Symbolic Security Analysis of Ruby-on-Rails Web Applications

Avik Chaudhuri
University of Maryland, College Park
avik@cs.umd.edu

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

ABSTRACT

Many of today's web applications are built on frameworks that include sophisticated defenses against malicious adversaries. However, mistakes in the way developers deploy those defenses could leave applications open to attack. To address this issue, we introduce Rubyx, a symbolic executor that we use to analyze Ruby-on-Rails web applications for security vulnerabilities. Rubyx specifications can easily be adapted to a variety of properties, since they are built from general assertions, assumptions, and object invariants. We show how to write Rubyx specifications to detect susceptibility to cross-site scripting and cross-site request forgery, insufficient authentication, leaks of secret information, insufficient access control, as well as application-specific security properties. We used Rubyx to check seven web applications from various sources against our specifications. We found many vulnerabilities, and each application was subject to at least one critical attack. Encouragingly, we also found that it was relatively easy to fix most vulnerabilities, and that Rubyx showed the absence of attacks after our fixes. Our results suggest that Rubyx is a promising new way to discover security vulnerabilities in Ruby-on-Rails web applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques, Mechanical verification*

General Terms

Languages, Security, Verification

Keywords

web-application security, symbolic execution, automated analysis

1. INTRODUCTION

Today, online services are a crucial part of many industries such as banking, government, healthcare, and retail. Unfortunately, the web applications that underlie these services often face serious security threats, and vulnerabilities in these applications can lead to loss of revenue, damage to credibility, and legal liability.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

Many web applications are built on top of frameworks whose APIs provide extensive defense mechanisms against common attacks such as cross-site scripting (XSS) and cross-site request forgery (CSRF). However, the mere existence of these APIs is insufficient—to be effective they must be used correctly by the programmer, who must ensure that the logic of the web application cooperates with the design of the APIs. Moreover, even if security-relevant APIs are used correctly, application-specific security vulnerabilities, such as insufficient access control checks or leaks of confidential information, could still remain.

In this paper, we propose addressing this challenge by using *symbolic execution* [19, 15] to analyze Ruby-on-Rails (or just "Rails") web applications. Rails is a popular framework based on Ruby, an object-oriented scripting language. We focus on server-side code, and we are concerned with protecting the web application and honest users from dishonest users or other adversarial clients.

We developed Rubyx, a symbolic executor for Rails, and use it to detect potential vulnerabilities such as XSS, CSRF, susceptibility to session manipulation, and allowing unauthorized access, among others. Unlike most previous work on web-application security [17, 4, 7, 33], we do not study such threats in isolation; by using symbolic execution, we can perform end-to-end reasoning about all of these vulnerabilities simultaneously. And although the low-level details of our approach are targeted toward Rails, we believe the same ideas can be applied to other web application frameworks.

Briefly, symbolic execution involves running code with *symbolic variables*, which are unknowns that range over sets of concrete values. At conditional branches involving symbolic variables, the symbolic executor consults an underlying Satisfiability Modulo Theory (SMT) solver to decide which branches could be taken. If both are possible, the executor conceptually forks execution, trying both paths. Thus, if run to completion, symbolic execution explores *all* paths and hence can verify the absence of vulnerabilities.

Of course, verifying all paths in general would be intractable, since programs can have an unbounded number of paths. However, we have found that web applications are typically "broad" and "shallow"—while there are many possible requests and responses, each request-response path is usually short. Hence this domain is ideal for symbolic execution, because the shallowness of the paths controls the exponential blowup from branches. To handle unbounded data structures, we rely on the small model hypothesis—we initialize databases with a small number of symbolic objects, and prove the absence of vulnerabilities up to that bound. Thus, soundness in our setting is the same as in bounded model checking.

A major advantage of our approach is that it is *programmable*: it can be used to specify and check arbitrary properties of interest. In Rubyx, the programmer calls **assert** e to check that the (arbitrary) Ruby expression e always evaluates to *true*; **assume** e to tell Rubyx

to assume that e holds; and defines methods named **invariant** to specify properties that must be invariant during execution.

