, Exp(:Conversation, :current_user, :conversations),
    :last_author)
```

which is, in turn, equivalent to this set comprehension:

```
{ u: User | u in current_user.conversations.last_author }
```

Other loops. When a Rails programs contains a "**while**" loop, our approach simply

unrolls the loop a finite number of times (by default, 3 times). These loops are so rare in practice that this simple technique has so far produced correct results in all cases. For example, this loop in Diaspora is used to find the original "sharer" in a chain of "re-shares:"

```
@absolute_root ||= self
@absolute_root = @absolute_root.root while @absolute_root.is_a?
    Reshare
```

Unrolling this loop three times allows the system to consider re-sharing chains up to length three, which does not precisely match the semantics of the original loop, but which is sufficient to check the security properties of the application.

Array & hash operations. When array operations are performed on an array containing a "foreach_var" expression (i.e. the results of a side-effecting loop), those operations must treat the entire collection as if it is defined by the "foreach_var" expression. Consider a functional map performed on the final value of "@authors" from the code above:

```
@authors = []
@conversations = current_user.conversations
@conversations.each { |c| @authors << c.last_author }
@names = @authors.map{ |a| a.first_name }
```

When the final line of code runs (setting the "@names" variable), the "@authors" variable is bound to the value:

```
[Exp(:foreach_var, Exp(:Conversation, :current_user,
    :conversations),
:last_author)]
```

We redefine "map" to recognize this structure and produce the following value for "@names:"

```
Exp(:map,
  Exp(:map, Exp(:Conversation, :current_user, :conversations),
            :last_author),
  :first_name)
```

resulting in the following set comprehension:

```
{ s: String | s in
    current_user.conversations.last_author.first_name }
```

Database queries. ActiveRecord is designed so that all database queries are eventually recast in terms of the "where" and "update_attributes" methods, which generate SQL queries. We redefine these to return symbolic database queries:

```
class ActiveRecord::Base
  def where(query)
    Exp.new(class_of(self), :query, query)
```

```
  end

  def update_attributes(tuple)
    set_updated(self, tuple)
    true
  end
end
```

The new definition of "where" returns an expression representing the results of the query in a symbolic database; the new definition of "update_attributes" records the specified changes and returns true. Since the other ActiveRecord methods use these two as their interface, we execute them directly until one of the two is called.

Model class calls. ActiveRecord defines a huge number of methods for model classes, many of which are seldom-used but large. We redefine all of these to return symbolic expressions recording simply that they were called, by redefining each one to call "base_method" instead:

```
class ActiveRecord::Base
  def base_method(method_name, *args)
    Exp.new(class_of(self), method_name, args)
  end
end
```

This is a conservative approximation of the behavior of the methods we replace. The symbolic expression returned represents the results of a particular method call. Thus, if two pieces of code call the same method, they will produce syntactically identical results, and be comparable. Our experience has been that these methods perform tasks like time zone localization, and therefore calls must be identical for most properties to hold anyway.

This modification is *unsound* if the replaced methods might perform side effects. We manually checked the ActiveRecord methods we replace this way to make sure that none of them do perform side effects.

4.9 Evaluation

The goals for our symbolic evaluator were **compatibility**—the ability to symbolically execute a wide variety of web applications—and **scalability**—the ability to symbolically execute those applications quickly.

4.9.1 Experimental Setup

We tested our symbolic evaluator on the 1000 most-popular open-source Ruby on Rails applications hosted by Github. We approximated popularity based on the number of "stars" a project has. These applications fall into two basic categories: mature, stable applications intended for installation by end-users (which are popular

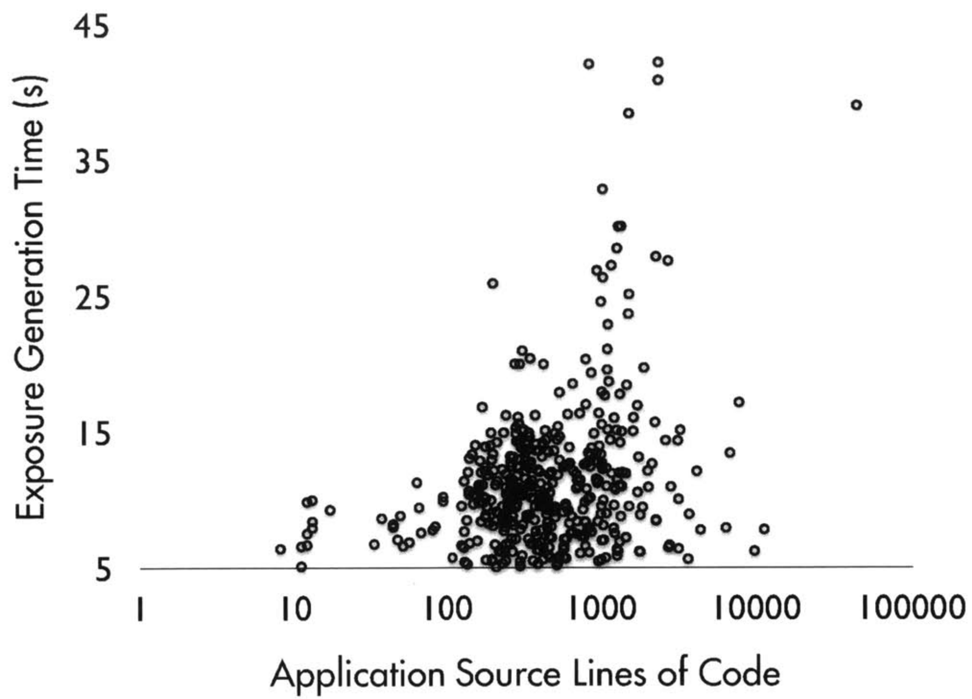


Figure 4-5: Scalability of Exposure Generation: Generation time vs application lines of code for 1000 applications

because of their user base), and fragments of example code intended for helping programmers to build new applications (which are popular because they are copied and pasted into other applications). Security bugs are a serious issue in both types of application: in mature applications, they make user data vulnerable, while in example code, they can propagate to other applications that have copied the code.

We used Ubuntu GNU/Linux on a virtual machine equipped with a single Intel Xeon core at 2.5GHz and 2GB of RAM. We used RVM to install the version of Ruby required by each application and Bundler to install each application's dependencies.

4.9.2 Results

The results of this experiment are presented in Figure 4-5, which plots each application's symbolic execution time against the number of lines of source code it contains. Most importantly, symbolic execution finished within one minute for every application, with most taking much less time.

These results indicate that our symbolic execution framework scales to large, real-world Rails applications. The largest of these applications is Diaspora, with over 40k lines of code—our evaluator finishes in just 45 seconds.

Many of the applications we tested were much smaller than Disapora. Indeed, the Rails framework is designed to encourage concise implementations, and the Rails community values small codebases. Our evaluator analyzes these small applications especially fast—normally finishing within 20 seconds.

Moreover, the number of lines of code we report in Figure 4-5 may be an over-estimate of the number of lines actually executed during analysis. The number of reported lines corresponds to the total number of lines of Ruby code present in the project (i.e. the number of lines reported on Github). Since not all of this code may be reachable during our analysis (for example, test suite code), the actual number of lines executed during our analysis may be much smaller. We discuss one example of this phenomenon in the next section.

We examined the error output of our symbolic evaluator, and found only a handful of cases in which the symbolic execution process did not finish normally. These were situations in which the application used features like TCP socket and email APIs—features our system does not support, and ones that generally will not affect detection of security bugs.

4.9.3 Discussion

To further explore the performance characteristics of our own symbolic execution system and of symbolic execution on web applications in general, we examined the codebase of the largest application we tested, Diaspora, in more detail. Figure 4-6 summarizes the number of lines of code of each implementation language present in each subdirectory of the codebase. Including every single file, Diaspora has more than one hundred thousand lines of code.

However, for the purposes of our analysis, the actual amount of code considered is much smaller. The YAML files in the configuration directory, for example, which

Directory	Language					Total
	Ruby	JS	HAML	YAML	SASS	
app/assets/	0	5690	0	0	5077	10767
app/controllers/	2228	0	0	0	0	2228
app/helpers/	902	0	0	0	0	902
app/models/	2981	0	0	0	0	2981
app/presenters/	526	0	0	0	0	526
app/views/	0	0	4679	0	0	4679
app/workers/	440	0	0	0	0	440
config/	848	0	0	82664	0	83512
db/	1361	0	0	0	0	1361
features/	1656	0	0	0	0	1656
lib/	3375	1828	0	0	0	5203
spec/	16608	4617	0	14	0	21225
vendor/	1824	0	0	0	0	1824
Total	32749	12135	4679	82664	5077	137304

Figure 4-6: Size of Diaspora's Codebase, by Language and Directory

Feature	Occurrences
while	2
each	54
map	52
if	745

Figure 4-7: Occurrences of Language Features in Diaspora's Codebase

comprise over 80,000 lines, are localization options for translating the site into different languages, and actually contain structured constant data rather than code (i.e. no YAML file can perform a side effect or make a database query). Our analysis runs the application's configuration file, which specifies a set of concrete localization options, and then consults the appropriate subset of these YAML files during rendering. Because the YAML files cannot do any computation, however, ignoring them entirely would be a sound optimization.

Similarly, our analysis runs only Ruby code, and does not examine the Javascript code (about 12,000 lines) delivered with Diaspora. Since this code runs on the client side, we consider it untrustworthy no matter what its semantics, and therefore consider any value passed to a Javascript function to be exposed.

We also do not consider some of the Ruby code. The test suite, for example, is more than 16,000 lines of code—nearly half the total amount of Ruby code—and is not needed for our analysis. On the other hand, the HAML files representing view templates (about 4,600 lines of code) contain Ruby expressions executed by the rendering engine, and are therefore also executed in our analysis.

The actual controller code, which defines the application's logic, is only 2000 lines of code; the model code, which defines the behavior of resources, is less than 3000 lines. Some methods in this code may call procedures defined in the helpers, presenters, and workers directories, as well as library code from the features, lib, and vendor directories, plus additional code from libraries installed via Rubygems. But as a lower bound, our analysis need consider only the roughly 5000 lines defining the application itself—a number far smaller than the total lines of code.

Diaspora's codebase also contains evidence that its code is especially suited to symbolic execution. Figure 4-7 summarizes the number of times various language features are used in Diaspora's Ruby code. Diaspora's controller code contains only two while loops, but uses more than 50 "for each" loops and more than 50 functional "maps"—precisely the kind of loops our analysis is optimized for.

Diaspora's controllers contain 745 conditionals, but these are distributed over a total of 225 actions. This means that each action contains an average of 3.3 conditionals; since our system analyzes each action independently, the number of possible paths per analysis is small. The amount of code per action is also small, averaging just 10 lines.

While Diaspora is only one application, its codebase suggests that our analysis runs quickly for three reasons. First, the actual code considered for our security analysis is smaller than the total size of the codebase. Second, the language features used in the code we analyze is conducive to symbolic execution. Third, our optimizations target precisely those language features most used in the code under analysis.

Like many of the applications we considered, Diaspora's main functionality is CRUD-y: its goals are centered around users creating content and sharing it with others. This results in a set of controller actions whose logical operations are relatively simple. The same may not be true of applications that resemble traditional desktop software, such as Microsoft Office or Google Docs, so we do not expect our analysis to perform nearly so well on these applications.

4.10 Related Work

Existing work on the application of static analysis to web applications focuses on modeling applications, and especially on building navigation models. Bordbar and Anastasakis [8], for example, model a user's interaction with a web application using UML, and perform bounded verification of properties of that interaction by translating the UML model into Alloy using UML2Alloy; other approaches ([35, 49, 41]) perform similar tasks but provide less automation. Nijjar and Bultan [38] translate Rails data models into Alloy to find inconsistencies, but do not examine controller code. Subsequent to the work presented here, Bocić and Bultan [7] used a similar technique to check Rails code.

Techniques that do not require the programmer to build a model of the application tend to focus on the elimination of a certain class of bugs, rather than on full verification. Chlipala's Ur/Web [18] statically verifies user-defined security properties of web applications, and Chaudhuri and Foster [15] verify the absence of some particular security vulnerabilities for Rails applications.

Part II

Verification

Chapter 5

Exploring Exposures with Derailer

Derailer is a tool for finding security bugs without a specification. Instead, it uses a combination of symbolic evaluation and user interaction to help the programmer discover mistakes.

This particular combination is motivated by two hypotheses. First, web applications differ from traditional programs in ways that improve the scalability of symbolic execution. In particular, web applications typically use fewer loops and simpler branching structures than traditional programs, minimizing the exponential behavior of symbolic execution. Second, security policies tend to be uniform: sensitive data is usually subject to security checks everywhere it is used, so an access that is *missing* one of those checks is likely to be a mistake.

Our approach considers web applications that accept requests and respond with sets of *resources* obtained by querying the database. Each response is characterized by the *path* through the database leading to the resource, and the control flow of the application's code imposes a set of *constraints* under which a particular resource is exposed to a client. We call the combination of a path and a set of constraints an *exposure*.

An automatic strategy for finding security bugs might enforce that all exposures with the same path also share the same set of constraints; if a security check is forgotten, a constraint will be missing. But many constraints—like those used to filter sets of results for pagination—have nothing to do with security, and would cause an automatic strategy to report many false positives.

Our approach therefore asks the user to separate constraints into those representing security checks and those that are not security-related. In making this separation, the user effectively constructs a *specification* of the desired security policy—but by selecting examples, rather than writing a specification manually. Our tool allows the user to drag-and-drop constraints to build the policy. The tool then highlights exposures missing a constraint from the security policy—precisely those that might represent security bugs.

We have built a tool implementing this approach. Called *Derailer*[1], it performs symbolic execution of a Ruby on Rails application to produce a set of exposures.

[1] available for download at http://people.csail.mit.edu/jnear/derailer

```
def index
  @notes = Note.where(user: current_user.id) +
    Note.permissions.find_by_user_id(current_user.id)

  respond_to do |format|
    format.html # index.html.erb
    format.json { render json: @notes }
  end
end

def show
  @note = Note.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.json { render json: @note }
  end
end
```

Figure 5-1: Example Controller Code from Student Project

We evaluated Derailer on five open-source Rails applications and 127 student projects. The largest of the open-source applications, Diaspora, has more than 40k lines of code, and our analysis ran in 112 seconds. The student projects were taken from an access-control assignment in a web application design course at MIT. Derailer found bugs in over half of these projects; about half of those bugs were missed during manual grading. The bugs we found supported our hypothesis: most bugs were the result of either a failure to consider alternate access paths to sensitive data, or forgotten access control checks.

5.1 Derailer: An Exposure Exploration Tool

Derailer uses an automatic static analysis to produce an interactive visual representation of the exposures produced by a Ruby on Rails web application. The tool displays the constraints associated with each exposure, and allows the user to separate the security-related ones from those unrelated to security. Then, Derailer highlights inconsistencies in the implemented security policy by displaying exposures lacking some security-related constraints.

To see how this works, consider the controller code in Figure 5-1, taken from a student project. The application's purpose is to allow users to create "note" objects and share them with other users; users should not be able to view notes whose creators have not given them permission. This code, however, has a security bug: the "index" action correctly builds a list (in line 2) of notes the current user has permission to view, but the "show" action displays a requested note without checking its permissions (line

(1) The user has expanded the **Note** node and its child node representing the note's content, since this node represents sensitive data.

(2) The user has selected the **NotesController / index** action, which can result in a note's content appearing on a page. A constraint representing the security policy on a note's content appears in the **Constraints** area.

Figure 5-2: Example Bug-finding Session using Derailer (Part 1)

(3) The user has dragged the constraint that appears into the **Filtered Constraints** area. The **NotesController / index** action disappears, because it is subject to a filtered constraint.

(4) The user has selected the remaining action, which remains visible because it is *not* subject to the filtered constraint. In fact, it is not subject to any constraints at all. This represents a security bug in the application.

Figure 5-3: Example Bug-finding Session using Derailer (Part 2)

11). The application's programmer assumed that users would follow links from the index page—which *does* correctly enforce access control—to view notes, and neglected the case in which the user requests bypasses the "index" action and requests a specific note directly using the "show" action.

Figures 5-2 and 5-3 contain a sequence of screenshots demonstrating the use of Derailer to find this security bug. In shot (1), the user has expanded the "Note" node, which represents the note resource type, and then the "{note: Note | note.id in params[id]}.content" node, which is a resource path rooted at the note type, representing a note's contents. The "User" node represents the programmer-defined user resource, while "session" and "env" are system-defined resource types containing information about the current session and configuration environment.

Then, in shot (2), the user picks the "index" action from the list of actions resulting in that exposure. When an action is selected, the set of constraints governing the release of that data by that action is displayed in the Constraints area. In this case, the displayed constraint is:

