# Derailer: Interactive Security Analysis for Web Applications

Joseph P. Near, Daniel Jackson
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA
jnear@csail.mit.edu, dnj@mit.edu

## ABSTRACT

Derailer is an interactive tool for finding security bugs in web applications. Using symbolic execution, it enumerates the ways in which application data might be exposed. The user is asked to examine these exposures and classify the conditions under which they occur as security-related or not; in so doing, the user effectively constructs a specification of the application's security policy. The tool then highlights exposures *missing* security checks, which tend to be security bugs.

We have tested Derailer's scalability on several large open-source Ruby on Rails applications. We have also applied it to a large number of student projects (designed with different security policies in mind), exposing a variety of security bugs that eluded human reviewers.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

## Keywords

Web applications; security; static analysis.

## 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.

We propose a solution that avoids the need for new frameworks or specifications. Instead, it uses a combination of symbolic evaluation and user interaction to help the programmer discover mistakes.


*ASE'14,* September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2643012.

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 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 have built a prototype tool implementing our proposed approach. Called *Derailer*[1], it performs symbolic execution of a Ruby on Rails application to produce a set of exposures. Derailer's symbolic execution is based on Rubicon [17], a tool we previously built for specifying and checking security properties of Rails applications. Rubicon was effective at checking security policies, but our experience suggested that completely specifying a security policy is difficult. Derailer is therefore motivated by the desire to find security problems without requiring an explicit written specification.

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

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.

The contributions of this paper include:

- A lightweight scheme for detecting security vulnerabilities in web applications, which involves (1) constructing an abstraction of an application's behavior that indexes the constraints under which resources can be accessed by access path and action, (2) asking a user to identify which constraints are security related, and (3) displaying discrepancies that show cases in which the same resource is guarded inconsistently across different actions.
- Derailer, an implementation of this analysis for Ruby on Rails applications with an interactive interface for interpreting its results.
- Two case studies, one providing evidence that Derailer scales to real-world applications, and the other indicating that Derailer is helpful in finding security bugs.
- A lightweight static analysis, based on our previous work, that enumerates the ways in which data can flow from the database to rendered pages in a web application.

Section 2 introduces our approach by describing the use of Derailer to discover a security bug in an example application. Section 3 formalizes our analysis, while Section 4 contains details of Derailer's implementation. Section 5 contains our case studies evaluating Derailer's scalability and effectiveness, and Section 6 discusses related work. Finally, Section 7 relates some conclusions.

## 2. USING DERAILER

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 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 11). The application's programmer assumed that users would follow links from the index page—which *does* correctly enforce access control—to

```
1   def index
2     @notes = Note.where(user: current_user.id) +
            Note.permissions.find_by_user_id(current_user.id)
3
4     respond_to do |format|
5       format.html # index.html.erb
6       format.json { render json: @notes }
7     end
8   end
9
10  def show
11    @note = Note.find(params[:id])
12
13    respond_to do |format|
14      format.html # show.html.erb
15      format.json { render json: @note }
16    end
17  end
```

**Figure 1: Example Controller Code from Student Project**

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.

Figure 2 contains 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.

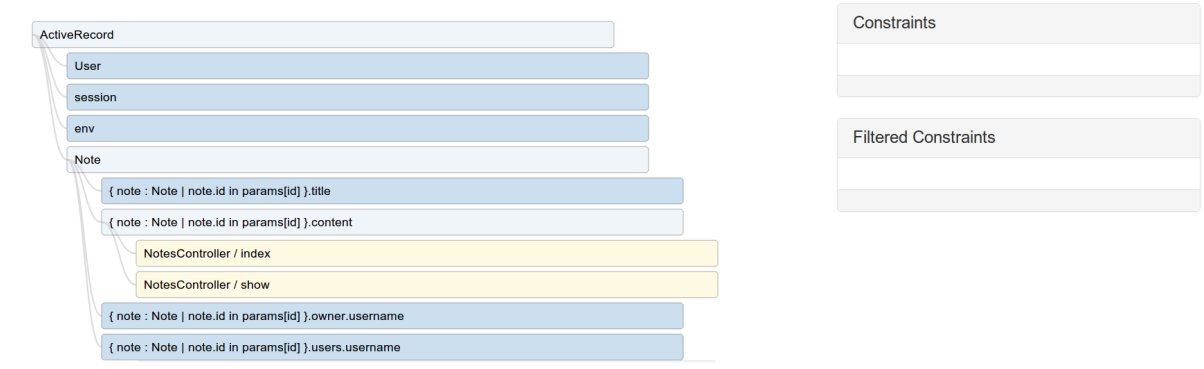
Then, in shot (2), the user picks the "index" action from the list of actions resulting in that exposure. When an action is selected, the set of constraints governing the release of that data by that action is displayed in the Constraints area. In this case, the displayed constraint is "`note.user ==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