Using this simple interface, we show how to encode a variety of security properties at various levels of abstraction. We implemented a proxy Rails API that simulates the original API, and uses notions such as principals—distinguished by knowledge of secrets—and trust to assert XSS safety, CSRF protection, and password authentication. At a high level, XSS safety specifies that only trusted strings can be part of a response; CSRF safety specifies that the principal that sends a request must be at least as trusted as the principal that receives the response; and password authentication specifies that senders and receivers of requests are at least as trusted as the logged-in user. These specifications concisely rule out several classic attacks as well as recent variants studied in the literature [4]. Moreover, our specifications are generic: we can check for XSS safety, CSRF protection, and password authentication simply by symbolically executing a target application in conjunction with our proxy API.

In addition to these generic security specifications, we can also use Rubyx to specify and check application-specific security properties, such as access control and functional correctness. We believe that the breadth of these properties, along with the generic properties above, demonstrates the flexibility and power of using Rubyx to reason about security vulnerabilities.

Rubyx is implemented on top of DRails, a tool we developed previously to "compile" Rails code by making many implicit Rails conventions explicit, which simplifies analysis [1]. Rubyx uses Yices [29] as its SMT solver. To improve performance, Rubyx uses several optimizations, including a careful encoding of the necessary constraints in Yices as well as caching to reduce solver queries.

We applied Rubyx to analyze security of seven Rails applications obtained from various sources. Rubyx found several serious vulnerabilities in these applications, including XSS, CSRF, authentication failures, insufficient access control, and application-specific problems. Encouragingly, we found that it was generally easy to manually fix these vulnerabilities, and that after doing so Rubyx could show that the attacks were eliminated for the fixed applications. Rubyx took between half a minute to 3 minutes in its analysis of these programs, which range from 5k–20k lines of code. Informally, we found the effort required to apply Rubyx to be similar to what we would expect in testing, and we believe this approach will prove viable in practice. Finally, our experiments revealed several common misunderstandings about the defense mechanisms provided by Rails. We have reported these observations to the Rails security team, and are working with them on improving the design and documentation of these mechanisms.

In summary, this paper makes the following contributions:

• We study a range of attacks and defenses in Rails, and explain the intricacies of correctly using Rails security APIs—going much deeper than the rough overview in the OWASP guide [36]. We believe this discussion is of independent interest, as our experiments indicate that developers often do not appreciate the subtleties of Rails's defenses, often rendering them ineffective (Section 2).

• We introduce Rubyx, which we believe is the first symbolic execution engine for Ruby and for Rails. The broad and shallow nature of Rails applications makes them a particularly attractive target for symbolic execution. Rubyx includes several optimizations, including SMT solver query caching (Section 3).

• We show how to encode specifications of low-level security properties such as XSS, CSRF, password authentication, and secrecy, and several application-specific properties, using Rubyx's assume/assert annotation mechanism. We believe that our concise formal specifications of some of these properties are not only new and important for end-to-end security analysis of web applications, but that they help clarify the relevant security concerns and can serve as a guideline for Rails developers (Section 4).

• We evaluate Rubyx and our specifications on seven Rails applications. We discovered several serious attacks against these applications, and that the vulnerabilities were generally straightforward to fix (Section 5). We are working with the Rails security team to ensure that such vulnerabilities can be more easily avoided by future developers.

We believe these results suggest that symbolic execution in general, and Rubyx in particular, is a promising approach for detecting and preventing security vulnerabilities in web applications.

2. ATTACKS AND DEFENSES IN RAILS

In this section we discuss several important vulnerabilities that can arise in Rails programs. In Section 4, we will see how to detect these vulnerabilities using Rubyx.