```
note.user == current_user or
    note.permissions.find_by_user_id(current_user.id)}
```

which says that the currently logged-in user must either be the creator of or have permission to view all visible notes. The user drags security-related constraints to the Filtered Constraints area, which will eventually contain the complete security policy of the application.

Dragging constraints to the Filtered Constraints area causes the nodes subject to those constraints to disappear, as in shot (3). Once the Filtered Constraints area contains all security-related constraints, remaining exposures of sensitive information represent inconsistencies in the implemented security policies, and are likely to be bugs. In shot (4), the "show" action remains, despite the filtered constraint; selecting it, the user discovers that it is not subject to any constraints at all.

We took the approach described in this section in using Derailer to analyze 127 similar student projects from the same course, and found bugs in nearly half of them. The results of that experiment are described in Section **??**.

5.2 Implementation

Derailer uses our symbolic execution framework to produce a list of data exposures from the target Rails application, then uses a web-based user interface to display a hierarchical view of those exposures and help the user filter them based on user-defined policies.

5.2.1 Exposure Generation

Normalization. Derailer's interface allows the user to explore candidate security policies through filtering, which compares constraints syntactically. Since two constraints can be logically equivalent but syntactically different, Derailer attempts to

normalize the set of constraints so that whenever possible, two logically equivalent constraints will also be syntactically equal.

Derailer uses two basic methods to accomplish its normalization. First, calls to the ActiveRecord API are rewritten in terms of the find method, and queries are merged when possible. For example, User.find (:name ⇒ 'Joe').filter (: role ⇒ 'admin') becomes User.find (:name ⇒ 'Joe', :role ⇒ 'admin'). Second, Derailer converts all constraints to conjunctive normal form, eliminating issues like double negation.

Filtering and Formatting. Next, Derailer eliminates duplicate constraints by comparing them syntactically. After normalization, this strategy tends to detect the vast majority of logically duplicate constraints.

For each exposure, Derailer builds a structure containing the exposure's path, symbolic expression, constraints, and the location in the code that produced the exposure. This structure is used to build Derailer's hierarchical exposure view. Derailer uses this list of structures to build a tree of exposures: the root of the tree is the common superclass of the exposures' data types (usually ActiveRecord); the next level of the tree contains the application-defined data types; the third level contains exposures, sorted into bins based on their symbolic expressions.

This tree structure is serialized to a JSON file for processing by Derailer's web-based front end.

5.2.2 Exposure Visualization and Exploration

Derailer's GUI. Derailer implements a web-based GUI to display the tree of exposures. The interface takes advantage of the D3 [2] Javascript library for data visualization to display the tree and allow the user to interact with it.

Derailer extends D3's standard tree layout by adding panes on the right-hand side for displaying constraints and allowing the user to specify which constraints form the security policy. When the user selects a particular exposure, the constraints associated with that exposure are displayed in the "Constraints" pane.

Derailer specifies the origins of each exposure by listing the set of actions allowing a particular exposure as children of that exposure. In some cases, two different actions allow the same exposure (according to the symbolic expression defining the data it exposes) under different conditions—so constraints are attached to these code locations, rather than to the exposure itself.

Filtering. To find bugs, the user selects an exposure and examines its constraints. The user drags those constraints that are security-related from the constraints pane to the filtered constraints pane. After every such change, Derailer performs a filtering step to determine which exposures correctly respect the new security policy and which ones do not.

Derailer's front end stores an array of constraints for each data type representing that type's security policy. When the user adds a constraint to the policy, Derailer updates this array and re-performs the filtering step. Derailer examines each exposure with the same data type as the updated constraint: if the set of constraints on the

[2] http://d3js.org/

Type of Bug	No.
No access control implemented	15
No write control implemented	10
No read control implemented	27
Security bug in read control	13
Security bug in write control	17

Figure 5-4: Types of Bugs Found During Analysis of Student Projects

exposure is a superset of the policy, then the exposure is marked in green to indicate that it correctly respects the new policy; otherwise, the exposure is marked in red to indicate a bug. Red markings are propagated to the parent of a node, so that the path through the tree to a bug is immediately apparent.

5.3 Evaluation

To evaluate whether or not Derailer is effective at finding security bugs, the authors used Derailer to examine 127 student assignments from a web application design course at MIT. The assignment was open-ended, so the applications had similar, but not identical, intended security policies. The results show that Derailer was able to highlight significantly more security bugs than were found by the course's teaching assistants during grading.

The project asks students to implement access control for a "Notes" application, which allows users to log in, write short textual posts, and share them with others. The assignment requirements are purposefully vague: students are expected both to design a security policy and to implement that policy. Each assignment must therefore be graded against *its own* intended security policy, making it impossible to write a single specification for all assignments. The existing grading process consists of a teaching assistant running the application and experimenting with its capabilities in a browser, along with extensive code reivew. The teaching assitants estimated that they spent an average of 30 minutes grading each project.

The author, who was not a teaching assistant for this course, used Derailer to evaluate all 127 student submissions for this assignment. We used the constraints present on Note accesses to infer the security policy the student intended to implement, and then we looked for situations in which those constraints *were not* applied. Our goal was not to evaluate the policies the students had chosen—though we found some that did not seem reasonable—but rather to determine whether or not the students correctly implemented those policies. These are the kind of bugs Derailer is intended to find: situations in which the programmer has simply forgotten to enforce the intended security policy.

It took about five minutes per student submission to interpret the results of Derailer's analysis. For most projects, we were able to determine the intended security policy after examining only one or two exposures; we spent roughly a minute as-

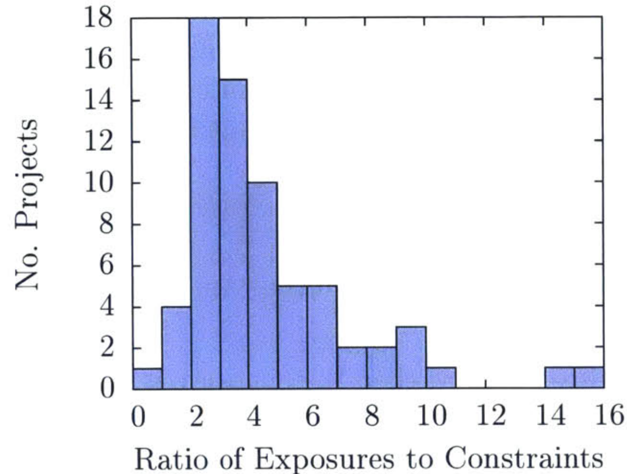


Figure 5-5: Ratio of Exposures to Constraints in Student Projects

sembling constraints into a description of that policy, and then another couple of minutes to decide whether the highlighted exposures were security bugs. Since Derailer points directly to the action responsible for each exposure, confirming each bug in the student's code also took only a couple of minutes.

5.3.1 Results

Figure 5-7 contains information about our analysis, including average, minimums, and maximums for lines of source code, analysis time, number of exposures generated during analysis, and number of unique constraints applied to those exposures. Figure 5-6 contains a histogram of analysis times, showing that the vast majority of analyses took fewer than 10 seconds.

The average number of exposures generated by the analysis was 47. Projects with very few generated exposures were instances in which the student had not completed the project. Projects with a very large number of exposures—the maximum was 236—generally used many different ways to query the database for similar kinds of information. We found it easy to distinguish these cases, because the Rails API places heavy emphasis on making database queries human-readable.

The average number of unique constraints was only 12, and the average ratio of exposures to constraints was 3.1 (meaning that for each unique constraint, there were more than three exposures on average). Figure 5-5 contains a histogram showing that the majority of assignments had exposure-to-constraint ratios close to the average. While these ratios are lower than those for the open-source applications, they are still overwhelmingly greater than one, again supporting our hypothesis that similar data is accessed in similar ways.

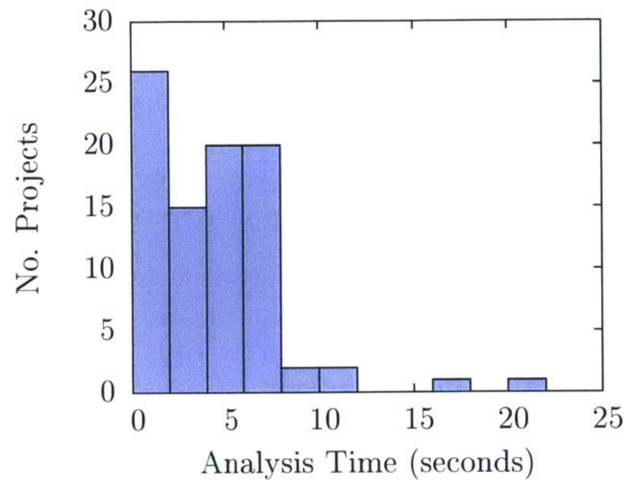


Figure 5-6: Analysis Times for Student Projects

Metric	Avg.	Max.	Min.
Lines of Code	2,125	24,341	278
Analysis Time	4.37s	20.28s	0.23s
Exposures	47	236	4
Unique Constraints	12	45	0

Figure 5-7: Results of Analyzing 127 Student Projects

5.3.2 Bugs Found

Figure 5-4 contains a summary of the bugs we found in student projects using Derailer. Roughly 20% of the students failed to complete the project (they did not implement access control). Another 65% did implement access control, but failed to implement a consistent security policy. In other words, less than 15% of the student assignments were correct.

The bugs we found could be roughly partitioned into the following groups:

- **No access control implemented.** The student has implemented functionality for creating Notes, but not for restricting access to them. The analysis results for these projects no security constraints on any "Note" objects rendered to pages.
- **No write control implemented.** The student has added access control to the application, but has not restricted the ability of users to write to others' Notes. The analysis results show constraints on Note objects in the context of reading (the "index" and "show" actions) but none in the context of writing (the "update" and "destroy" actions).
- **No read control implemented.** The student has added access control for

writing only, meaning that any user can read the Notes of any other user. Results show constraints in write actions but none in read actions.

- **Security bug in read control.** The student has added full access control, but made a mistake allowing users to read Notes they should not have access to. The most common example was implementing read control in the "index" action but forgetting to check the same policy in the "show" action.
- **Security bug in write control.** The student has added full access control, but made a mistake allowing writing that should not be allowed. The most common example was checking access in the "edit" action but forgetting in the "update" and "destroy" actions.

The most common issue seemed to be that students considered only the *most common* method of accessing a piece of data, and failed to consider other ways of accessing it. For example, many students correctly checked for permission when a user loads the "edit" page for a Note, but failed to check again when the user issues a POST request to the "update" action for that Note. Most of the time, users will issue the POST request only after loading the "edit" page, and so will be shown the "access denied" message instead of the editing form. However, a malicious user can construct a POST request directly to the "update" action, bypassing the security check. Since the student did not consider this access path, he or she did not secure it. Derailer is perfect for finding this kind of problem, since it considers *all* the ways data can be accessed.

5.3.3 Comparison with Teaching Assistants

We also compared the set of bugs we found in student assignments with the grading reports given to those students by their teaching assistants. Out of 56 grade reports we obtained, 38 (or 68%) agreed with our analysis. In 17 cases (30%), we found a bug using Derailer that the teaching assistants missed.

Only one assignment contained a bug that the teaching assistants found, but that we missed. In this case, it was possible for a user to grant permissions to a non-existant user ID which might later be associated with some new user. This situation does not cause a sensitive exposure at the time it occurs. Since Derailer only reports exposures—which by definition require some output to the user—our analysis was unable to uncover this bug.

We asked the teaching assistants to validate the security bugs we found, and in each case, they agreed that the additional bugs we found were indeed violations of the student's intended security policy.

5.4 Related Work

Derailer combines static analysis with human interaction to find security bugs. In this section, we discuss the related work in each of these areas, and compare Derailer to other existing solutions.

5.4.1 Interactive Analyses

We share the idea of using human insight as part of a semantic program analysis with Daikon [25] and DIDUCE [28], which use runtime traces to produce possible program invariants and ask the user to verify their correctness. Like Derailer, these tools do not require a specification. However, both systems rely on dynamic analysis (e.g. collecting traces during execution of a test suite) and therefore may miss uncommon cases. Derailer, by contrast, uses symbolic execution to ensure coverage.

Teoh et. al [47] apply a similar strategy to the problem of network intrusion detection, producing visual representations of the current state of the network. Over time, users of the tool learn to recognize normal network states by their visual representations, and can therefore quickly determine when an intrusion has occurred.

5.4.2 Automatic Anomaly Detection

In addition to producing candidate invariants, DIDUCE raises errors at runtime when these invariants are violated, allowing it to run in a completely unsupervised mode. This approach is a type of automatic anomaly detection—an area which has received much attention [14]. Most approaches to anomaly detection use machine learning techniques to learn the appearance of "normal operation," and then use the resulting classifier to automatically find anomalies at runtime. These techniques have not often been applied to code, however, since software specifications are often specialized and difficult to learn.

It may be possible to use an automatic anomaly detection technique along with Derailer's analysis to detect security problems without human input. However, anomaly detection techniques rely on a large training set of examples, often spread across many applications in the same domain; Derailer's results, by contrast, usually contain only a handful of elements per data type, and since security policies are not common across applications, pooling results from many applications is not likely to be helpful.

5.4.3 Run-Time Approaches to Web Application Security

Resin [54] is a runtime system that enforces policies attached to data objects; it has been successfully applied to web applications. Jeeves [53], a similar language for enforcing privacy policies, has also been applied to the web. Jif [36], an extension of Java, also supports checking access-control policies at runtime.

GuardRails [10] allows the programmer to annotate ActiveRecord classes with access control information, then performs source-to-source translation of the application's implementation, producing a version of the application that enforces, at run time, the access control policies specified by the annotations. Nemesis [23] is a similar effort for PHP applications: it is a tag-based system that enforces authentication and access-control at runtime.

Chapter 6

Checking Security Patterns with SPACE

This chapter discusses an automated technique for finding application-specific security bugs using a *theory of access control patterns*. Our theory is a catalog of *security patterns*, each of which models a common access control use case in web applications. We built this catalog based on our experience with real-world web applications, which suggests that while applications often mix and match different security patterns for different kinds of resources, they usually intend for a particular pattern to be applied uniformly to all uses of a given resource type.

Our approach checks that for every kind of data exposure allowed by an application's code, some security pattern in our catalog also allows the exposure. When the application allows a data exposure *not* allowed by a security pattern, we report that exposure as a security bug. This process requires only that the user provide a mapping of application resources to the basic types (such as user, permission, etc.) that occur in our access control patterns. From this information alone, application-specific security bugs are then identified *automatically*, based on the predefined catalog of patterns.