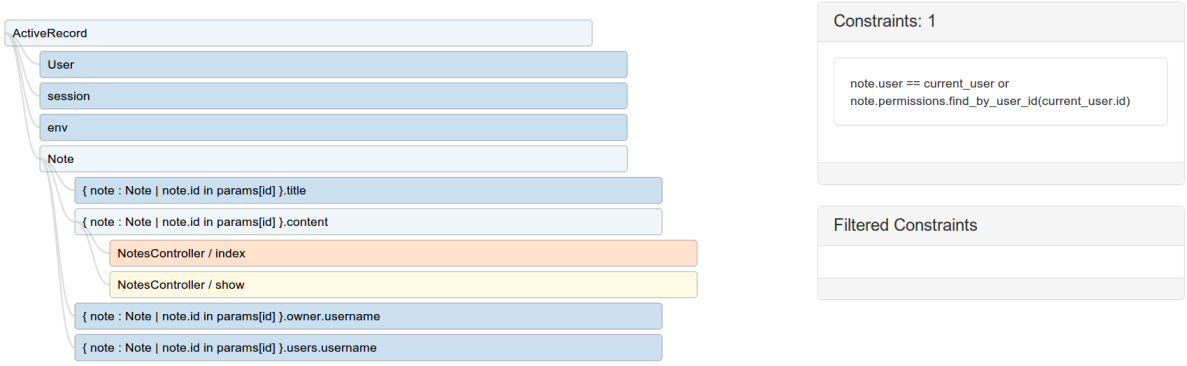
We took the approach described in this section in using Derailer to analyze 127 similar student projects from the same course, and found bugs in nearly half of them. The results of that experiment are described in Section 5.

## 3. MODEL

This section explains the fundamental model that underlies our approach. The model is important for several reasons. First, understanding the model is a prerequisite to




**(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.


**(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 2: Example Bug-finding Session using Derailer**

understanding the results the tool produces. Second, the model is the basis of our implementation strategy, and informs the assumptions made in our implementation. Third, the model makes clear which aspects of web applications make this problem more tractable than it would be for arbitrary applications.

Clients issue requests that contain a binding of parameters to values, and a choice of action[2]:

```
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
}
```

[2]The model is given in Alloy [11]. 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.

(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
}
```

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.

## 3.1 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.

## 3.2 Filtering

Our approach uses interaction with the user to build the desired security policy and discover exposures that do not obey it. We allow the user to choose a set of constraints representing a candidate security policy and *filter* the set of exposures based on those constraints. The result is exactly the set of exposures *missing* one of the filtered constraints—in other words, exposures for which a security check is missing.

```
fun filter [es: set Exposure,
            cs: set Constraint]: set Exposure {
  {e': Exposure |
    some c: cs | c !in e'.constraints}
}
```

Our implementation satisfies this specification by simply comparing each exposure's constraints against the filtered set, and displaying those exposures missing a constraint from that set.

# 4. IMPLEMENTATION

Derailer is implemented as a library for Ruby. Derailer enumerates the application's actions, runs each one symbolically to obtain a set of exposures, and organizes the exposures for presentation to the user.

Rather than implement a standalone symbolic evaluator—a difficult task for a large, under-specified, dynamic environment like Rails—we hijacked the existing Ruby runtime to do symbolic execution. Derailer uses Ruby's metaprogramming features to wrap the standard libraries of Ruby and Rails in a thin layer that allows the existing code to compute with symbolic values.

We pioneered this technique in our previous work on Rubicon [17], which performed bounded verification of user-defined properties on Rails applications. A more detailed description of our symbolic evaluator is available as part of that work; a formal description of our technique, including an embedding in a simplified Ruby-like language with a formal semantics and a proof that our technique produces the same results as the standard approach to symbolic execution, is available in [18].

For completeness, we present a brief explanation of our approach to symbolic execution, with a focus on the strategy Derailer uses to explore all possible page requests and extract the resulting exposures.

## 4.1 Symbolic Values

To add symbolic values to Ruby, we define a class of symbolic objects to represent the paths and constraints of Section 3.

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

class Exp < SymbolicObject
  def initialize (meth, args)
    @meth = meth
    @args = args
  end
end
```

These objects use Ruby's method_missing feature to return new symbolic objects whenever methods are invoked on them.