For illustration, we use examples from *pubmgr*, an application developed by one of the authors to manage publications by members of our research group. Specifically, *pubmgr* maintains a database of users, authors, and publications. An author must be a group member or a co-author of a group member; each author may have several publications. Conversely, a publication may be linked to several authors, some of which must be group members. A distinguished user, *admin*, may identify other users as group members. Such users can then manage their co-authors and publications.

Like all Rails applications, *pubmgr* consists of three kinds of components: *models*—Ruby classes that interface to the database; *views*—either HTML pages with embedded Ruby code or, equivalently, Ruby methods that generate HTML pages; and *controllers*—methods that are invoked when the client requests a web page. A controller receives as inputs any GET or POST parameters submitted by the user, as well as state information encoded in the session. As a controller runs, it may redirect to other controllers, use models to access the database, and return by specifying which view should be rendered in response to the user's request.

2.1 XSS

Several web attacks use *cross-site scripting* (XSS) to execute arbitrary (malicious) code on the browser. In XSS attacks, an adversary embeds executable code (likely JavaScript) in text fields in the web application's database. When a user receives a web page containing those compromised fields, the browser executes the code, possibly leaking the user's secrets or carrying out operations with the user's privileges on behalf of the adversary.

To illustrate potential attacks, consider the following code:

```
1  class AuthorsController
2    def insert_author
3      @author = Author.new(params[:name], params[:webpage])
4      @author.save; render view_author
5    end
6    def find_author
7      @author = Author.find(params[:id]); render view_author
8    end
9  end
10 module AuthorsView
11   def view_author()
12     show(@author.name); show(@author.webpage)
13   end
14 end
```

This code contains two controllers. The first, insert_author, is given a name and a web page, which are passed in via the params hash. On line 3, the controller creates a new Author, which is a model representing a database row. The controller then writes the new author to the database and calls view_author to display a web page in response. That view shows the author's name and web page (line 12). Along the same lines, the find_author controller looks up the input author id in the database and renders the same view to show the author's information.

Unfortunately, while this code is straightforward, it is also vulnerable to XSS attacks: an attacker can use insert_author to create an author whose name or web page contains malicious code.

A typical countermeasure against XSS is to sanitize any text that may ultimately be rendered by the browser, to ensure that untrusted inputs do not embed executable code. One way to do this in Rails is to validate text before writing it to the database, as in the code:

```
15 class Author # model for Author
16   validates_format_of :name, ... # regexp
17 end
```

Here the programmer calls the **validates_format_of** method to tell Rails that before the name field can be written to the database, it must match a given regular expression (elided by . . .). In this way we can prevent code from being included in author names.

Another countermeasure is to HTML-escape text before display. Here is code to do just that whenever the webpage field is rendered:

```
18 module AuthorsView
19   def view_author() ...; show(html_escape(@author.webpage)) end
20 end
```

Both of these countermeasures prevent executable code in displayed web pages. Critically, however, the programmer must remember to use them to enable their protection. Moreover, notice that even for something as simple as sanitization there are different approaches, and the point at which sanitization is applied may vary.

2.2 CSRF

Recently, *cross-site request forgery* (CSRF) has emerged as a powerful technique for several web attacks. CSRF has been described by some experts as a sleeping giant, because its power is (as yet) widely underestimated. CSRF attacks work as follows. Suppose that a user interacts with a web application A while also browsing another web site B. Pages retrieved from B may cause the user's browser to send further requests (e.g., GET requests for images) on behalf of the user. By compromising site B, an attacker can control those requests; in particular, such requests can be sent to application A, and appear to come from the user when in fact they come from the attacker. This is especially harmful if the requests are *non-idempotent* (*i.e.*, they cause state changes).