We have built a prototype implementation of this technique, called SPACE (Security PAtern CheckEr). Our implementation uses symbolic execution to extract the set of all possible *data exposures* [37] from the source code of a Ruby on Rails application. The constraints associated with these exposures and the user-provided mapping are passed through a constraint specializer, which uses the mapping to re-cast the constraints in terms of the role-based access control model upon which our catalog of patterns is based. Then, SPACE translates the specialized constraints into the Alloy specification language, and uses the Alloy Analyzer to perform automatic bounded verification that each data exposure allowed by the application is also allowed by a security pattern in our catalog.

Of the 50 most popular open-source Rails applications on Github, 30 implement access control. We have used SPACE to find security bugs in nearly 1/3 of these—a total of 23 unique bugs. Both the symbolic execution and bounded verification steps of our technique scale well to applications as large as 45k lines of code—none of our analyses took longer than 64 seconds to finish.

6.1 Formal Model of Access Control

In this section, we formalize the comparison SPACE makes between a web application's code and our generalized security patterns. The first step is to derive a list of *data exposures* from the application code, representing the different ways in which users of the application can read or update data stored in the database. Second, SPACE uses the *mapping* defined by the user to specialize the security pattern library to the resources defined by the application code. Finally, SPACE verifies that each data exposure allowed by the application code is also allowed by some pattern in the library.

6.1.1 Web Applications

We consider web applications in terms of sets of *Databases*, *Requests*, and *Resources*. An application defines relations *response* and *update* describing (in the context of a given database) the resources sent to the requester and updated in the database, respectively. Note that the updates do not specify the resulting state of the database; our concern is only whether modifications can be made to particular resources, and not the actual values of those modifications.

$$\begin{aligned} \mathit{Databases} &\subseteq \mathcal{P}(\mathit{Resources} \times \mathit{Values}) \\ \mathit{response} &\subseteq \mathit{Databases} \times \mathit{Requests} \times \mathit{Resources} \\ \mathit{update} &\subseteq \mathit{Databases} \times \mathit{Requests} \times \mathit{Resources} \end{aligned}$$

SPACE approximates these relations by using symbolic execution to build the *exposures* relation, representing the application's set of data exposures:

$$\begin{aligned} \mathit{Operations} &= \{\mathsf{read}, \mathsf{write}\} \\ \mathit{Constraints} &\subseteq \textit{first-order predicates over requests} \\ &\quad\; \textit{and databases} \\ \mathit{exposures} &\subseteq \mathit{Requests} \times \mathit{Constraints} \times \\ &\quad\; \mathit{Operations} \times \mathit{Resources} \end{aligned}$$

The *exposures* relation characterizes the conditions (ϕ) under which the application code allows a particular resource to be read or written. $(db, req, res) \in \mathit{response} \Rightarrow \phi$ means that if the application exposes resource res, it *must* be under conditions ϕ—in other words, ϕ captures the conditions the application places on that exposure. The set of read and write exposures is constrained to contain exactly one exposure per possible response or update allowed by the application, each one accompanied by the condition imposed by the application.

$$\begin{aligned} &\forall db, req, res, \phi \,.\, ((db, req, res) \in \mathit{response} \Rightarrow \phi) \iff \\ &\qquad\qquad (req, \phi, \mathsf{read}, res) \in \mathit{exposures} \\ &\forall db, req, res, \phi \,.\, ((db, req, res) \in \mathit{update} \Rightarrow \phi) \iff \\ &\qquad\qquad (req, \phi, \mathsf{write}, res) \in \mathit{exposures} \end{aligned}$$

SPACE builds the *exposures* relation using symbolic execution. It invokes each action of the web application under analysis, passing a symbolic value for the database and for the user-supplied parameters. At each invocation of the render method (which causes Rails to render an HTML page), SPACE enumerates the symbolic values flowing

from the database and parameters to the renderer and constructs an exposure. This exposure has the form $(req, \phi, \mathsf{read}, res)$, where req is the original symbolic request, ϕ is the path condition derived from the symbolic execution, and res is the symbolic expression representing the database resource being exposed.

Similarly, when SPACE finds an invocation of the save or update methods (which cause Rails to update the database), it constructs an exposure of the form $(req, \phi, \mathsf{write}, res)$, where req is the request, ϕ is the current path condition, and res is a symbolic expression representing the database resource being updated.

6.1.2 Role-Based Access Control

As a basis for representing security policies, we adopt the role-based access control model of Sandhu et al. [42] (specifically, $RBAC_0$). The standard model of role-based access control consists of sets *Users*, *Roles*, *Permissions*, and *Sessions*, over which the following relations are defined to assign permissions and users to roles:

$$\begin{aligned} permission_a &\subseteq Permissions \times Roles \\ user_a &\subseteq Users \times Roles \end{aligned}$$

We adopt the extension of this model by Ferraiolo et al. [26] with *Objects* and *Operations*, where *Objects* contains the targets of permissions (which will correspond to the web application's resources) and *Operations* (for our applications) contains the operation types read and write already used in the definition of the *exposures* relation. This model defines *Permissions* to be a set of mappings between operations (either read or write) and objects allowed by that permission:

$$Permissions = \mathcal{P}(Operations \times Objects)$$

6.1.3 Security Pattern Catalog

Each pattern in our catalog specializes the generic RBAC model by adding specific constraints on these relations. These constraints are designed to prevent users from accessing objects in ways that differ from the common web application use-cases. Each one of these pattern definitions corresponds to a standard use case for resources in web applications; in effect, the patterns "whitelist" the kinds of data accesses that we expect to succeed.

Access Pattern 1: Ownership. In most applications, resources created by a user "belong" to that user, and a resource's creator is granted complete control over the resource. To express this use case, we define a relation *owns* between users and objects, and then allow those owners to perform both reads and writes on objects they own:

$$\begin{aligned} & owns \subseteq Users \times Objects \\ & \forall u, o \,.\, (u, o) \in owns \Rightarrow \\ & \quad \exists r \,.\, (u, r) \in user_a \land \\ & \quad\quad ((\mathsf{read}, o), r) \in permission_a \land \\ & \quad\quad ((\mathsf{write}, o), r) \in permission_a \end{aligned}$$

Access Pattern 2: Public Objects. Many applications make some resources public. A blog, for example, allows anyone to read its posts. This pattern defines *PublicObjects* to be a subset of the larger set of *Objects*, and allows anyone to read (but not write) those objects:

$$\begin{aligned} &PublicObjects \subseteq Objects \\ &\forall u, r, o, p \,.\, o \in PublicObjects \wedge (u, r) \in user_a \Rightarrow \\ &\qquad ((\mathsf{read}, o), r) \in permission_a \end{aligned}$$

Access Pattern 3: Authentication. Every application with access control has some mechanism for authenticating users, and many security holes are the result of the programmer forgetting to check that the user is logged in before allowing an operation. To model authentication, this pattern defines *logged_in*, a (possibly empty) subset of *Users* representing the currently logged-in users, and constrains the system to allow permission only for logged-in users (except for public objects):

$$\begin{aligned} &logged_in \subseteq Users \\ &\forall u, r, o, op, p \,. \\ &\quad (u, r) \in user_a \wedge ((op, o), r) \in permission_a \Rightarrow \\ &\qquad (op = \mathsf{read} \wedge o \in PublicObjects) \vee \\ &\qquad u \in logged_in \end{aligned}$$

Access Pattern 4: Explicit Permission. Some applications define a kind of resource representing permission, and store instances of that resource in the database. Before allowing access to another resource, the application checks for the presence of a permission resource allowing the access. To model this use case, this pattern defines *PermissionObjects* to be a subset of *Objects*, defines a relation *permits* relating a permission object to the user, operation, and object it gives permission to, and allows users to perform operations allowed by permission objects:

$$\begin{aligned} PermissionObjects &\subseteq Objects \\ permits &\subseteq PermissionObjects \times Users \times \\ &\quad\; Operations \times Objects \end{aligned}$$

$$\begin{aligned} &\forall u, o, p, op \,.\, (p, u, op, o) \in permits \Rightarrow \\ &\quad \exists r \,.\, (u, r) \in user_a \wedge ((op, o), r) \in permission_a \end{aligned}$$

Access Pattern 5: User Profiles. Applications with users tend to have profile information associated with those users, and allow other users to view that information. Programmers commonly forget checks requiring that the user updating a profile must be the owner of that profile; this pattern constrains the allowed writes on users so that no user can update another user's profile:

$$\forall u \,.\, \exists r \,.\, (u, r) \in user_a \wedge ((\mathsf{write}, u), r) \in permission_a$$

Access Pattern 6: Administrators. Many applications distinguish a special class of users called *administrators* that have more privileges than normal users. We can represent these users with a special role called Admin, which grants its users full permissions on all objects:

$$\begin{aligned}\mathsf{Admin} &\in Roles \\ \forall o\ .\ ((\mathsf{read}, o), \mathsf{Admin}) &\in permission_a \wedge \\ ((\mathsf{write}, o), \mathsf{Admin}) &\in permission_a\end{aligned}$$

Access Pattern 7: Explicit Roles. A few applications specify distinct roles and represent them explicitly using a resource type. They then assign roles to users and allow or deny permission to perform operations based on these assigned roles. This pattern introduces a level of indirection to allow mapping these resource-level role definitions to the RBAC-defined set of roles:

$$\begin{aligned}RoleObjects &\subseteq Objects \\ object_roles &\subseteq RoleObjects \times Roles \\ user_roles &\subseteq Users \times RoleObjects \\ \forall ro, r, u\ .\ (ro, r) &\in object_roles \wedge \\ (u, ro) &\in user_roles \Rightarrow \\ (u, r) &\in user_a\end{aligned}$$

6.1.4 Mapping Application Resources to RBAC Objects

To compare the operations allowed by an application to those permitted by our security patterns, a mapping is required between the objects defined in the RBAC model and the resources defined by the application. In many cases, this mapping is obvious (a resource named "User" in the application, for example, almost always represents RBAC users), but in general it is not possible to infer the mapping directly.

SPACE asks the user to define this mapping. Formally, it is a mapping from *types* of application resources to *types* of RBAC objects; the mapping is a relation, since some application resources may represent more than one type of RBAC object. The set of resource types can be derived from the application's data model (which is present in its source code), and the set of RBAC object types is based on the formal model of RBAC defined here.

$$\begin{aligned}\tau_{Resource} &= \textit{application-defined types} \\ \tau_{RBAC} &= \{\mathsf{User}, \mathsf{Object}, \mathsf{PermissionObject}, \\ &\quad \mathsf{OwnedObject}, \mathsf{PublicObject}, \\ &\quad \mathsf{RoleObject}\} \\ map &\subseteq \tau_{Resource} \times \tau_{RBAC}\end{aligned}$$

We also need to provide definitions from the application for the new concepts introduced in our pattern definitions: *PublicObjects*, *PermissionObjects*, and the *owns* and *permits* relations. The *map* relation can be used to define public and permission objects, but we must define a separate mapping from field names of resources to the corresponding relations they represent in our security patterns. We use map_{fields} for this purpose.

$$\begin{aligned}FieldNames &= \textit{application-defined field names} \\ RelationNames &= \{\mathsf{owns}, \mathsf{permits}, \mathsf{logged_in}, \\ &\quad \mathsf{object_roles}, \mathsf{user_roles}\} \\ map_{fields} &\subseteq FieldNames \times RelationNames\end{aligned}$$

Finally, we define *session* relating web application *Requests* and the currently logged-in RBAC *User*:

$$session \subseteq Requests \times Users$$

The *session* relation is needed to determine the currently logged-in user (if any—some requests are sent with no authentication) of the application. Since Rails has session management built in, this mapping can be inferred automatically.

6.1.5 Finding Pattern Violations

To find bugs in a given application, the goal is to find exposures that are not allowed by some security pattern. For each exposure in *exposures*, this process proceeds in two steps.

To check exposure (req, ϕ, op, res):

1. Build a *specialized constraint* ϕ' from ϕ by substituting RBAC objects for application resources using the *map* relation supplied by the user.

2. Determine whether or not some pattern allows the exposure by attempting to falsify the formula:

   $\phi' \Rightarrow (u, r) \in user_a \Rightarrow ((op, obj), r) \in permission_a$

   where u, r and obj are RBAC objects corresponding to the current user sending req and the resource res. Intuitively, this formula holds when the conditions imposed by the application imply that some pattern allows the exposure.

Building ϕ'. We use *map* to build a specialized constraint ϕ' from ϕ as follows:

- Replace each reference res to an application resource in ϕ with an expression $\{\tau | (res, \tau) \in map\}$ representing the corresponding set of possible RBAC objects.

- Replace each field reference $o.fld$ in ϕ with an expression $\{o' | (fld, r) \in map_{fields} \wedge (o', o) \in r\}$ representing a reference to the corresponding relation defined by our security patterns.

Intuitively, this means replacing references to application resources with the corresponding RBAC objects (based on the user-supplied *map*), and replacing references to fields of resources that represent object ownership, permission type, or permission for a particular object with formulas representing those concepts in terms of RBAC objects.

Checking Conditions. For each $(req, \phi, op, res) \in exposures$:

- Let ϕ' be the result of performing substitution on ϕ using the user-supplied *map*.

- Let obj be the RBAC objects corresponding to the application resource res, so that if $(res, o) \in map$ then $o \in obj$.

- Let u be the current user, so that $(req, u) \in session$.
- Then the following formula must hold:

  $\phi' \Rightarrow \forall r.\ (u, r) \in user_a \wedge ((op, obj), r) \in permission_a$

This process checks that if the specialized condition holds, then a pattern exists allowing the current user u to perform operation op on the RBAC object obj corresponding to the resource res acted upon by the application code. If a counterexample is found, it means that the application allows the user to perform a particular operation on some object, but no security pattern exists allowing that action. In other words, such a counterexample represents a situation that does not correspond to one of our common use-cases of web applications, and is likely to be a security bug.

6.2 SPACE: A Pattern-Based Bug Finder

SPACE is designed primarily to find mistakes in the implementation of an application's security policy. Most often, these mistakes take the form of missing security checks. In this section, we describe an example open-source application (MediumClone) and demonstrate how we used SPACE to find security bugs in its implementation.

6.2.1 Example Application: MediumClone

MediumClone[1] is a simple blogging platform designed to mimic the popular web site Medium. Using MediumClone, users can read posts and log in to create new posts or update existing ones. The site requires users to sign up and log in before writing posts, and prevents users from modifying others' posts.

MediumClone is written in Ruby on Rails[2], a popular web application programming framework for the Ruby programming language. A Rails application is composed of *actions*, each of which handles requests to a particular URL within the application. *Controllers* are collections of actions; conceptually, each controller's actions implement an API for interacting with a *resource* exposed by the site. Resources are defined using the ActiveRecord library, which implements an *object-relational mapper* to persist resource instances using a database, and which provides a set of built-in methods for querying the database for resource instances.

Figure 6-1 contains a part of the controller code for MediumClone's UserController, which define actions to provide a RESTful API for user profiles. Following the REST convention, the show action is for displaying profiles, the edit action displays an HTML form for editing the profile, and submitting that form results in a POST request to update action, which actually performs the database update (using update_attributes). The call to before_filter installs a *filter*—a procedure that runs before the action itself—for the show and edit actions. The filter checks that the logged-in user has permission to perform the requested action, and redirects them to an error page if

[1] https://github.com/seankwon/MediumClone

[2] http://rubyonrails.org/