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

⇒ Exp(foo, [x, y])

```
x + y == y + x
```

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

**Code rewriting.** When a conditional expression depends on a symbolic value, symbolic execution requires that we execute both branches of the conditional and record both possible results. Unfortunately, Ruby does not allow the programmer to attach special behavior to conditionals, so Derailer rewrites the application's code, transforming **if** expressions into calls to Derailer's own definition.

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

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

**Side effects.** To support side effects, our symbolic evaluator must consider the fact that both branches of a conditional may contain updates to the same variable. As such, our strategy for executing conditionals must split the current state and save both results. The new definition of **if** introduced in our code rewriting performs this step, splitting the current state and recording the side effects of each branch separately. It uses a Choice object to record the path constraint causing the split and the results of both branches.

```
x = SymbolicObject.new
if x.even? then x = 1
else x = 2 end
x
```

⇒ Choice(Exp(even?, [x]), 1, 2)

## 4.2 Rails

Figure 3 summarizes Derailer's symbolic execution architecture. Derailer enumerates the actions defined by the application; for each one, Derailer 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, which Derailer normalizes and groups by resource type.

**Wrapping ActiveRecord.** Rails applications interact with the database through ActiveRecord, an object-relational mapper. ActiveRecord provides methods like "find" and "all" to query the database. Given a "User" class extending ActiveRecord, the following code finds users with the name "Joe."

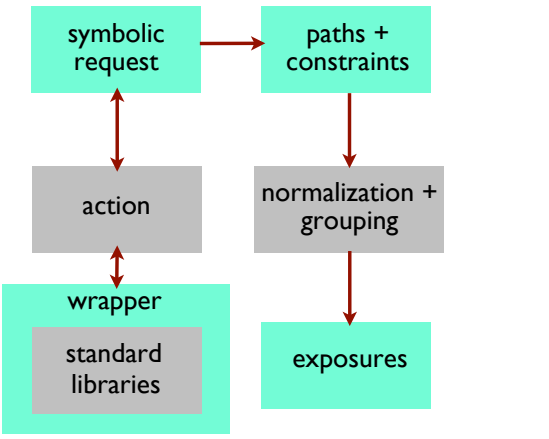



**Figure 3: Derailer's Architecture**

```
User.find :name ⇒ "Joe"
```

Derailer's analysis must produce results valid for *all* values of the database, so database queries must return symbolic values during analysis. To accomplish this, Derailer wraps the Rails API to ignore the concrete database and return symbolic objects instead. The query above, for example, evaluates to:

```
Exp[User, [Exp(query, [name ⇒ "Joe"])]]
```

**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. Derailer 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.

**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.

## 4.3 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 Derailer 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. Derailer rewrites the application's code, transforming `if` expressions into calls to Derailer code that runs both branches. This can result in the exponential blowup characteristic of symbolic execution, so Derailer executes 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 Derailer's versions of the same methods are essentially specifications of the default Rails behavior, Derailer's replacement methods eliminate the extra security checks introduced by CanCan.

Our solution is to allow Derailer's 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 renderer unmodified. After rendering has finished, Derailer collects the set of symbolic values that have been converted to strings, and returns them as the set of results.

## 4.4 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.

| Application | SLOC | Analysis Time | Exposures | Constraints | Paths/Const. |
|---|---|---|---|---|---|
| Redmine | 32,011 | 95.5s | 1536 | 59 | 26.0 |
| FatFreeCRM | 15,794 | 52.8s | 2216 | 34 | 65.2 |
| Diaspora | 41,188 | 112.8s | 2859 | 171 | 16.7 |
| Amahi | 10,561 | 23.1s | 74 | 54 | 1.4 |
| Selfstarter | 2,450 | 8.1s | 10 | 4 | 2.5 |

**Figure 4: Results of Analyzing Five Popular Open-source Rails Applications**

**Derailer assumes 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.

**Derailer doesn't handle symbolic string manipulation.** Derailer records 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 Derailer's inability to solve string manipulation constraints to be a problem in practice.