Preventing these attacks in Rails requires employing several related countermeasures. First, we ensure that any calls that may change state are POST requests. In the following code, we use Rails's **before_filter** method to specify that ensure_post must be called before a request is routed to the insert_author controller.

```
21 class AuthorsController # continued
22   before_filter :ensure_post, :only => :insert_author
23   def ensure_post() redirect_to :error unless request.post? end
24 end
```

Second, since POST requests can still be surreptitiously issued from other web sites open in the browser [36], we require that POST requests include a secret token, which is only available to web pages that may legitimately send POSTs. We do this by calling Rails's **protect_from_forgery** method:

```
25 class AuthorsController # continued
26   protect_from_forgery :only => :insert_author
27 end
```

This call ensures that POSTs to insert_author must include a parameter named :authenticity_token (automatically included by Rails in forms), and this parameter must match an internal token returned by form_authenticity_token(), which is part of the Rails API. Here is a fragment of the code we use in Rubyx for this part of the API:

```
28 class Controller::Base
29   def form_authenticity_token
30     session[:_csrf_token] ||= fresh(:TOKEN)
31   end
32   def forgery_safe?
33     !post? || (params[:authenticity_token] == form_authenticity_token)
34   end
35 end
```

On line 30, form_authenticity_token() either return the current token (stored in session[:_csrf_token]) or generates a fresh one (if session[:_csrf_token] is **nil**). The method forgery_safe? then ensures that this token matches the parameter :authenticity_token for POST requests.

Finally, we must account for "insider attacks," *i.e.*, attacks by users of the application (against other users of the application). To understand this issue, we need to look again at the implementation of token generation on line 30 above. The complication here is that session[:_csrf_token] is not reset automatically by Rails between logins, hence different users that log in from the same IP address could inadvertently be given the same CSRF token. To properly protect against CSRF, the application should always change session[:_csrf_token] to **nil** before logging in a user, so that the token is regenerated whenever a different user logs in. Rails provides a method, **reset_session**, which has just this effect:

```
36 class Controller::Base # continued
37   def reset_session() session = {} end
38 end
```

We should stress that this mechanism is fairly delicate; for example, calling **reset_session** after logging out a user may be inadequate, since we cannot assume that a malicious user will politely log out (and most applications will still log in a different user after).

Our experiments suggest that **reset_session** is seldom used correctly (if at all) to prevent CSRF attacks. One possible reason is that the Rails documentation for **reset_session** focuses on XSS attacks, and developers may think it is unnecessary if they take other measures to prevent XSS. In contrast, when we developed a specification for CSRF protection (Section 4), we pinpointed the significance of **reset_session** for CSRF.

Notice that using CSRF protection is not that easy, and checking that CSRF protection is used correctly requires delicate reasoning. We need to track dynamic checks that ensure requests are POST; we need to distinguish new objects based on context to differentiate tokens generated for different users; and so on. Techniques developed for reasoning about trace properties of security protocols may apply [14]—but such techniques require extensive annotations that the usual Rails developer cannot be expected to provide. In contrast, our symbolic execution-based analysis can readily verify CSRF safety for such code.

2.3 Session manipulation

Next, we consider session manipulation attacks. Sessions usually maintain crucial state. For example, after a user successfully authenticates (and logs in), the identity of the user is often stored

in the session and trusted by the web application. Furthermore, as we have seen above, CSRF tokens are stored in sessions. Thus, maintaining the integrity of sessions is very important for security.

Rails provides two modes for storing sessions. In *database-store* mode, the session is stored in the database, a session identifier is stored in a cookie. This mode is secure but involves some overhead since the database must be accessed for every request.

In contrast, in *cookie-store* mode (the default), the session is stored in the cookie as a marshaled string. This is efficient, but requires that the session be cryptographically protected for integrity; otherwise the attacker may be able to fool the server with a maliciously crafted session. Thus, in Rails, a session sent by the server is hashed with a server-side secret, and the hash is verified for every request. Unfortunately, this does not fully guarantee the integrity of sessions, because it does not guard against *replay attacks*. For example, consider the following code, which defines a controller method authenticate for logging in users in *pubmgr*.

```
39 class UsersController
40   def authenticate
41     password = User.find(params[:id]).password
42     unless params[:password] == password
43       if session[:retrying] then redirect_to :error
44       else session[:retrying] = true; redirect_to :login end
45     end
46     session[:user_id] = params[:id]
47   end
48 end
```

Here the user's password is looked up (line 41) and, if not found, the user is given one chance to retry (lines 42–45). The field :retrying of session tracks whether the user has already retried. Unfortunately, this code is vulnerable to a replay attack: if some user fails to log in, that user can replay the session before the try to "roll back" the state of :retrying, effectively allowing any number of tries.