```
class UserController < ApplicationController
  before_filter :signed_in_user, :only => [:show, :edit, :update]
  before_filter :correct_user, :only => [:show, :edit]
  ...
  def show
    @user = User.find(params[:id])
    @posts = find_posts_by_user_id @user.id
    @editing = true if signed_in?
  end

  def edit
    @user = User.find(params[:id])
    @url = '/user/' + params[:id]
  end

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(user_params)
      redirect_to @user, success: 'Editing successful!'
    else
      redirect_to edit_user_path(@user.id), error: 'Editing failed!'
    end
  end
end
```

Figure 6-1: Controller Code for MediumClone

not. This use of filters—to enforce security checks—is common in Rails applications, and helps to ensure that checks are applied uniformly to all the appropriate actions.

6.2.2 MediumClone's Security Bug

The code in Figure 6-1 contains a security bug: it fails to apply the `correct_user` filter to the `update` action. As a result, an attacker can craft an HTTP POST request and send it directly to the `update` URL of the MediumClone site to update *any* user profile. The filter *is* properly applied to the `edit` action, ensuring that a user can only retrieve this page for his or her own profile.

In our experience, this kind of mistake is common in web applications. The developer's assumption is that users will use the interface provided by the site (in this case, the user will use the `edit` page to make changes, and will be shown an error message if that action is not allowed), and will not craft malicious requests to constructed URLs. As a result, developers often do not consider paths to a particular piece of action code that are not accessible through the application's interface, and therefore forget to include vital security checks.

6.2.3 Finding the Bug Using SPACE

SPACE compares the checks present in the application code against its built-in catalog of common security patterns. When the application code is missing security checks required by the appropriate pattern, SPACE reports a bug in the code.

Mapping to Role-based Access Control. To ensure generality, SPACE's security patterns are formalized in terms of the concepts of the role-based access control (RBAC) model. This model defines sets of *users, roles, operations,* and *objects*, and defines a set of *permissions* allowing an operation on an object.

SPACE's formal models of security patterns enforce security checks by restricting the definitions of these relations. In order to check these patterns against a piece of code, the user provides a mapping of the application's resources to the sets of conceptual objects defined in the RBAC model. MediumCloneâĂŹs `User` type represents the *User* type in the RBAC pattern, and `Posts` are special RBAC objects that have an owner. The user provides this mapping for MediumClone as follows:

```
Space.analyze do
  mapping User: RBACUser,
          Post: OwnedObject(user: owns)
end
```

Running SPACE. SPACE takes the mapping defined above and MediumClone's source code and attempts to find a security violation based on its catalog of security patterns. SPACE uses symbolic execution to discover the checks (security-related and otherwise) present in the application code, then verifies that those checks *imply* the set of checks required by the pattern catalog. Intuitively, this is the case when the set of checks present in the code is a superset of those present in the pattern catalog, so

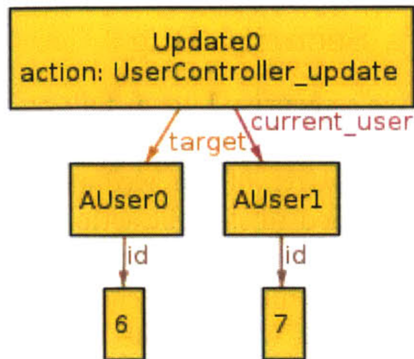


Figure 6-2: SPACE Counterexample Showing MediumClone Security Bug: user can update another user's profile

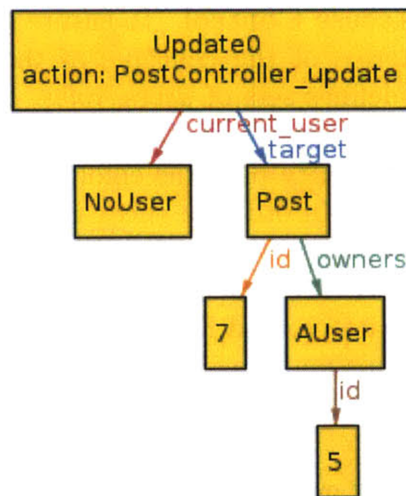


Figure 6-3: SPACE Counterexample Showing a Second MediumClone Security Bug: unauthenticated user ("NoUser") can update any post

checks unrelated to security issues simply do not contribute to satisfying this property (and are therefore ignored).

The mapping provided by the user translates between the RBAC concepts constrained by the pattern catalog and the resource types defined in the application code. SPACE uses this mapping to specialize the constraints derived from the checks present in the code to the set of RBAC objects, so that the two sets of security checks can be compared.

SPACE compares the specialized constraints against the constraints required by the applicable pattern from the catalog. When MediumClone exposes user profiles, this is the *User Profile Pattern*, which requires that users can update only their own profiles.

When SPACE finds a missing check, it builds a counterexample demonstrating the security vulnerability caused by the missing check. The counterexample is specific to the application code under analysis: it involves resource types defined in the code, rather than RBAC concepts. SPACE also reports the check present in the pattern catalog that is missing in the application code.

For MediumClone, SPACE produces the counterexample shown in Figure 6-2. The "Update0" box indicates that an update is possible on the profile of "AUser0" (the "target"—a renaming of "User" to avoid clashes) by the distinct "AUser1" (the "current_user", or currently logged-in user).

This counterexample demonstrates a user updating another user's profile using the update action—a situation that, according to the security pattern catalog, should not be allowed. The bug can be fixed by adding :update to the list of actions for which the :correct_user filter is used to enforce security checks.

More Bugs in MediumClone. Running SPACE on the fixed MediumClone code, we discover a new counterexample, shown in Figure 6-3. This counterexample suggests that Post resources in MediumClone can be updated not only by users who did not create them, but by users who are not logged in ("NoUser") at all! The code for PostController begins as follows:

```
class PostController < ApplicationController
  before_filter :signed_in_user, :only => [:new, :create, :edit]

  def new
    @post = Post.new
    @url = '/post/create'
  end
  ...
end
```

This controller definition checks that the user is logged in before performing the new, create, and edit actions, but omits this check for the update action. In addition, the definition makes no attempt to ensure that the current user owns the post being modified.

This bug is especially interesting because a condition of the intended security policy—that users only modify their own posts—is forgotten across the entire appli-

cation codebase. This mistake exemplifies a class of bugs that are difficult to detect with testing or with existing consistency checkers like Derailer [37].

6.3 Implementation of SPACE

We have developed an implementation of our technique, called SPACE (Security PAtern CheckEr), that finds security bugs in Ruby on Rails web applications. SPACE uses the symbolic execution framework previously developed for Derailer [37] to extract a set of exposures from a given application. Then, SPACE's constraint specializer uses the resource-to-object mapping provided by the user to specialize the constraints of each exposure to the set of role-based access control (RBAC) objects. The resulting specialized constraints are written in the Alloy specification language, a variant of first-order relational logic with transitive closure; the basic RBAC framework and our security patterns are also specified in Alloy. Finally, SPACE uses the Alloy Analyzer—an automatic bounded verifier for properties of Alloy specifications—to find security bugs by comparing the specialized constraints against the formal models of security patterns. Figure 6-4 contains a graphical overview of SPACE's architecture.

6.3.1 Extracting Exposures

SPACE uses the symbolic execution framework we defined in chapter 4 to extract exposures from a target Rails application. SPACE enumerates the actions defined by the application; for each one, it constructs a symbolic request and runs the action symbolically. The set of symbolic objects present on the resulting rendered pages is the set of possible exposures for that action.

Rendering. A Rails action serves a request in two steps: first, the code defined in the controller populates a set of instance variables; then, Rails evaluates an appropriate *template*—which may reference those instance variables—to produce an HTML string. SPACE wraps the Rails renderer to extract the set of symbolic values that appear on each rendered page. These values, along with the constraints attached to them, contribute to the set of exposures resulting from the action being executed.

6.3.2 Constraint Specializer

The constraint specializer substitutes objects from our model of role-based access control for references to application resources in the constraints present on the exposures generated in the symbolic execution phase. The user provides a mapping defining how this substitution should proceed; the specializer uses the mapping to perform substitution and translate the specialized constraints into the Alloy language.

User-provided Mapping. As in the example in Section ??, the user provides the mapping between application resources and role-based access control objects using a SPACE-provided embedded domain-specific language. This mapping specification has the following form:

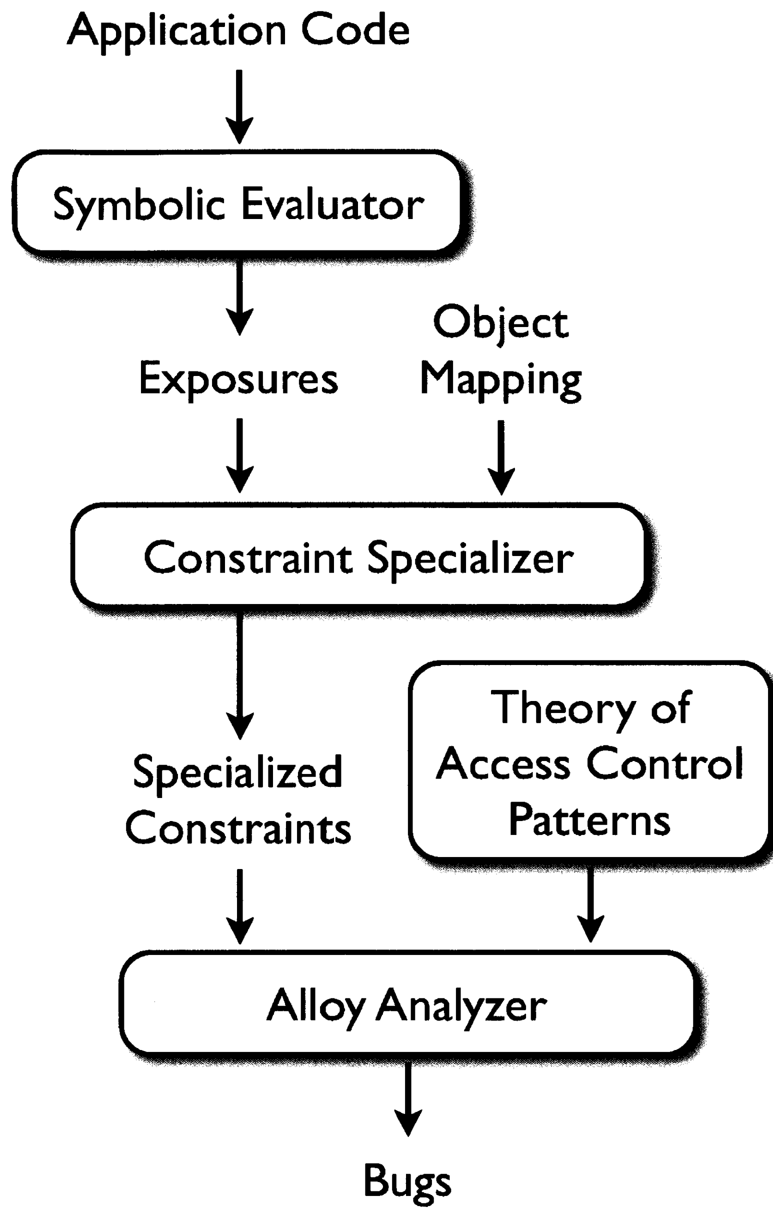


Figure 6-4: Summary of the Architecture of SPACE