**Derailer assumes that all actions are reachable.** Rails uses *routes* to define the mapping between URLs and actions. Derailer ignores routes and simply analyzes 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 for Derailer to miss an exposure by ignoring some code that is actually live—Derailer simply analyzes *all* the code.

**Derailer's filtering assumes that logically equivalent constraints are syntactically equal.** Determining logical equivalence is a difficult problem. Fortunately, Rails applications tend to focus on querying the database and formatting the results—they typically do not perform complex arithmetic or string operations—so normalization of database queries is usually enough to make logically equivalent constraints also syntactically equal. And like the other assumptions, this one is conservative: if the user filters based on a constraint that is logically equivalent but *not* syntactically equal to another one, the associated exposure will be highlighted as *not* satisfying the desired security policy—so it is impossible to miss a security bug due to this assumption.

**Derailer assumes that the application under analysis uses ActiveRecord.** Derailer 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 Derailer is 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, Derailer has a simple mechanism to provide a specification of that API.

# 5. EVALUATION

In evaluating Derailer, we sought answers to two basic questions:

**Does Derailer's analysis scale to real-world applications?** To answer this question, the authors ran Derailer on five popular open-source Rails applications. Since all applications are well-tested and we do not know their intended security policies, we did not expect to find bugs. The results show that Derailer's analysis scales to even quite large real-world Rails applications.

**Is Derailer useful in finding security bugs?** To answer this question, 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.

## 5.1 Scalability

We tested Derailer's scalability on five open-source web applications:

- **Redmine**, a content-management system
- **Fat Free CRM**, a customer-relationship management system
- **Diaspora**, a social networking platform
- **Amahi**, a personal media server
- **Selfstarter**, a crowdfunding platform

The results are summarized in Figure 4. All analyses finished in less than two minutes, even in the case of Diaspora, which has more than 40,000 lines of code.

All five applications are popular and mature. The Diaspora project, for example, has been in development for more than three years, has more than 250 contributors, and has addressed more than 4000 filed bugs. As such, we did not expect to discover security bugs in these applications.

However, we did find one bug in Diaspora—a situation that causes a crash when the current user is not a friend of the owner of a requested piece of data. If the bug causing the crash were fixed, a security bug would remain: the sensitive data would be visible to the non-friend.

The larger open-source applications produced many exposures—more than 3000, in the case of Diaspora. The relatively small number of constraints, however, made constraint filtering effective. Filtering by a single constraint often resulted in a huge reduction in the number of visible paths, since so many paths share some common constraints. The large ratio of paths to constraints supports our hypothesis that applications typically access data in only a small number of consistent ways.

## 5.2 Bug-Finding

To evaluate Derailer's ability to find security bugs, we applied it to 127 student assignments from 6.170[3], a web application design course at MIT. The course teaches design principles and implementation strategies, and evaluates students on a series of open-ended projects in which students

[3] `http://stellar.mit.edu/S/course/6/fa13/6.170/`

| 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: Types of Bugs Found During Analysis of Student Projects**

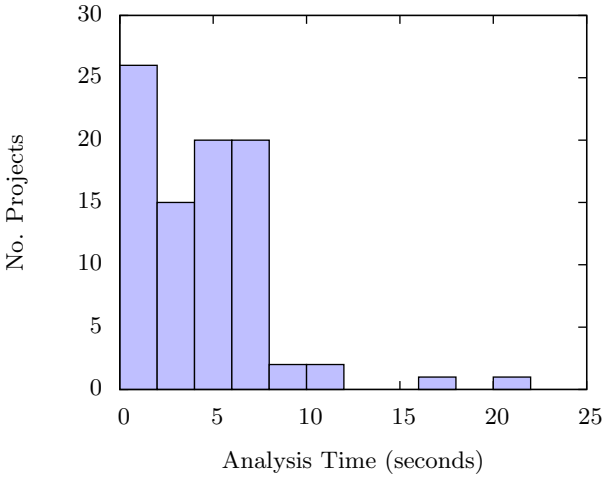



**Figure 6: Analysis Times for Student Projects**

implement actual applications. Students are taught and expected to use Rails, and one of the topics covered is security.