The Rails documentation recommends not maintaining sensitive information in the session, although it does not help in deciding what information is sensitive. In our experience, whenever developers store non-standard information in sessions (possibly to cut down on database accesses), there is a high probability that application-specific properties can be violated using replay attacks. Safe use of cookie-store mode requires careful programming and thorough reasoning about sensitive information and sessions.

Replay attacks are easy to detect with symbolic execution, by exploring paths in which the current session may be any of a set of past sessions. In contrast, specialized techniques that do not take session replay into account would be unsound in Rails.

2.4 Unauthorized access

Finally, applications can implement access control to prevent unauthorized access to data and enforce specific secrecy and integrity properties. The following code snippet illustrates one way we enforce authorization in *pubmgr*:

```
49 class PublicationsController
50   before_filter :internal_user, :only => :show_manuscript
51   before_filter :user_is_author, :only => :edit_publication
52   def internal_user
53     redirect_to :login unless User.internal?(session[:id])
54   end
55   def user_is_author
56     all = Publication.find(params[:id]).authors
57     redirect_to :login unless all.include?(User.author(session[:id]))
58   end
59 end
```

This code specifies filters before the controllers show_manuscript and edit_publication (not shown). The internal_user method checks whether the current user is a member of our research group and, if not, redirects the user to a login controller, thus ensuring that only internal users can view unpublished manuscripts (a secrecy property). Similarly, the user_is_author method ensures that the current user is an author of the publication to be edited, so that authors can edit only their own publications (an integrity property).

As with the previous examples, we can see that Rails provides support for preventing unauthorized access, but it is still up to the programmer to determine how to code their access controls using the Rails API. Moreover, reasoning about this code relies on other security properties, such as correct password authentication and safety against session manipulation, CSRF, XSS, and so on. Our symbolic execution-based approach is very effective because it can reason uniformly and simultaneously about all of these properties.

3. SYMBOLIC EXECUTION WITH RUBYX

As we saw in the previous section, ensuring that Rails's security defenses are used correctly requires reasoning about many low-level details of code. This is a perfect task for symbolic execution, which can automatically explore many possible program executions, including corner cases that may be hard to find otherwise. In this section, we discuss the design of Rubyx, our symbolic execution engine for Rails. In Section 4, we give details of how we encode detection of security vulnerabilities using Rubyx.

At its core, Rubyx is a Ruby source code interpreter, with one key difference: in addition to modeling concrete program values, Rubyx can interpret programs that contain *symbolic variables*, which are unknowns that represent arbitrary sets of concrete values. Rubyx tracks these unknowns as they flow through the program. Rubyx also maintains *path conditions*, that track constraints on symbolic variables; initially, the path condition is simply *true*.

When we reach a branch with a guard p that involves symbolic variables, we conceptually split the current *world* (*i.e.*, the state of the Ruby program) into two new worlds, one in which p is conjoined with the path condition, and one in which $\neg p$ is conjoined with the path condition. We pass the new path conditions to an SMT solver, Yices [29], to decide whether one or both conditions are actually satisfiable, *i.e.*, whether the corresponding world is *reachable* from the start of the program. We continue executing the reachable world(s) forward, splitting the worlds in the future as necessary. In this way, Rubyx can simulate all paths through the program that are reachable for any concrete values that the unknown might take. More discussion of symbolic execution can be found elsewhere [19, 15].