```
Space.analyze do
  mapping map_spec
end
```

where **map_spec** is a dictionary (a Ruby hash) mapping application-defined resource types and RBAC object types (for example, User: RBACUser means that the User class represents RBAC users). The RBAC object types optionally accept another dictionary as an argument, to specify a mapping between field names of the application-defined resources and relation names of the pattern catalog (for example, Post: OwnedObject(user: owns) means that the Post class is an RBAC object with an owner, and the user field of that class stores the post's owner).

Resource/Object Substitution. As in Section 6.1.5, the specializer replaces all references to names in the *Resource* set with references to their corresponding RBAC objects in *Object*. A field reference $r.f$ is replaced with a constraint in terms of the relations defined by our patterns using the mapping from *FieldName* to *MapRelation* (if one is present). Some field references are not security-related; for these, since no mapping is provided, the specializer leaves them unchanged.

Translation to Alloy. Finally, the specializer translates the substituted constraints into Alloy for comparison to the security pattern models. Since Alloy is based on first-order relational logic, this translation is straightforward.

6.3.3 Pattern Library

We model web applications and role-based access control in Alloy after the formal description given in Section **??**. We also model the security patterns described in Section 6.1.3. Each pattern defines an Alloy predicate specifying the general constraints it imposes on the security checks required in an application. The patterns that define new subsets of the set of RBAC objects are modeled Alloy's *extends* keyword, and the new relations defined by the pattern catalog are represented using Alloy's global relations.

6.3.4 Bounded Verification using the Alloy Analyzer

Alloy [30] is a specification language based on first-order relational logic with transitive closure. SPACE uses Alloy to represent both the complete pattern catalog presented in Section **??** and the specialized constraints derived from an application's exposures.

SPACE uses the Alloy Analyzer to compare the specialized constraints to the security pattern models and find bugs. The Alloy Analyzer is a tool for automatic bounded analysis of Alloy specifications; it places finite bounds on the number of each type of atom present in the universe, then reduces the specification to propositional logic. Alloy's model finder, Kodkod [50], handles the translation from relational logic to a satisfiability problem using algorithms that optimize relational expressions. This makes Kodkod, and thus Alloy, especially suited to database-backed applications.

Application	LOC	Exposures	Exposure Generation Time	Verification Time	Bugs Found	False Positives	Finite Bound Required
rails3-bootstrap-devise-cancan	1103	16	45.02 s	7.66 s	0	0	-
fb-graph-sample	655	87	41.13 s	8.31 s	2	1	2
jquery-fileupload-rails-paperclip	727	15	28.71 s	7.11 s	0	0	-
ember-auth-rails-demo	742	28	28.94 s	6.48 s	0	0	-
ember-rails-devise-demo	619	24	29.84 s	3.25 s	0	0	-
sportbook	3568	94	49.67 s	7.16 s	4	0	3
rails-container-and-engines	778	39	23.94 s	7.20 s	0	3	-
TOT2	1470	54	34.07 s	5.21 s	2	0	2
bootstrap-rails-startup-site	389	17	41.00 s	4.72 s	0	0	-
furatto-rails-start-kit	293	16	44.69 s	4.95 s	0	1	-
postgis-on-rails-example	206	22	21.56 s	4.14 s	0	0	-
jqm-rails	340	43	24.66 s	3.84 s	3	0	2
rails-4-landing	800	21	41.67 s	3.09 s	0	0	-
redport	142	9	22.68 s	3.73 s	0	0	-
league-tutorial	833	31	22.50 s	3.14 s	0	2	-
requirejs-rails-jasmine-template	197	11	23.36 s	8.17 s	0	0	-
grinch	1059	24	32.27 s	5.44 s	0	0	-
rails-angularjs-example	423	17	23.31 s	8.18 s	0	0	-
railscrm-advanced	2373	132	59.75 s	4.63 s	5	0	3
dreamy	243	13	22.14 s	8.84 s	0	1	-
tada-ember	200	8	26.60 s	6.67 s	0	0	-
ribbit	473	41	24.53 s	3.35 s	2	0	2
test-signet-rails	282	20	27.05 s	4.51 s	0	0	-
yelp	1077	48	21.78 s	6.55 s	0	1	-
world.db.admin	866	63	27.95 s	7.22 s	0	0	-
MediumClone	797	46	38.43 s	3.43 s	4	0	2
permissions	523	58	22.82 s	7.82 s	0	0	-
rails-devise-backbone-auth	296	14	26.19 s	7.69 s	0	1	-
geochat-rails	9596	89	49.32 s	3.99 s	0	0	-
rails-api-authentication	1003	51	32.22 s	4.56 s	1	0	2

Figure 6-5: Results of Running SPACE on the 50 Most-Popular Open-Source Rails Applications on Github

To perform the analysis, SPACE builds a single Alloy specification containing the model of role-based access control, the definitions of security patterns, an *Operation* definition for each exposure, and a predicate application_policy imposing the specialized constraints on those operations. Finally, SPACE invokes the Analyzer to check that for all exposures, application_policy implies pattern_catalog—the name of the predicate in our model that imposes the constraints from our pattern catalog.

6.4 Evaluation

In evaluating SPACE, we examined two questions:

- **Is SPACE effective at finding bugs?** We applied SPACE to the 50 most popular open-source Ruby on Rails applications on Github. Of those applications implementing access control, SPACE found bugs in nearly one-third—a total of 23 bugs. SPACE reported a total of 10 false positives in addition to these actual bugs. These results suggest that SPACE is effective at finding security bugs and does not produce excessive false positives.

- **How does the bound selected for verification affect the number of bugs found and the scalability of the analysis?** Since SPACE uses our existing symbolic execution framework, our earlier experiments suggest that exposure generation scales well to large applications. The addition of a bounded verification step, however, creates new scalability challenges. For the bugs found in the experiment described above, we re-ran the bounded verification step at progressively lower finite bounds until the verifier no longer detected the bug. The minimum finite bound sufficient for all of the bugs we found was 3; SPACE uses a bound of 8 (meaning 8 objects of each type are included in the finite universe considered by the analysis) by default. We recorded the verification time for these analyses, and found that for a bound of 8, the verification step finished in under 10 seconds for all applications. These results suggest that SPACE's default bound of 8 is a good compromise between minimizing verification time and maximizing the number of bugs detected.

6.4.1 Bug-Finding and False Positives

We tested SPACE on the 50 most-popular open-source Ruby on Rails applications hosted by Github. We approximated popularity based on the number of "stars" a project has. These applications fall into two basic categories: mature, stable applications intended for installation by end-users (which are popular because of their user base), and fragments of example code intended for helping programmers to build new applications (which are popular because they are copied and pasted into other applications). Security bugs are a serious issue in both types of application: in mature applications, they make user data vulnerable, while in example code, they can propagate to other applications that have copied the code.

Building Mappings. We first performed a brief examination of each application's code, and eliminated those applications with no access control whatsoever. This left 30 applications implementing some form of access control. For each of these 30 applications, we constructed a mapping from application resource types to RBAC object types as detailed in section 3. We used application documentation and the code itself as a guide in constructing the mapping; in most cases, it was a trivial process, involving mapping the "User" type to RBAC users, a "Permission" type to permission objects, a subset of resources to owned objects, and perhaps one or more resources representing roles to role objects. This process took us approximately 10 to 15 minutes per application.

Experimental Setup. We used Ubuntu GNU/Linux on a virtual machine equipped with a single Intel Xeon core at 2.5GHz and 2GB of RAM. We used RVM to install the version of Ruby required by each application and Bundler to install each application's dependencies. We ran SPACE on each application using its default finite bound of 8 (atoms per data type) for the verification step. After running the experiment, we examined each reported bug and classified it as either a duplicate, a false positive, or a real bug.

No. Bugs	Pattern Violated
5	Ownership
2	Public Objects
10	Authentication
3	Explicit Permission
3	User Profiles

Figure 6-6: Classification of Bugs Found by Pattern Violated

Results. The results of the experiment are summarized in Figure 6-5. In total, SPACE reported 23 bugs and 10 false positives. The longest analysis took 64 seconds, while the average time was 38 seconds. In most cases, the symbolic execution step (generating the exposures) took most of the time. All of the bugs we found would have been detected using a finite bound of 3—well below SPACE's default bound of 8.

Bugs Found. Figure 6-6 classifies the 23 bugs we found according to the pattern they violated. The largest category by far is Authentication, indicating that the check programmers forget most often is the one ensuring that the user is logged in. The next largest categories, Ownership and Explicit Permission, indicate situations in which a user is allowed to view or modify a resource owned by someone else without permission. The User Profiles category includes situations where a user can modify another user's profile, and the Public Objects category includes bugs where a resource was marked public when it should not have been.

We did not find any bugs violating the Administrator or Explicit Roles patterns. Where the applications in our experiment used these patterns, they used them correctly.

- **Authentication failure (6/23)**: the programmer forgot to make sure that the user was logged in before allowing access (either read or write) to a resource.
- **Forgotten Read Check (5/23)**: the programmer forgot a security check, allowing the user to read a resource he or she did not have permission to see.
- **Forgotten Write Check (12/23)**: the programmer forgot a security check, allowing the user to write to a resource he or she did not have permission to modify.

The vast majority of vulnerabilities we discovered involved accessing a URL directly, rather than clicking on links defined by the programmer. This is the kind of security bug that is difficult to detect using testing, because the programmer writing the test cases is biased towards the standard use cases of the application, and often ignores access paths outside of those use cases.

Incorrect Mappings. SPACE relies on the user-provided mapping both to determine the correct pattern and to correctly populate the set of RBAC objects for its analysis.

If the wrong pattern is applied, then SPACE could miss bugs: for example, if all objects are marked public in the mapping, then SPACE will not report any read access as a bug. An incorrect mapping can also cause false positives: applying a too-restrictive pattern will yield bugs on accesses that the developer actually intended. We consider false positives less problematic than false negatives, because the user receives some feedback that a problem exists (rather than a silent failure to find a bug).

In our experiments, we mitigated both problems by starting with very restrictive patterns and using the false positives reported to refine our mappings. This was effective for our experiment, since we did not always know ahead of time what the target applications were intended to do. We expect this strategy will also be useful for users, who will be even less likely than we were to create buggy mappings (since they know their own applications well).

False Positives. In our experiment, SPACE produced a total of 10 false positives. All of these were situations in which the application exposed a particular field of a resource in a different way from the other fields of that resource. Because SPACE maps whole resource types to RBAC objects, it does not allow per-field permissions to be specified.

However, this limitation is not inherent in our approach—it is a decision made in the design of SPACE to make constructing the mapping from resources to RBAC objects easier. A finer-grained mapping would eliminate these false positives, but constructing it would require more effort from the user. For our prototype implementation, we decided that users would find it easier to examine and discard false positives than to spend more time constructing the mapping.

False Negatives. The possibility of false negatives *is* inherent to our approach: our security patterns are intended to capture common use cases of web applications, but some applications may intend to implement a security policy that is *more restrictive* than the common cases expressed by our patterns. A mistake in such an implementation would not be caught by our technique.

For example, an e-commerce web site may implement administrative access, but may disallow even administrators from viewing customers' saved credit card numbers. Our catalog, on the other hand, includes a pattern allowing administrators to view everything. In this case SPACE would not raise a false positive, since every exposure allowed by the application is also allowed by the pattern. However, if the programmer introduced a bug allowing administrators to view credit card numbers, then SPACE would *not* find the bug, since the intended security policy is actually more restrictive than the pattern SPACE has applied.

We examined the code of the applications in our experiment for precisely this situation—security policies intended (based on evidence in the code itself) to be more restrictive than the corresponding patterns in our catalog—and found none. Given the correct user-provided mapping, the patterns applied by SPACE were always at least as restrictive as those enforced by the target applications.

This result gives us confidence that SPACE did not miss any bugs. Since our patterns were just as restrictive as the policies enforced by the applications, any mistake in enforcing those policies would have contradicted the corresponding pattern,

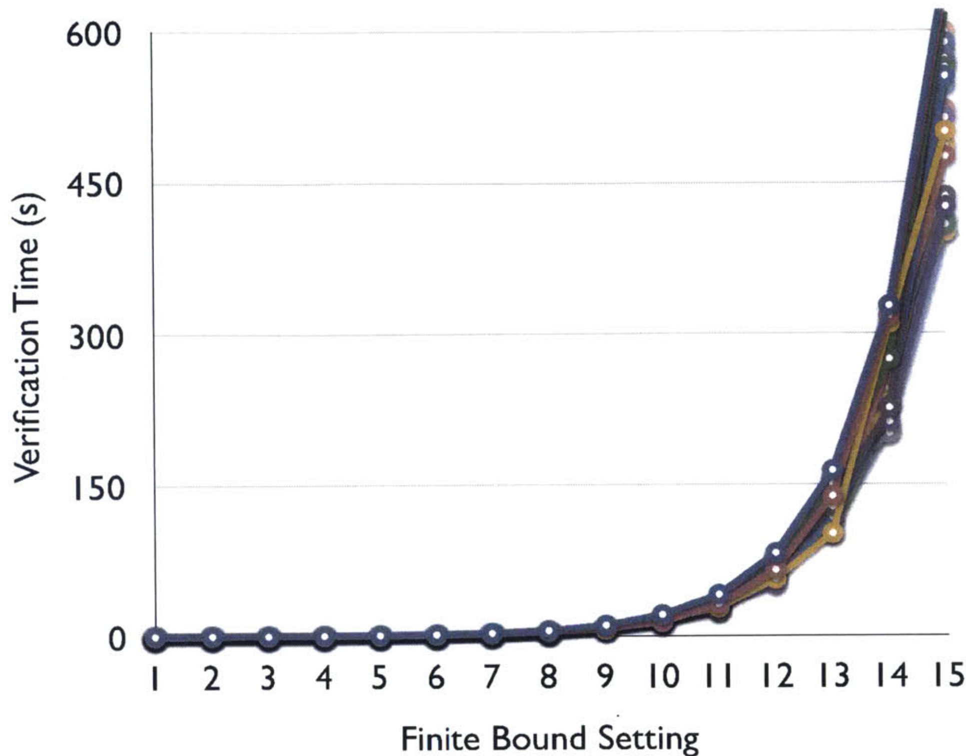


Figure 6-7: Effect of Finite Bound on Verification Time

and resulted in a bug report.

6.4.2 Choice of Finite Bounds

SPACE uses the Alloy Analyzer, a bounded first-order verifier, to perform its verification step. For each analysis, the Alloy Analyzer requires the user to place finite bounds on the number of objects of each type it will consider. Since these bounds can affect not only the scalability of SPACE but also its ability to find bugs (because bugs requiring a counterexample larger than the finite bound will be missed), choosing the default bound carefully is vital.

To determine the effect of the choice of finite bounds on the time taken in the verification step, we ran that step against the applications in Figure 6-5 using finite bound settings from 3 to 15. Figure 6-7 contains the results, which indicate that the verification step seems to run in an acceptable time at finite bound settings below 10. Above that bound, the verification time grows quickly. This experience—that performance is good below a particular "threshold" but degrades quickly thereafter—is common when using the Alloy Analyzer.

We used these results, along with the knowledge that a finite bound of 3 was sufficient to detect all the bugs we found, to select SPACE's default bound of 8. Users of SPACE are welcome to select a different bound in order to tune the verifier for either faster running time or more confidence that no further bugs exist.

6.5 Related Work

Conceptually, our catalog of security patterns is similar to the built-in specifications of common bugs that static analysis tools have used for decades: memory errors, race conditions, non-termination, null pointer exceptions, and so on. Like SPACE, tools that attempt to find these classes of bugs can run without user input. However, these specifications represent a blacklist—a model of situations that should *not* occur—in contrast to our catalog of patterns, which whitelist only those situations that *should* occur.

Our pattern catalog is similar in approach to the formal model of web security by Akhawe et al. [1], which defines a general model of the actors (browsers, servers, attackers etc.), and threats (XSS, CSRF, etc.) involved the larger picture of web security problems, then uses a constraint solver to discover attacks that potentially involve several different components; the model is not directly applicable to code. Our technique, by contrast, focuses on mistakes in access control policies, and compares the model against the code automatically.

Commercially-available web application "scanners" also use built-in libraries of specifications to automate their analyses, but these are typically specifications of *attacks*, and in contrast to our whitelist-based approach, are used as a blacklist to find potential attacks. Examples of such tools include the Trustwave App Scanner[3], netsparker[4], IBM AppScan[5], and HP WebInspect[6]. These tools are useful for finding common cross-application bugs, such as injection, XSS, and CSRF, but cannot find the application-specific bugs we target.

[3]https://www.trustwave.com/Products/Application-Security/App-Scanner-Family/
[4]https://www.netsparker.com/web-vulnerability-scanner/
[5]http://www-03.ibm.com/software/products/en/appscan
[6]http://www8.hp.com/us/en/software-solutions/webinspect-dynamic-analysis-dast/

Chapter 7

Full-functional Verification with Rubicon

This chapter discusses Rubicon, a bounded verifier for Ruby on Rails applications. Rubicon allows programmers to write specifications of the behavior of their web application and performs automatic bounded analysis to check those specifications against the implementation. Rubicon's specification language is expressive but based on a popular domain-specific testing language, and its analysis is implemented as a Ruby library.

Rubicon's specification language extends the RSpec testing language [16] with the quantifiers of first-order logic, allowing programmers to replace RSpec tests over a set of mock objects with general specifications over all objects. This compatibility with the existing RSpec language allows converting existing test cases into specifications.

Rubicon's automated analysis comprises two parts. First, Rubicon generates verification conditions from the code and specifications; second, it invokes a constraint solver to check those conditions. Rubicon uses symbolic execution to execute both code and specification, producing verification conditions rather than values. To check the verification conditions, Rubicon compiles them into Alloy [30], a lightweight specification language whose analyzer is an automatic, bounded model finder for relational first-order logic. Alloy's logic is a good match because its semantics closely match those of relational databases, but the solving of the verification conditions is a separate problem, and in principle might be handled with a different technology (e.g. an SMT solver or theorem prover).

We evaluated Rubicon on five open-source web applications for which the original developers had already written RSpec tests. The author converted a random sample of these tests into Rubicon specifications; in every case, Rubicon's analysis took no more than a few seconds per specification. In the largest of these applications, a customer relationship management system called Fat Free CRM, Rubicon's analysis uncovered a previously unknown security bug. The authors of Fat Free CRM have acknowledged and fixed this bug.

7.1 Specifying Behavior

Rubicon provides an embedded domain-specific language for writing specifications. This language is based on RSpec [16], a testing framework for the Ruby language whose goal is to make testing easier and more useful, and that has made test-driven design popular amongst Ruby programmers. RSpec tests are concise, avoid repetition, and resemble English specifications of application features. The framework also encourages programmers to write documentation for each test by providing fields for that documentation and using it to generate error reports when tests fail.

The Rubicon language is designed for writing specifications of Rails applications. Rails is a popular web programming framework for Ruby, and was designed specifically to allow testing with RSpec. The integration of Rails with RSpec means that test-driven development is just as popular in the Rails community as it is in the larger Ruby community, and many open-source Rails applications are shipped with large RSpec test suites.

7.1.1 The RSpec Approach

To introduce the style of testing promoted by the RSpec library, we take the open-source Rails application Fat Free CRM[1] as a case study. Fat Free CRM is a customer relationship management system (CRM) designed to be tailored for different organizations. Development began in 2008. Fat Free CRM is released under the AGPL, has a codebase of roughly 23kloc, and is accompanied by a suite containing more than 1000 developer-written RSpec tests.

Figure 7-1 contains two examples of RSpec tests from the Fat Free CRM codebase. The first (lines 1-8) is intended to test that the `show` action correctly displays summaries of the site's users. What it actually tests is that a particular user (created by the call to the `Factory`) is displayed. The test begins with a natural-language specification of the feature under test (line 2); the body of the test (lines 3-7) is Ruby code written in the RSpec domain-specific language. This particular test constructs a mock object representing a single user (line 3), requests the summary page for that user (line 4), and then checks that the user that will be displayed matches the mock object just created (line 5). This check is written as an RSpec assertion, the general form of which is $a.should\ p\ b$, where a and b are objects and p is a predicate describing the desired relationship between them. The *assigns* variable is actually a Ruby hash populated by the Rails framework to contain the names and values of the instance variables set by the page request in line 4.

The second test (lines 10-17) is intended to check that private contacts are never displayed to users other than their owners. What it actually tests is that a particular private contact is hidden from a particular user. The test begins by building a mock object for the private contact owned by a mock user (line 12). The test then attempts to display the private contact (line 13). Since the mock user created as the owner of the contact in line 12 is by default distinct from the mock user representing the currently

[1] http://www.fatfreecrm.com/