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 first 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

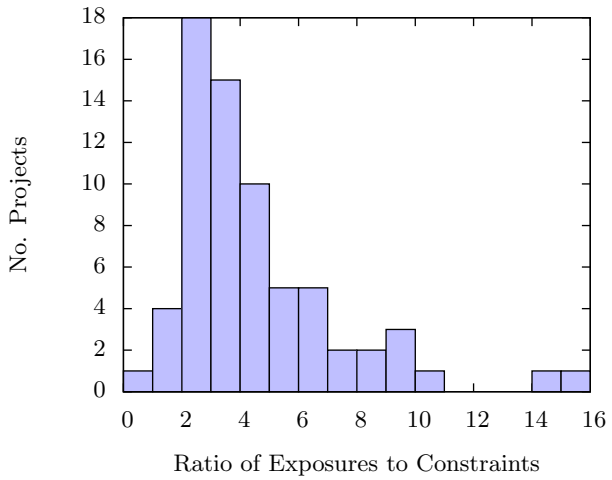



**Figure 7: Ratio of Exposures to Constraints in Student Projects**

minute assembling 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.

### 5.2.1 Results

Figure 8 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 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 7 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.

### 5.2.2 Bugs Found

Figure 5 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

| 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 8: Results of Analyzing 127 Student Projects**

words, less than 15% of the student assignments were correct.

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.

#### 5.2.3 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.

### 5.3 Threats to Validity

**Internal Validity.** Our experimental results may not support our findings for several reasons. If our analysis incorrectly skips some code or misses some constraints, then it might report incorrect results. If our timing strategy was not accurate, then our scalability results might be incorrect. And if we misinterpreted the analysis results, then the set of bugs reported might be incorrect. We mitigated these factors by testing our tool extensively on small applications with seeded bugs; by timing each scalability experiment three times; and by checking our interpretation of each bug in the student projects with the teaching assistant who graded it.

**External Validity.** Similarly, our findings may not generalize to real-world applications. If our assumptions about Rails applications—that they typically use ActiveRecord, have few branches, and iterate only over collections—do not hold, then our scalability results may not be indicative of Derailer's real-world performance. If the types of security bugs that occur in the real world are not similar to the ones in our experimental subjects, then our tool may be less useful for those applications.

We have collected evidence to mitigate the first factor. We examined the 21 most-starred Rails projects on Github. We found that all 21 projects used ActiveRecord exclusively to access the database. And, in a combined 562,527 lines of Ruby code, these applications contain a total of just 36 loops—and more than half contain no loops at all. When Derailer's analysis encounters a loop with a symbolic constraint, it unrolls the loop to the bound specified by the user and marks the resulting exposures for special review, so the user is unlikely to miss security bugs due to loops in controller code.

The second factor is more difficult to mitigate, because determining the prevalence of each type of security bug first requires that those bugs be found. Our experience teaching web application design, as well as the collective experience of the security community, seems to support the idea that the kind of bugs Derailer is designed to find are among the most common.

## 6. 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.

### 6.1 Interactive Analyses

We share the idea of using human insight as part of a semantic program analysis with Daikon [8] and DIDUCE [10], 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 [25] 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.

### 6.2 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 [3]. 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 detec-

tion 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.

## 6.3 Symbolic Evaluation

King [13] and Clarke [6] developed the first symbolic execution systems in 1976, and modern systems [2, 9, 12, 20, 22, 23] have been used to do many types of program analyses. Two notable examples are the symbolic extension of Java PathFinder [12, 20], which has been used to analyze Java code used by NASA, and CUTE [23], 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 number of tools for executing these languages symbolically: for example, Saxena et. al [22], Rozzle [7], and Kudzu [22] for Javascript, and Rubyx [4] for Ruby. In contrast to Derailer, all of these use standalone symbolic evaluators.

Like Derailer, Yang et. al [27] and Köskal et. al [14] 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.

## 6.4 Static Analysis of Web Applications

Existing work on the application of static analysis to web applications focuses on modeling applications, and especially on building navigation models. Bordbar and Anastasakis [1], 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 ( [15, 21, 26]) perform similar tasks but provide less automation. Nijjar and Bultan [19] translate Rails data models into Alloy to find inconsistencies.