3.1 Specification and verification

Rubyx includes several built-in primitives for specifying and checking properties. The method call **fresh**(n) returns a fresh symbolic variable named after n, which may be any Ruby symbol (*i.e.*, interned Ruby string). (The name is just a convenience in understanding Rubyx's output.) Such a symbolic variable can range over any Ruby object, although its structure is constrained by subsequent operations on it. The method call **assume**(p) conjoins the path condition with p, which may be any Ruby expression. (In conditional tests in Ruby, **false** and **nil** are both treated as *false*, and all other values are treated as *true*.) The **assume** primitive is used to specify a precondition for a property we wish to verify. Dually, the method call **assert**(p) checks whether the path condition implies p; if not, Rubyx reports an error. In other words, **assert** specifies postconditions for properties we wish to verify.

Lastly, Rubyx supports *object invariants*. A method definition of the form **def invariant**() p **end** in any class maintains p as an invariant for all objects of that class. More precisely, we **assume** p

when an object instance is created, and we **assert** p whenever there is an update that changes the object's state. Rubyx uses an efficient algorithm that monitors parts of the state relevant to an invariant and re-enforces the invariant only when those parts are modified.

3.2 Integration with Yices and Optimizations

As with most symbolic execution systems, Rubyx's capabilities and performance depend heavily on exactly how it uses Yices, its underlying SMT solver. Next, we will discuss some of the key challenges we encountered in working with Ruby, Rails, and Yices.

First, Ruby hashes, such as params and session, are used pervasively in code. To reason as generally as possible in these situations, we model hashes with uninterpreted functions in Yices, which allows us to leave the hashes as unknowns while still supporting usual lookup, update, and equality operations [24].

Second, we found that strings appear pervasively in code (in particular as inputs and outputs), and we often want to treat them as arbitrary unknowns while still supporting concatenation and other operations. In our experience, burdening Yices with constraints generated by string operations leads to poor performance. Instead, we evaluate and reason about string operations abstractly in Rubyx by maintaining partial solutions for strings in the state.

Third, defining an appropriate datatype for Ruby values in Yices is crucial for sound reasoning with (in)equalities. Unfortunately, Yices does not allow the kind of recursive datatype definitions necessary to express most Ruby values. We get around this problem by using an uninterpreted type in the definition, and carefully designing the form of constraints so that this type is always interpreted as the original datatype when solving those constraints.

Overall, we put a lot of effort into ensuring that Yices can determine the satisfiability of constraints generated by Rubyx. We encode queries in the decidable fragment of the input language of Yices, so Yices should always terminate with either "satisfiable" or "unsatisfiable" on our queries. This is in contrast to related tools for which theorem provers may fail and require further annotations or dynamic checks [6].

Finally, we implemented a number of optimizations in Rubyx to dramatically improve performance. Most importantly, we found that many of the worlds Rubyx explores share logically identical path conditions. Thus, we maintain a cache of constraints that Yices has already solved, and avoid resolving such constraints.

Another important factor for performance is the ordering of clauses in the constraints passed to Yices. On several occasions we found that changing the ordering can reduce Yices solving time from almost an hour to less than a second. Thus, we keep constraints in a normalized form, with clauses sorted in a fixed order. Our ordering is designed to place simple conditions before more complex ones, and Yices calls have never taken more than a couple of minutes (at the extreme) with our ordering. Maintaining clauses in a normalized order also improves cache hits.

Our last important optimization is to implement some basic operations, such as lattice operations on secrecy levels (see Section 4), in Yices rather than in Rubyx. This increases the complexity of the constraints passed to Yices, but greatly reduces branching in the interpreter, which saves space and time.