```
describe UsersController do
  it "should expose the requested user as @user and render [show] template" do
    @user = Factory(:user)
    get :show, :id => @user.id
    assigns[:user].should == @user
    response.should render_template("users/show")
  end
end

describe ContactsController do
  it "should redirect to contact index if the contact is protected" do
    @private = Factory(:contact, :user => Factory(:user), :access => "Private")
    get :show, :id => @private.id
    flash[:warning].should_not == nil
    response.should redirect_to(contacts_path)
  end
end
```

Figure 7-1: RSpec Tests for Displaying Users and Restricting Access to Private Contacts

logged-in user, this request should fail. The test checks that it does indeed fail by asserting that the value of flash[:warning], which displays site errors, is populated (line 14), and that the user should be redirected to the index of contacts (line 15).

Each test in Figure 7-1 checks a desirable property—the property the programmer *intended* the program to have—for only a single case; these tests may easily miss corner cases that the programmer does not anticipate. For example, users in Fat Free CRM can be *suspended.* Fetching the page associated with a suspended user results in an error. The first test in Figure 7-1 succeeds despite this corner case because of the assumption, implicit in the use of a factory that creates non-suspended users by default, that the user being tested is not suspended. Unfortunately, this assumption is hidden from the programmer when factories are used, and it is difficult to anticipate the resulting corner cases.

7.1.2 Adding Quantifiers

Rubicon allows programmers to write *specifications* for web applications by extending the RSpec language with the quantifiers of first-order logic. Rather than testing just a single case for compliance with the application's specification, Rubicon performs symbolic execution for a truly arbitrary case, generates verification conditions that cover *all* possible cases, and uses an automatic bounded verifier to check all of them below a certain size.

Rubicon introduces two new methods on objects—*forall* and *exists*—which represent the universal and existential quantifiers of first-order logic. Both methods accept a single argument: a Ruby block, representing the property over which the quantifier

SpecFile	::	describe **<class> do** <Spec∗> **end**
Spec	::	it <string> **do** <Expr∗> **end**
Expr	::	<Ruby Expression>
	|	<object>.forall **do** |x| <Expr> **end**
	|	<object>.exists **do** |x| <Expr> **end**
	|	**<class>**.forall **do** |x| <Expr> **end**
	|	**<class>**.exists **do** |x| <Expr> **end**
	|	<Expr>.implies **do** <Expr> **end**
	|	<object>.should_equal <Expr>
	|	<object>.should_not_equal <Expr>
	|	<object>.should <Pred> <Expr>
	|	<object>.should_not <Pred> <Expr>
Pred	::	==
	|	be
	|	include
	|	raise
	|	throw
	|	respond_to
	|	have

Figure 7-2: Syntax of Rubicon Specifications

ranges; quantifiers succeed or fail in the same way as other RSpec tests.

Figure 7-2 specifies the core syntax of Rubicon specifications. Rubicon's syntax is exactly that of RSpec, with the addition of quantifiers. A set of Rubicon specifications begins with a *describe* block specifying the class being specified; each individual specification is written inside an *it* block with a string documenting that specification. A specification is simply a sequence of expressions, each of which may be a standard Ruby expression or a Rubicon assertion. A quantifier is also an assertion; *a.forall do* $|x|$ *b end* means that the assertions in b should hold for all possible values of x from the set a, and *a.exists do* $|x|$ *b end* means that the assertions in b should hold for at least one value of x from a.

Figure 7-3 presents Rubicon versions of the RSpec tests from Figure 7-1. This specification for displaying users (lines 1-9) quantifies over *all* users, rather than testing the intended property for just a single user. The block passed to the *forall* method is precisely the code we wrote in the original test—only the underlined code has changed, reflecting the introduction of quantified variables in place of mock objects.

The specification for restricting access to private contacts (lines 11-25) changes only slightly more. In order to check that permissions are respected no matter which user is logged in, we quantify over users (line 13) and then set the logged-in user correspondingly (line 15). To express the class of contacts for which the property should hold, we introduce a logical implication (lines 17-18) whose left-hand side requires that the contact's access should be set to private and that its owner should be distinct from the logged-in user.

```
describe UsersController do
  it "should expose the requested user as @user and render [show] template" do
    User.forall do |user|
      get :show, :id => user.id
      assigns[:user].should == user
      response.should render_template("users/show")
    end
  end
end

describe ContactsController do
  it "should redirect to contact index if the contact is protected" do
    User.forall do |user|
      Contact.forall do |private|
        set_current_user(user)
        get :show, :id => private.id
        ((private.access == "Private") &
         (private.user != user)).implies do
          flash[:warning].should_not == nil
          response.should redirect_to(contacts_path)
        end
      end
    end
  end
end
```

Figure 7-3: Rubicon Specifications for Displaying Users and Restricting Access to Private Contacts

7.1.3 The Power of Specification

By generalizing tests into specifications of application behavior, programmers can catch errors that tests alone are unlikely to find. Rubicon's automated analysis of formal specifications can explore corner cases, check for general regression errors, and build complicated object hierarchies for testing—areas in which standard RSpec tests fall short. Moreover, Rubicon allows the programmer to replace complicated code for constructing mock objects with simpler quantifiers.

The previously mentioned assumptions that accompany the use of factories to construct test data can easily lead to bugs that are difficult to find using testing alone. Because mock objects must take concrete values, a programmer will tend to construct mock objects that represent the most common situation. Fat Free CRM provides a good example: each contact has an associated list of *opportunities*, but the contact factory assumes that list to be empty. The original RSpec test therefore makes no mention of opportunities.

In writing specifications, on the other hand, the tendency is to provide as *little* information as possible, so as to have the highest chance of discovering corner cases that were not considered by the programmer.

The Rubicon specification in Figure 7-4 checks that permissions on all displayed elements are respected when displaying a contact. The specification quantifies over users, contacts, and opportunities (lines 3-5), sets the current user to the quantified one (line 6), and requests the contact display page (line 7). The specification then checks two properties: first, that for *any* contact, if that contact's permissions are set to private and its contact's owner is not the current user, then that contact should *not* be displayed (lines 9-12); and second, that for *any* opportunity, if that opportunity's permissions are set to private and its owner is not the current user, then that opportunity should *not* be included in the list of opportunities to be displayed (lines 14-17).

This specification makes no assumptions about any properties of the contact and opportunities in question; most importantly, it does not rule out the case that the requested contact is public, but one of the associated opportunities is private and not owned by the current user. Indeed, Rubicon catches this case immediately, returning the following counterexample for the second property:

```
⇒ u = User { :id ⇒ 1 ... }
   c = Contact { :access ⇒ 'public', :user ⇒ u,
                 : opportunities ⇒ [o] ... }
   o = Opportunity { :access ⇒ 'private',
                     :user ⇒ user1 ...}
```

This counterexample represents the case in which the current user owns the contact being displayed, but does *not* own the opportunity associated with that contact. Even though that opportunity is private, it will be displayed to the user.

The blame for this fault lies in the assumption, made in the show action of the contacts controller, that the permissions and ownership of a contact and its opportunities will be identical. The show method uses the my method of the Contact class to determine which contact to display; the developers of Fat Free CRM have redefined

```
describe ContactsController do
  it "should not display other users' private contacts or opportunities" do
    User. forall do |u|
      Contact. forall do |c|
        Opportunity. forall do |o|
          set_current_user(u)
          get :show, :id => c.id

          ((c.access == 'private') &
           (c.user != u)). implies do
            assigns [:contact]. should_not == c
          end

          ((o.access == 'private') &
           (o.user != u)). implies do
            assigns [:contact]. opportunities should_not
              include o
          end
        end
      end
    end
  end
end
```

Figure 7-4: Rubicon Specification for Displaying Contacts

```
describe ContactsController do
  it "should be able to associate newly created contact with the opportunity" do
    @opportunity = Factory(:opportunity, :id => 987);
    @contact = Factory.build(:contact)
    Contact.stub!(:new).and_return(@contact)

    xhr :post, :create, :contact => { :first_name => "Billy"}, :account => {},
        :opportunity => 987
    assigns(:contact).opportunities.should include(@opportunity)
    response.should render_template("contacts/create")
  end
end
```

Figure 7-5: RSpec Test for Contact Creation

the my method elsewhere so as to return only those records that the current user has permission to view. Referencing a particular contact's opportunities field, however, accesses the associated opportunities *directly*, bypassing the redefined my method, and *ignoring* the opportunities' permissions settings. If a user has permission to display a contact, then, he or she will be able to access all of its associated opportunities, regardless of their permission settings.

7.1.4 Exploiting the Bug

The corner case discovered in the previous section only qualifies as a serious security bug if it is possible to exploit it. The exploit described in the counterexample requires an invariant on the database to be broken—that the permissions of all the opportunities associated with a particular contact are compatible with the permissions on that contact. Rubicon can help us again, this time in checking whether or not it is possible to create a new contact that violates the invariant.

Invariants can be checked the same way using tests, but the contrived nature of test cases makes this strategy less useful than might be hoped—run-time assertions checking for invariant violations are therefore much more popular. Figure 7-5 contains the RSpec test written by the Fat Free CRM developers to check that the invariant is preserved. As usual, this test uses mock objects (lines 3-5) to construct the common case—one with the default permissions—creates a new contact (line 7), and tests that the opportunity is correctly associated with the contact (line 8).

The corresponding Rubicon specification (Figure 7-6), on the other hand, quantifies over all users, contacts, and opportunities (lines 3-5), constructs a new contact associated with the quantified opportunity (line 7), and checks that if the opportunity is private and not owned by the current user, then it should not be included in the resulting contact's set of associated opportunities (lines 9-11)—in other words, it should be impossible to create a contact that violates the invariant.

Once again, Rubicon's analysis yields a counterexample, informing us that it is indeed possible to violate the invariant:

```
describe ContactsController do
  it "should not associate another user's private opportunity with newly created
      contact" do
    User.forall do |user|
      Contact.forall do |contact|
        Opportunity.forall do |opportunity|
         set_current_user(user)
          xhr :post, :create, :contact ⇒ contact.attributes, :opportunity ⇒
              opportunity.id

          ((opportunity.user != user) &
           (opportunity.access=='private')).implies do
            assigns[:contact].opportunities.should_not include opportunity
          end
        end
      end
    end
  end
end
```

Figure 7-6: Rubicon Specification for Contact Creation

```
⇒ user = User { :id ⇒ 1 ... }
   contact = Contact { :opportunities ⇒
       [opportunity] ... }
   opportunity = Opportunity { :access
       ⇒ 'private', :user ⇒ user1 }
```

Investigating the code for the create action in the contacts controller, we found that the code that looks up the attached opportunity uses the find method, rather than the permission-enforcing my method, to find the opportunity whose ID is referenced in the HTML form submitted by the user.

Exploiting this security bug, then, is as easy as submitting a contact creation request with the ID of another user's opportunity in the "opportunity_id" HTML field. The developers of Fat Free CRM have acknowledged the bug, and are working on a fix.

7.2 Rubicon's Analysis

Rubicon is implemented as a library on top of Ruby, RSpec, and Rails. It extracts verification conditions from specifications by using the standard Ruby interpreter to perform symbolic, rather than concrete, execution; instead of concrete values, this style of execution produces abstract syntax trees representing the appropriate verification conditions. Rubicon then compiles the verification conditions into an Alloy [30], a lightweight specification language for software engineering, and performs

E(obj. forall b)	$\equiv$	$\forall x$. E(b. call (x))
E(obj. exists b)	$\equiv$	$\exists x$. E(b. call (x))
E(obj.should p e)	$\equiv$	E(obj.p(e))
E(obj.should_not p e)	$\equiv$	$\neg$E(obj.p(e))
E(e)	$\equiv$	RubyInterpreter (e)

Figure 7-7: Semantics of Rubicon Specifications; "b" is a Ruby Block, "p" a Predicate, "e" an Expression

bounded analysis using the Alloy Analyzer.

7.2.1 Rubicon's Semantics

Rubicon's semantics are intended to match those of Ruby precisely, and to combine Ruby's semantics in a natural way with the standard semantics of the quantifiers of first-order logic. Figure 7-7 contains an informal summary of Rubicon's semantics, using the standard quantifier symbols ($\forall$, $\exists$) and a function representing the semantics of the standard Ruby interpreter (RubyInterpreter). Given standard quantification over Ruby values, we can represent Rubicon's basic assertions (*should* and *should_not*) by simply invoking the predicate on the object in question; a true result means that the assertion holds. It is the goal of Rubicon's implementation to implement these semantics faithfully.

7.2.2 Rubicon's Implementation

To implement the semantics in Figure 7-7, Rubicon transforms specifications into verification conditions represented as abstract syntax trees and checks those conditions using a constraint solver. Figure 7-8 summarizes that transformation; the transformation is based on the use of symbolic objects defined by the Rubicon library to represent quantified variables.

To avoid re-implementing the Ruby interpreter, Rubicon implements the transformation from Figure 7-8 by persuading the standard Ruby interpreter to perform symbolic execution. Rubicon accomplishes this by defining symbolic objects in such a way that all method invocations on symbolic objects yield abstract syntax trees rather than values. Since specifications necessarily refer to symbolic objects if they use quantifiers, Rubicon can use the results of running code with symbolic objects to build abstract syntax trees representing verification conditions for the specification.

This strategy seems to work especially well for web applications. We offer two possible explanations based on our experience: first, the database schema of a web application specifies exactly the set of possible symbolic objects, limiting the size of that set; and second, web applications typically move much of the application's logic into the database, so the remaining code has few branches. Moreover, when specifications are partial (as they tend to be when generalized from tests), much of

C(obj. forall b)	$\equiv$	*all(*x*,* C(b. call (x))
C(obj. exists b)	$\equiv$	*some(*x*,* C(b. call (x))
		(x is a new symbolic object)
C(obj.should p e)	$\equiv$	*call(:should*, C(obj), C(p), C(e)*)*
C(obj. attribute)	$\equiv$	*field_ref(* C(obj), C(args)*)*
C(obj.meth(args))	$\equiv$	*call(* meth, C(obj), C(args)*)*
C(obj.where(e))	$\equiv$	*query(* v, obj, C(e)*)*
		(if obj is symbolic)
C(e)	$\equiv$	RubyInterpreter (e)
		(if e is concrete)

Figure 7-8: Compiling Specifications to Abstract Syntax Trees

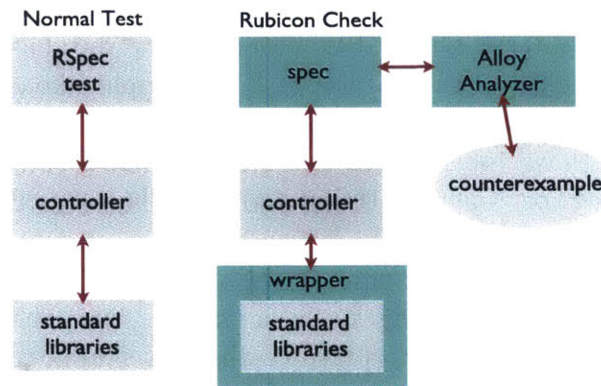


Figure 7-9: Comparison Between RSpec Execution and Rubicon Analysis

the execution takes place over concrete objects (e.g. the application's settings, or the page to be fetched) using the standard Ruby interpreter.

A diagram comparing Rubicon's execution model to that of RSpec is presented in Figure 7-9. Rubicon's analysis proceeds in three parts: first, the Rubicon library stubs the standard libraries and ActiveRecord objects; second, Rubicon runs the specification body in the standard Ruby interpreter, producing verification conditions; and finally, Rubicon compiles those verification conditions for the Alloy Analyzer to check.

7.2.3 Preprocessing: Stubbing Objects

Rails uses the ActiveRecord class as the basis for its object-relational mapper. In a standard Rails application, every object to be stored in the database is descended from ActiveRecord. To determine the set of classes that should be represented by symbolic objects, then, Rubicon builds a list of all the classes descended from ActiveRecord.

T(*all(x, e)*)	≡	**all** x: *typeOf(x)* | T(e)
T(*some(x, e)*)	≡	**some** x: *typeOf(x)* | T(e)
T(*field_ref(obj, f)*)	≡	T(obj).f
T(*call(:include, obj, a)*)	≡	T(obj) **in** T(a)
T(*call(:==, obj, a)*)	≡	T(obj) = T(a)
T(*query(name, type, e)*)	≡	{ name: type | T(e) }
T(*call(:should, obj, p, a)*)	≡	T(*call(p, obj, a)*)
T(*v*)	≡	v

Figure 7-10: Compiling Abstract Syntax Trees to Alloy Specifications

The next step is to *stub* those classes—replacing them with new classes that respond the same way to the operations defined on them, but with different behavior. Rubicon takes each ActiveRecord class and redefines two basic types of methods: class methods that query the database, and instance methods that allow the programmer to get and set the attributes (values to be stored in the database) of an object representing a particular record in the database. For example, the call User.all returns a list of all users, and given a user object, user.id returns that user's ID number.

Rubicon redefines these methods to produce abstract syntax trees rather than values: User.all returns Expr_Call(:all, User), and user.all returns Expr_Field_Ref(user, :id).

The classes that define abstract syntax trees are defined so that methods invoked on them produce new trees reflecting these operations, and the *should* method for building assertions is also redefined to produce abstract syntax trees. As a result, the following Ruby expression:

```
user1.id.should == user2.id
```

Evaluates to the following abstract syntax tree, instead of a value (Expr_Call represents the constructor of an abstract syntax tree node):

```
Expr_Call(:should, :==, Expr_Field_Ref(user1, :id), Expr_Field_Ref(user2, :id))
```

Having prepared the environment this way, running Rubicon specification code in the standard Ruby interpreter yields abstract syntax trees representing verification conditions.

7.2.4 Postprocessing: Producing Alloy

All that remains is to translate verification conditions into Alloy specifications. Since our abstract syntax trees are designed for this purpose, doing so is straightforward. We summarize the translation in Figure 7-10.

The Alloy language includes all of first-order relational logic, plus transitive closure. Quantifiers, therefore, are translated into their analogues in Alloy; field references become relational joins based on Alloy's global relations, database queries become Alloy set comprehensions, and Rubicon assertions turn into logical formulas.

Alloy's universe is made up of uninterpreted atoms, which are divided into sets based on user-defined *signatures.* In addition to designating a set of atoms, signatures may define global *relations*; Alloy specifications are typically based on these global relations, and logical formulas are built from the results of relational joins or membership tests over these relations.

Rubicon uses Alloy's atoms to represent objects. It defines a signature for each type of object that is mentioned in a given specification, with relations to represent the values of the object's fields. Given this representation, a field reference in Ruby is equivalent to a relational join in Alloy. This paradigm is popular in Alloy, so the syntax of relational join is designed to make the representation of a field reference appear similar to the typical syntax in programming languages—*obj.f* means the relational join of the atom *obj* with the global relation *f*, but it also represents the reference to field *f* of object *obj*, given our encoding.

To generate a complete Alloy specification, Rubicon constructs a signature definition for every class referenced in the corresponding Rubicon specification. For each signature, Rubicon consults the application's database schema (encoded in the ActiveRecord) to determine the set of attributes associated with that type, and adds a field definition to the signature for each attribute. For example, the first part of the signature definition for the *User* class from Fat Free CRM is as follows:

```
sig User {
  name: String,
  created_at: Datetime,
  updated_at: Datetime,
  email: String,
   ...
}
```

Rubicon combines these signature definitions with the translated verification conditions and invokes the Alloy Analyzer to determine whether or not the specification is satisfied. The Alloy Analyzer is a tool for automatic bounded analysis of Alloy specifications; it places finite bounds on the number of each type of atom present in the universe, then reduces the specification to propositional logic. Alloy's model finder, Kodkod [50], handles the translation from relational logic to a satisfiability problem using algorithms that optimize relational expressions. This makes Kodkod, and thus Alloy, especially suited to database applications. If a counterexample is found, Kodkod maps the SAT result back to a valuation for the original specification's relations.

Rubicon, in turn, maps the counterexample Alloy produces back to a Ruby object structure, which forms the printed counterexample the user sees when a specification is found not to hold.

7.2.5 An Example: Contact Permissions

As a complete example of Rubicon's analysis, consider the example of checking that the user has permission to view a contact and its associated opportunities (Figure 7-

```
def show
  @contact = Contact.my.find(params[:id])
  @stage = Setting. unroll (:opportunity_stage)
  @comment = Comment.new
  ...
end

User. forall do |u|
  Contact. forall do |c|
    Opportunity. forall do |o|
      set_current_user(u)
      get :show, :id => c.id

      ((o.access == 'private') &
       (o.user != u)). implies do
        assigns [: contact ]. opportunities should_not
          include o
      end
    end
  end
end
```

⇒

```
Expr_Implies(
  Expr_And(
    Expr_Call(:==, Expr_Field_Ref(o, :access),
                  ' private '),
    Expr_Call(:!=, Expr_Field_Ref(o, :user),
                  u)),
  Expr_Not(
    Expr_Call(:include ,
              Expr_Field_Ref(c, : opportunities ),
              o)))
```

⇒

```
one sig string_private extends String {}
sig Opportunity{
  id: ID, access: String, ...,  user: set User
}
sig Contact{
  id: ID, access: String, ...,
  opportunities: set Opportunity
}
sig User{
  id: ID, ...
}

check {
  all u: User |
    all c: Contact |
      all o: Opportunity |
        (o.access = string_private and
         o.user != u) implies
          !(o in c.opportunities)