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 [5] statically verifies user-defined security properties of web applications, and Chaudhuri and Foster [4] verify the absence of some particular security vulnerabilities for Rails applications.

# 7. CONCLUSIONS

This project explores two hypotheses. First, that **web applications, despite their widespread deployment and apparent flexibility, are in some key respects simpler than traditional applications, making new, lightweight analyses possible**. These include: the conventional structure of the Rails framework, with the database typically accessed through ActiveRecord, a simpler interface than full SQL; the lack of loops in controller actions, which makes symbolic evaluation straightforward; that request parameters are named consistently across actions, so that conditions are often identical rather than just isomorphic; and the statelessness of HTTP, which makes analysis across actions unnecessary (since the programmer must encode all relevant security context in persistent state).

At the same time, performing a symbolic evaluation on the code of a web application is challenging. Web frameworks such as Rails are elaborate and make extensive use of language and API features that are not always well defined. Execution begins with complex, system-wide configuration. To address these issues, we implemented a symbolic evaluator that runs the standard interpreter, intercepting calls to crucial interfaces (such as ActiveRecord) using method overriding. This not only simplified the task of handling a complex language and framework whose full formalization in a conventional symbolic evaluator would be burdensome, but also allows for hybrid evaluation, in which some aspects (notably configuration settings) are executed concretely.

The second hypothesis is that **security policies tend to be highly uniform, and that many vulnerabilities result simply from failing to implement security checks consistently for different actions, formats, and so on**. Our initial experiments seem to support this, with average ratios of exposures to constraints overwhelmingly greater than one.

The uniformity of security policies suggests that it would be profitable to pursue designs with a stronger separation of concerns, with policy more cleanly separated from functionality (and indeed this is the direction seen in many new developments, both in our own group [16] and elsewhere [24, 27], that allow policies to be expressed declaratively, in one place, and enforced globally). Such a move would of course make our analysis less useful, but this seems to us the inevitable push-pull between synthesis and analysis. We suspect that for a long time yet there will be systems that do not achieve such a separation that might benefit from the kind of analysis we have described here.

Our tool seems to strike a reasonable balance between automation and user intervention; as the case studies suggest, the burden on the user in identifying security-related constraints seems to be reasonable. Nevertheless, we are keen to try to automate the analysis fully, perhaps by identifying some reliable heuristics, or by incorporating some kind of machine learning. We also plan to look at other kinds of security problems, both static and dynamic, to see if the ideas we have explored might have broader application.


## Acknowledgements

We are grateful to the anonymous reviewers for their helpful comments. We also thank the 6.170 teaching assistants, Leonid Grinberg, Dalton Hubble, Manali Naik, Jayaprasad Plmanabhan, Vikas Velagapudi, Evan Wang, and Carolyn Zhang, for their feedback on our evaluation results. This research was funded in part by the National Science Foundation under grant 0707612 (CRI: CRD - Development of Alloy Tools, Technology and Materials).


# 8. REFERENCES


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

[2] 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.

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

[4] Avik Chaudhuri and Jeffrey 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.

[5] A. Chlipala and LLC Impredicative. 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] D. Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, 2012.

[12] 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.

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

[14] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *POPL*, pages 151–164, 2012.

[15] DR Licata and S. Krishnamurthi. Verifying interactive web programs. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 164–173. IEEE.

[16] Aleksandar Milicevic, Daniel Jackson, Milos Gligoric, and Darko Marinov. Model-based, event-driven programming paradigm for interactive web applications. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 17–36. ACM, 2013.

[17] Joseph P Near and Daniel Jackson. Rubicon: bounded verification of web applications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 60. ACM, 2012.

[18] Joseph P. Near and Daniel Jackson. Symbolic execution for (almost) free: Hijacking an existing implementation to perform symbolic execution. Technical Report MIT-CSAIL-TR-2014-007, Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 2014.

[19] 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.

[20] C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. *Model Checking Software*, pages 164–181, 2004.

[21] 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.

[22] 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.

[23] 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.

[24] Isaac Strack. *Getting Started with Meteor. js Javascript Framework*. Packt Publishing, 2012.

[25] 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.

[26] P. Tonella and F. Ricca. Dynamic model extraction and statistical analysis of web applications. 2002.

[27] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. pages 85–96, 2012.