4. SECURITY ANALYSIS WITH RUBYX

There are three steps to analyzing Rails applications with Rubyx. First, we apply DRails, a tool we previously developed [1] to make Rails code easier to analyze by making many implicit Rails conventions explicit. For example, DRails explicitly adds database access methods to models, inserts calls to **render** that are implicit in Rails, and translates HTML with embedded Ruby code into pure Ruby.

```
60 # No XSS
61   assert (output.trust?) unless (Prin.receiver == Lattice.Bot)
62 # Secrecy
63   assert (Lattice.leq (output.secrecy?, Prin.receiver))
64 # No CSRF
65   assert (Lattice.leq (Prin.receiver, Prin.sender)) if params[:post]
66   assert params[:post] if (Session.modified? || Db.modified?)
67 # Authentication
68   assert (Lattice.leq (session[Prin.Id], Prin.receiver))
69   assert (Lattice.leq (session[Prin.Id], Prin.sender)) if params[:post]
```

Figure 1: Specifications of common security properties

We likely could have used Rubyx for this purpose, but as DRails was already developed it was convenient to start with.

Second, we import proxy implementations of (a subset of) the remaining Rails API methods, the browser, and the web server. Overall, the imported code amounts to less than 150 lines of Ruby. Besides providing a functional environment for the application program to run in, the imported code specifies and verifies some common, low-level security properties.

Finally, we execute an *analysis script*, provided by the developer, that includes several symbolic "tests." Each such test populates the database with some symbolic objects, defines some invariants for those objects, assumes some preconditions, sends symbolic requests to the browser interface, receives responses, and asserts some postconditions. The goal of the analysis script is to direct the exploration of paths and specify and verify further, high-level properties of the application.

After running the analysis script, Rubyx reports the reachable worlds in which the properties do not hold, *i.e.*, it reports satisfiable path constraints that cause assertion failures, which in turn encode feasible path traces to exploit bugs.

Secrecy lattice and principals. Low-level security properties of code ultimately rely on the preservation of secrets, such as tokens and passwords. Thus, our specifications are based on *principals*, which are secrecy levels in a lattice. In practice, the principals include various users (as identified by the application), the application itself ($\top$), and the attacker ($\bot$); they are partially ordered by their knowledge of secrets, with $\top >$ *honest users* $> \bot$. The honest users do not know each others' secrets; dishonest users leak their secrets to the attacker; and the application's secrets are exclusive. To model secrecy levels, our proxy implementation includes a class Lattice with constants Bot and Top and a method leq.

For any interaction (request and response) with the web application, we consider three roles: $\mathcal{P}_s$, the principal that sent the request; $\mathcal{P}_r$, the principal that received the response; and $\mathcal{P}_i$, the principal of the logged-in user. Our proxy implementation includes a class Prin to model these roles: we have $\mathcal{P}_s$ = Prin.sender, $\mathcal{P}_r$ = Prin.receiver, and $\mathcal{P}_i$ = session[Prin.Id]. While Prin.Id is constant for an application (*e.g.*, for *pubmgr*, Prin.Id = :id), the roles $\mathcal{P}_s$, $\mathcal{P}_r$, and $\mathcal{P}_i$ may change between interactions; all of these are set by the analysis script.

With these roles, we specify four low-level security properties—*No XSS*, *Secrecy*, *No CSRF*, *Authentication*—in the proxy browser interface, so that they are verified for every interaction. These specifications rule out the kinds of vulnerabilities discussed in Section 2. The exact specifications are shown in Figure 1; we discuss these in detail in later subsections, but for now, observe that the specifications are quite simple despite covering several properties.

Furthermore, in combination they facilitate reasoning about end-to-end security: *No XSS* and *Secrecy* together imply preservation of the secrecy lattice, while *No CSRF* and *Authentication* rely on

secrecy to ensure the soundness of access control. Finally, other high-level properties specified in analysis scripts, such as enforcing that operations maintain consistency of the database, typically rely on these low-level properties for the specifications to be correct.