} for 5
```

Figure 7-11: Verification Condition and Corresponding Alloy Specification

4). Figure 7-11 demonstrates how the second of these two properties is checked.

First, during execution by the Ruby interpreter, the property, along with the Fat Free CRM implementation, produce a verification condition to be checked by the Alloy Analyzer. The page request (line 12) passes the quantified contact's ID to the `show` method, where the call to the stubbed version of `Contact.my.find` (line 2) yields a symbolic record representing precisely the quantified contact. The method returns, having set `assigns[:contact]` to that symbolic record.

The two conditions on the left-hand side of the implication (line 15) evaluate to expressions involving the quantified opportunity; the condition on the right-hand side of that implication (line 16) evaluates to an expression involving both the symbolic record constructed above and the quantified opportunity. The expression representing the complete verification condition is listed in lines 22-31. While Rubicon cannot yet handle constraints involving string manipulations, it does allow testing string equality, as in lines 24-25.

Second, the Rubicon postprocessor produces an Alloy specification equivalent to the verification condition. Each expression type produced by the methods Rubicon stubs corresponds directly to an Alloy expression. Rubicon wraps the verification condition produced above with the appropriate quantifiers, combines it with a set of signatures corresponding to the classes used in the specification, and produces the Alloy specification listed in lines 32-51.

7.3 Evaluation

To confirm that Rubicon's analysis is capable of scaling to real applications, we tested it on five open-source Rails applications whose distributions contain RSpec tests written by the original developers. We tried to select applications that perform a variety of tasks and that have a sizable user base. All of the applications we examined contain extensive RSpec test suites with documentation. The five applications we examined are:

- **Insoshi**, a social networking platform
- **Fat Free CRM**, a customer relationship management platform
- **RubyTime**, a time-management system
- **RubyURL**, a URL-shortening service
- **Tracks**, an application to implement the "Getting Things Done" methodology

7.3.1 Methodology

For each application, we selected a random subset of the files containing developer-written RSpec tests for the application's controllers. We included at least three test files, or all of them if the application provided fewer than three. The controller tests tend to express properties of the application's behavior, rather than properties about

Filename	Number of Tests	Average RSpec Time per Test	Average Rubicon Time per Test	Original RSpec Lines of Code	Rubicon Lines of Code
Insoshi (12k LOC)					
people_controller_spec.rb	27	0.41s	1.52s	272	314
topics_controller_spec.rb	4	0.52s	1.86s	42	57
comments_controller_spec.rb	14	0.38s	2.08s	126	143
Totals	**45**	**0.44s**	**1.82s**	**440**	**514**
Fat Free CRM (23k LOC)					
home_controller_spec.rb	8	0.64s	2.45s	98	115
comments_controller_spec.rb	21	0.53s	2.12s	254	301
users_controller_spec.rb	27	0.42s	1.87s	343	386
contacts_controller_spec.rb	53	0.44s	1.97s	696	754
Totals	**109**	**0.51s**	**2.10s**	**1391**	**1556**
RubyTime (11k LOC)					
users_spec.rb	31	0.32s	2.11s	271	294
clients_spec.rb	11	0.28s	2.54s	104	132
Totals	**42**	**0.30s**	**2.35s**	**375**	**426**
RubyURL (1k LOC)					
links_controller_spec.rb	8	0.36s	2.63s	73	94
Totals	**8**	**0.36s**	**2.63s**	**73**	**94**
Tracks (22k LOC)					
todo_spec.rb	20	0.27s	2.16s	182	207
user_spec.rb	26	0.23s	2.46s	174	197
Totals	**46**	**0.25s**	**2.31s**	**356**	**404**

Figure 7-12: Case-Study Summary: the Number of Tests, Average Runtime of Original Test and Corresponding Rubicon Specification, and Lines-of-Code Comparison Between Original Tests and Rubicon Specifications

the database schema, so we considered them both a better test of Rubicon's ability to check an application's behavior and more likely to find behavioral bugs in the application.

We converted all of the tests from each selected file into Rubicon specifications by replacing mock objects with quantifiers over objects. This process was straightforward for most tests, since the natural language description of each test combined with the Ruby code implementing it generally described the intent of the test. The conversion process took fewer than ten hours over the course of a month.

We ran both the original RSpec tests and our Rubicon specifications on an Intel Core 2 Duo E7500 with 4GB RAM under Ubuntu 10.04 and Ruby 1.8.7, with the latest version of each application (when available, the development version). Rubicon makes use of version 4.1 of the Alloy Analyzer.

By default, Rubicon uses a finite bound of five during analysis, meaning that the corresponding Alloy Analyzer analysis searches for a counterexample in universes containing five objects or fewer per Ruby class. We used this bound in our experiments. Specifications from this evaluation are available on the Rubicon webpage.

7.3.2 Results

We converted a total of 250 developer-written RSpec tests into Rubicon specifications. The table in Figure 7-12 contains a summary of the size of each application, the filenames of the test files we converted, the number of tests per file, the average

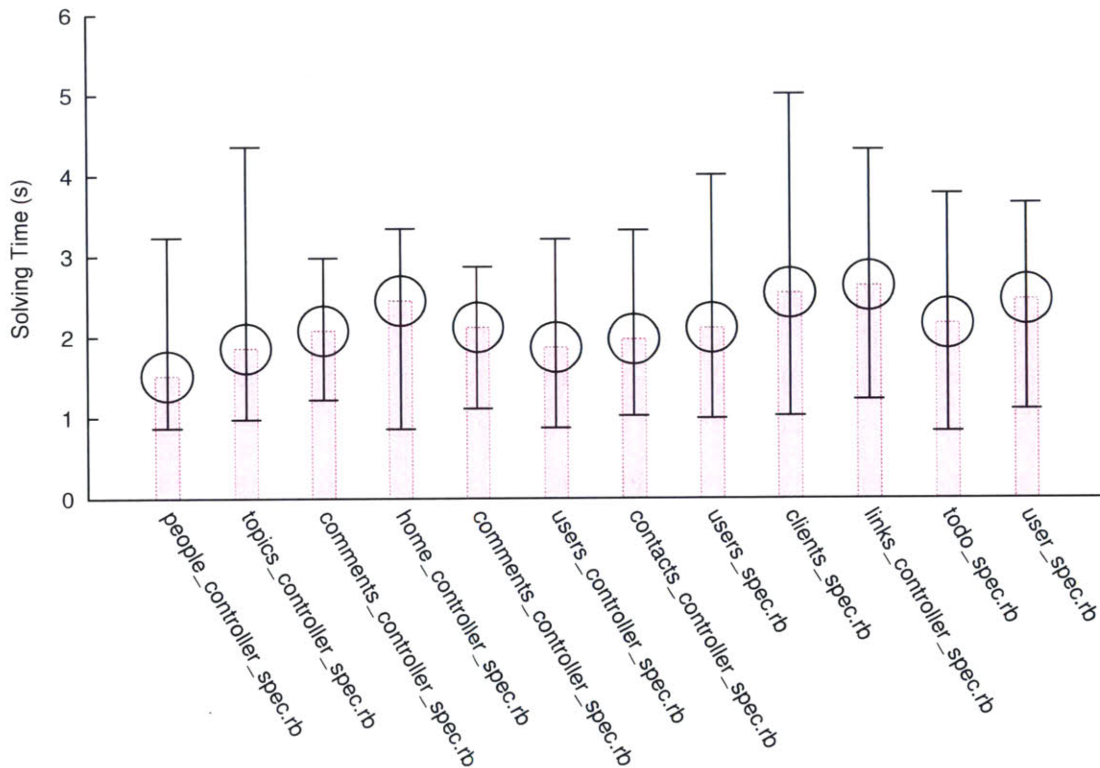


Figure 7-13: Range of Analysis Times for Fat Free CRM Rubicon Specifications: Average, Maximum, and Minimum Analysis Times for Each Test File

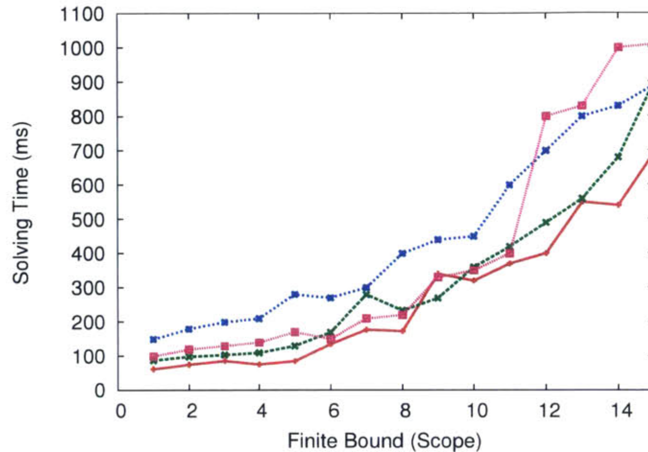


Figure 7-14: Effect of Finite Bound on Solving Time of Verification Conditions from Examples in Figures 3, 4, and 6

execution time of the original tests and their associated Rubicon specifications, and a comparison between the number of lines of code used before and after the conversion to Rubicon.

The results show that while analyzing Rubicon specifications is several times slower, on average, than executing the original RSpec tests, Rubicon's analysis usually takes only a second or two. Moreover, converting the original RSpec tests into Rubicon specifications made the source code only 11% longer, suggesting that writing Rubicon specifications may not be significantly more difficult than writing RSpec tests.

Figure 7-13 is a visual representation of the range of per-specification analysis times for each of the test files in Figure 7-12. For each test file, the bar represents the average analysis time over all of that file's specifications; the extent of the error bars represents the maximum and minimum analysis time for any specification in that file. The maximum analysis time for any test we converted was just over five seconds; more importantly, the maximum and minimum analysis times were always close to the average, suggesting that Rubicon's analysis is consistently fast.

To evaluate the effect of the finite bound on Rubicon's analysis time, we generated the Alloy specifications corresponding to each of the four Rubicon specifications listed in this paper (in Figures 3, 4, and 6) and manually tested the differences in solving times at different scopes using the Alloy Analyzer. The results of this experiment are displayed in Figure 7-14. As is common with SAT-based analyses, the solving time begins to increase exponentially as the finite bound rises above ten atoms per class. Rubicon's default bound of five is a compromise attempting to produce consistently fast analysis times without missing bugs.

7.3.3 Fat Free CRM Bug

Of the RSpec tests we converted, only one led to the discovery of a previously unknown bug. As discussed in Sections 2.2 and 2.3, testing the permissions of both contacts and their associated opportunities, along with the ability of bounded analysis to explore corner cases, allowed us to discover a situation in which a user can view another user's private opportunities.

Several other tests we converted led us to the discovery (also discussed in Section 2.3) that the Fat Free CRM developers had redefined the ActiveRecord `my` method to enforce permissions. Rubicon also redefines this method, and interprets the resulting verification condition as if its use has the semantics originally intended by the Rails developers. The result was a false positive: Rubicon originally reported that any user could view any entity in the system, regardless of permissions. We worked around this problem by renaming the method defined by Fat Free CRM and calling it directly. This was the only Fat Free CRM issue that caused false positives to be reported by Rubicon.

7.4 Related Work

Rubicon's specification language is based on RSpec, an embedded domain-specific language for testing [16]. Our approach of embedding an expressive specification language directly in a programming language is most heavily influenced by QuickCheck [19], a random testing framework for Haskell in which the programmer specifies properties by writing code. Parameterized unit testing [48] takes a similar approach, allowing the programmer to write specifications as tests parameterized by their inputs.

Existing work on the application of formal methods to web applications focuses on modeling applications, and especially on building navigation models. Bordbar and Anastasakis [8], for example, model a user's interaction with a web application using UML, and perform bounded verification of properties of that interaction by translating the UML model into Alloy using UML2Alloy; other approaches ([35, 49, 41]) perform similar tasks but provide less automation. Nijjar and Bultan [38] translate Rails data models into Alloy to find inconsistencies.

Most techniques focus specifically on navigation from one page to another, and can analyze only those properties related to possible navigations. Some existing work ([2, 13]) models possible navigations using UML, others ([6]) using directed graphs, and still others ([24, 17, 52]) using statecharts.

Those techniques that allow programmers to specify an application's behavior are closest in their aim to Rubicon. Syriani and Mansour [46] use SDL (the Specification and Description Language) to model some aspects of application, and provide automatic test case generation based on the model. Haydar et al. [29] use communicating automata to build the application's model, and have explored techniques for verifying properties of those automata. Finally, Andrews et al. [3] use finite state machines to model applications, and also generate test cases. All of these techniques require the programmer to build a separate model of their web application's behavior, and

limitations of the modeling technique used mean the set of properties that can be checked is also limited. Rubicon, in contrast, is capable of checking any property expressible in first-order logic.

Chapter 8

Conclusion

This thesis has proposed new techniques for finding and eliminating application-specific bugs in web applications. We have demonstrated three approaches to finding these bugs; all three are powered by a scalable symbolic execution specifically tailored to the structure of web application implementations, allowing analysis of even the largest real-world applications.

In contrast to existing general-purpose verification approaches, this work was inspired by the hypothesis that narrowing our focus might produce more effective tools. Our approach has been to take advantage of properties specific to application-specific security bugs in web applications in order to produce these tools. The results suggest that focusing on a particular class of applications (web applications) and on a particular class of bugs (missing security checks) we can build static analysis tools that are better at finding bugs than general-purpose tools.

8.1 Discussion

This thesis makes two key contributions. First, we explored how taking advantage of properties of a target application can make static analysis easier. Second, we explored how properties of a class of bugs can lead to better techniques for finding those bugs.

We took advantage of Ruby's flexibility to build a symbolic evaluator as a library, and use the standard Ruby interpreter to perform symbolic execution. This strategy ensures compatibility with Rails, since the standard interpreter is used. The symbolic evaluation library defines a class of symbolic values and overrides the operators of the standard library to compute over them. The resulting symbolic execution library is less than 1000 lines of code long, but can analyze even the largest Rails applications in just minutes.

We used the fact that programmers intend security policies to be uniformly applied to build Derailer, a bug finding tool based on exploring the data exposures allowed by the application. To find missing security checks, the developer and the tool together infer a specification of the application's security policy, and the tool finds exposures that do not obey the policy. When the developer is finished with this iterative process, the selected constraints represent a specification of the security

policy, and the remaining highlighted exposures are security bugs. This approach has been very successful in finding bugs, mainly because Derailer does not require the time-consuming step of writing a specification.

We automated this process even further by taking advantage of the fact that many applications share common patterns of access control. Based on this observation, we hypothesized that a small set of formal models—a catalog of access control patterns—is enough to capture a large portion of the access control policies that web applications use in practice. SPACE leverages this idea to find security bugs by checking every data exposure allowed by the application is also allowed by some pattern in the catalog. This approach produces few false positives and found a significant number of previously unknown bugs; it also re-discovers the bugs we found using Derailer, without requiring Derailer's interactive step.

Finally, Rubicon brings together the ideas of testing, formal specification, and bounded analysis, and applies them to web applications. Its combination of an existing testing framework and first-order quantifiers is powerful enough to express rich behavioral specifications without requiring programmers to learn a new specification language. Rubicon allows programmers to specify and check functional properties of web applications, including full-functional verification, and scales to large applications thanks to its use of our symbolic execution framework. Our anecdotal experience with security bugs suggests that most classes of bugs can be more easily detected with more specialized tools, but a broadly-applicable technique such as Rubicon is still useful when no specific tool exists for a target domain or when full-functional verification is desired.

In theoretical terms, Rubicon is the most powerful of these tools. It is capable of detecting all violations of the specifications it supports, including both security-related properties and others. Derailer is designed to detect a subset of these bugs: it is limited to security bugs, and only finds mistakes in the uniform application of policies. Derailer may therefore miss bugs where a security check is missing from every single access of a data type. It also requires that the user not mis-read or mis-classify any constraint; either mistake could cause an incorrect security policy to be applied. SPACE finds an even smaller class of bugs: only those that violate the policies built into the tool. In contrast to Derailer, SPACE *can* detect situations in which an entire data type is missing a check, and does not rely on the user to classify exposures; however, SPACE produces false positives when the target application deviates from one of the built-in patterns.

In our experience, Rubicon is effective at detecting the kinds of bugs it was designed to find. Its specifications are more general than test cases, and so catch bugs lurking in corner cases. However, it can still be difficult to write the *right* specification, and it is possible to miss bugs by forgetting to specify some behavior. For example, in one specification for Fat Free CRM, we simply forgot to specify that authentication was required, and missed a bug because of it. We later detected this bug using Derailer.

Our experience suggests that Derailer's visual representation of an application's behavior can be especially useful for users unfamiliar with the target application, and that its uniformity-based analysis is good at zeroing in on potential bugs quickly.

This makes Derailer potentially even better than SPACE for tasks like security audits or grading student homeworks. However, we have also experienced missed bugs with Derailer: one student application used the constraint "current_user.id = params[:id]," which *looked* reasonable, and which was applied uniformly, but which actually represented the wrong security policy. And in MediumClone, an entire controller (the Users controller) is missing authentication checks—again, a situation in which the *wrong* policy was applied uniformly.

While SPACE is intended to be more automated than Derailer, our experience suggests that it can be useful to use Derailer to learn more about an application before using SPACE, since this information helps in writing the mapping that SPACE requires. Once this mapping is written, however, SPACE does catch bugs that Derailer misses (it found both of the bugs described above that we missed when using Derailer). While we have not experienced it, there is also the potential to mis-classify a counterexample as a false positive when it is actually a bug. Thus the best use of SPACE may in fact be in tandem with Derailer, with each tool helping to cover the blind spots of the other.

While our experience suggests that tools with increased automation, such as Derailer and SPACE, can provide more effective interfaces for finding security bugs, these anecdotes are not strong evidence that our solutions will be more usable for users than existing approaches. Our tools could likely be improved significantly using the results of usability studies asking users to put them into practice.

However, our results suggest broader lessons on static analysis tools for improving program reliability. First, taking advantage of properties unique to a particular domain, as we did in our symbolic execution framework, can significantly improve scalability. Second, taking advantage of properties of a particular class of bugs allows formal specifications to be replaced by automated analyses or interactive tools, as in Derailer and SPACE.

8.2 Future Directions

The scalability of our symbolic execution framework comes from properties unique to the domain of web applications. Similarly, the effectiveness of our bug-finding techniques is based on more domain properties: the uniformity and commonalities of security policies. The success of these projects suggests both further improvements in the domain of web applications, and that similar improvements are possible in other domains.

8.2.1 Cyber-physical Systems

The same domain-specific approach to bug finding could be used to ensure both security and reliability for the new generation of internet-connected cyber-physical systems. Cyber-physical systems have similar structural properties to web applications, suggesting that the same kind of analysis techniques might apply. In particular, general-purpose verification techniques struggle to deal with the environmental con-

cerns inherent in cyber-physical systems; a domain-specific approach can tackle this issue head-on.

We have used our approach to perform a case study in building a dependability case for the proton therapy system at Massachusetts General Hospital [?], which provides radiation treatment to cancer patients. The safety requirements of this system involve statements about its environment, so verification requires building a formal model of the environment's properties. This is a difficult task, even for an expert, since the environment is complex; and if the environmental model is incomplete, the system may be "verified correct" even if it has a bug.

We found that it was sufficient to check only the environmental properties *on which the software actually depended*, eliminating the need for a complete model of the environment. We used static analysis to enumerate the calls to libraries used for environmental interaction, and produced a short list of conditions for a domain expert to check.

The same kind of environment-focused analysis could be used to find security flaws in other devices—like medical devices, cars, home automation products, and computer-controlled electric and water systems—whose security policies are intertwined with environmental concerns.

8.2.2 Access Control and APIs

Libraries for enforcing access-control policies are increasingly popular, since they represent a significant improvement over *ad hoc* security checks, but these libraries still require programmers to use them correctly. Derailer's model of analysis, interaction, and automation is also suitable for detecting inconsistent API use within an application; the techniques we have developed could be extended to detect these mistakes. In fact, these techniques might even be applicable to other kinds of APIs, since they, too, expect consistent use.

The idea of consistency is central to most of the properties leveraged in the domain-specific static analyses I have developed. In security, a consistency-based model of access control might be more broadly applied than my existing web-centric model. Such a model might be used to find new kinds of security bugs in web applications, and also security bugs in other kinds of applications. This model further increases the level of automation available to the developer—it can be used, for example, to automatically flag behaviors that are not defined in the model as likely bugs, as in our work on security patterns.

8.2.3 Statistical Models of Behavior

If two exposures of the same data have inconsistent constraints, one of them represents a bug. But without information about the security policy, it is impossible to determine which one is buggy. Automatic anomaly detection techniques consider a large majority of consistent examples to represent a specification, and highlight examples outside of that set. Since web applications contain relatively few exposures, however, these techniques do not apply.

The set of *all* web applications, on the other hand, contains a huge number of exposures. This aggregate set of exposures represents a *library of specifications for different kinds of applications.* Individual applications may contain bugs, but the "average" of all implementations of a particular piece of functionality represents a correct specification of that functionality. And as the amount of publicly available application code grows, so does the size of the library. We have already tested Derailer's scalability by running its analysis on more than 1000 open-source applications from Github; using Derailer's analysis to compile a database of the data exposures from every Rails application on Github would take only a couple months of compute time.

We hope to explore the use of our symbolic execution framework to build clusters of applications according to aspects of their functionality. Analysis results typical of a cluster would be considered a specification of that cluster, and would be compared against the analysis results of the target application. Security checks typical of the cluster but missing from the target represent likely bugs. A tool based on this approach may produce some false positives, but would enumerate bugs in a target application automatically.

Bibliography

[1] Devdatta Akhawe, Adam Barth, Peifung E Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304. IEEE, 2010.

[2] L. Alfaro. Model checking the world wide web. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 337–349. Springer-Verlag, 2001.

[3] A.A. Andrews, J. Offutt, and R.T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4(3):326–345, 2005.

[4] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272. ACM, 2008.

[5] T.H. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 921–938. ACM, 2011.

[6] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference (WWW'2002)*. Citeseer, 2002.

[7] Ivan Bocić and Tevfik Bultan. Inductive verification of data model invariants for web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 620–631. ACM, 2014.

[8] B. Bordbar and K. Anastasakis. Mda and analysis of web applications. *Trends in Enterprise Application Architecture*, pages 44–55, 2006.

[9] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 239–254. ACM, 2014.

[10] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. Guardrails: a data-centric web application security framework. In *Proceedings of the 2nd USENIX*

conference on Web application development, pages 1–1. USENIX Association, 2011.

[11] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224. USENIX Association, 2008.

[12] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, et al. A nice way to test openflow applications. In *NSDI*, volume 12, pages 127–140, 2012.

[13] D. Castelluccia, M. Mongiello, M. Ruta, and R. Totaro. Waver: A model checking-based tool to verify web application design. *Electronic Notes in Theoretical Computer Science*, 157(1):61–76, 2006.

[14] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[15] A. Chaudhuri and J.S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 585–594. ACM, 2010.

[16] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North. The rspec book: Behaviour driven development with rspec, cucumber, and friends. *Pragmatic Bookshelf*, 2010.

[17] J. Chen and X. Zhao. Formal models for web navigations with session control and browser cache. *Formal Methods and Software Engineering*, pages 46–60, 2004.

[18] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, page 1. USENIX Association, 2010.

[19] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 35(9):268–279, 2000.

[20] L.A. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222, 1976.

[21] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):10, 2015.

[22] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*. USENIX Association, 2011.

[23] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th conference on USENIX security symposium*, pages 267–282. USENIX Association, 2009.

[24] L. De Alfaro, T.A. Henzinger, and F.Y.C. Mang. Mcweb: A model-checking tool for web site debugging. In *Poster presented at WWW*, volume 10. Citeseer, 2001.

[25] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[26] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[27] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.

[28] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[29] M. Haydar, A. Petrenko, and H. Sahraoui. Formal verification of web applications modeled by communicating automata. *Formal Techniques for Networked and Distributed Systems-FORTE 2004*, pages 115–132, 2004.

[30] D. Jackson. *Software Abstractions: logic, language, and analysis.* The MIT Press, 2006.

[31] S. Khurshid, C. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[32] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 199–209. IEEE, 2009.

[33] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[34] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In John Field and Michael Hicks, editors, *POPL*, pages 151–164. ACM, 2012.

[35] DR Licata and S. Krishnamurthi. Verifying interactive web programs. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 164–173. IEEE.

[36] A.C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.

[37] Joseph P Near and Daniel Jackson. Derailer: interactive security analysis for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 587–598. ACM, 2014.

[38] Jaideep Nijjar and Tevfik Bultan. Bounded verification of ruby on rails data models. In Matthew B. Dwyer and Frank Tip, editors, *ISSTA*, pages 67–77. ACM, 2011.

[39] C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. *Model Checking Software*, pages 164–181, 2004.

[40] B.C. Pierce. *Types and programming languages.* The MIT Press, 2002.

[41] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer, editors, *ICSE*, pages 25–34. IEEE Computer Society, 2001.

[42] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[43] Samir Sapra, Marius Minea, Sagar Chaki, Arie Gurfinkel, and Edmund M Clarke. Finding errors in python programs using dynamic symbolic execution. In *Testing Software and Systems*, pages 283–289. Springer, 2013.

[44] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[45] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[46] J. Syriani and N. Mansour. Modeling web systems using sdl. *Computer and Information Sciences-ISCIS 2003*, pages 1019–1026, 2003.

[47] Soon Tee Teoh, Kwan-Liu Ma, Soon Felix Wu, and T.J. Jankun-Kelly. Detecting flaws and intruders with visual data analysis. *IEEE Computer Graphics and Applications*, 24(5):27–35, 2004.

[48] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, September 2005.

[49] P. Tonella and F. Ricca. Dynamic model extraction and statistical analysis of web applications. 2002.

[50] E. Torlak and D. Jackson. Kodkod: A relational model finder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

[51] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 54. ACM, 2014.

[52] M. Winckler and P. Palanque. Statewebcharts: A formal description technique dedicated to navigation modelling of web applications. *Interactive Systems. Design, Specification, and Verification*, pages 279–288, 2003.

[53] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. pages 85–96, 2012.

[54] A. Yip, X. Wang, N. Zeldovich, and M.F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304. ACM, 2009.