Analysis scripts. Analysis scripts typically proceed as follows. First, the database is populated with objects containing unknown fields. For example, to create a User the developer may call:

```
70 User.create(fresh(:ATTRIBUTES))
```

The create method, automatically added by DRails, populates the fields of a new User object according to the argument and saves the object in the database. In this case, we pass in a fresh unknown hash (named after ATTRIBUTES), and so Rubyx will in turn introduce fresh unknowns for all fields in the User object. Also, those unknowns will automatically be constrained by any **invariant**s given by the developer. For example, we may require that the hashed_pwd attribute should be the cryptographic hash of the @password field.

Having established an appropriate state, various controllers are "tested" by sending requests with unknown parameters. For example, the developer may request the browser to send a login request to UserController:

```
71 response = Browser.exec(UserController, :login, fresh(:PARAMS))
```

The parameters are given by a fresh unknown hash, and so parameters like the POST/GET type of the request and the CSRF token carried by the request, as well as other request-specific parameters such as a password, will be unknowns. Any conditions on such parameters that are established dynamically by code—such as checking the type of the request, matching the CSRF token, checking the password, and so on—cause Rubyx to explore alternative worlds with the relevant constraints. And in each of those worlds, Rubyx will check the low-level security properties in Figure 1.

In addition, the developer may specify and verify other properties. For example:

```
72 assume Browser.session[:id]
73 response = Browser.exec(UserController, :update, fresh(:PARAMS))
74 # Update by admin
75   assert(User.admin?(Prin.sender)) if Db.modified?(User)
```

Here the developer assumes that the session id is non-**nil**, indicating a successful login. Then the developer sends a request for an update operation, and asserts that if the User database is modified, then the user is an administrator.

The constraint passed to Yices to check this assertion will include various facts that arise from the specifications of the low-level properties in Figure 1. For example, the request must be POST (by our enforcement that only POST requests may write to the database); the sender must be Browser.session[:id] (by CSRF protection and write authentication for POST requests); and Browser.session[:id] must be an admin (by a specific dynamic check in the application code, not shown).

In the following subsections, we explain our formal specifications for the common security properties in Figure 1. These specifications apply to any Rails application that uses the Rails API in standard, recommended ways to achieve security. Other, application-specific properties that may arise are discussed on a case-by-case basis for our experimental benchmarks in Section 5.

4.1 Session manipulation

Since session manipulation can be used to violate other properties, it is important to implement sessions faithfully. Recall that in the database mode, only the session identifier is stored in a cookie, while the session information itself is stored in the database. In this mode there is not much room for session manipulation, and so in our proxy implementation, the browser is not allowed to affect the Rails internal @session field.

On the other hand, in the cookie-store mode, the browser tracks the current session internally, and sessions are stored cryptographically MAC-ed in the cookie. Thus, requests may change the current session by changing the cookie. In our proxy implementation, we abstract MAC-ing of an object x by an identity operation with the side effect of including x in a private list. Verifying the MAC of x then reduces to checking whether x is in this list. This encoding allows sessions to be replayed but not forged. Moreover, our proxy implementation ensures that principals cannot know (and thus, cannot replay) sessions received by other principals.

4.2 XSS

Next we tackle XSS attacks. Recall from Section 2.1 that Rails includes two defenses: using **validates_format_of** to prevent fields with code from being stored in the database, or calling **html_escape** to sanitize strings before display.

To track sanitization, our Rails proxy implementation extends Ruby's String class with a trust level. For a string s, calling s.trust? returns true if the string is trusted, and calling s.trusted marks a string as trusted. We reduce Rails's sanitization routines into statements on trust. To encode **html_escape**, we simply implement it as a proxy API that returns a trusted code of the string passed to it:

```
76 module View::Base # continued
77   def html_escape(s) s.trusted end
78 end
```

We translate **validates_format_of** into trust assumptions. For example, here is a translation of the call to **validates_format_of** from Section 2.1:

```
79 class Author # continued
80   def save() assume @name.trust?; Author.db ≪ self end
81 end
82 def Author.invariant() @db.forall {|author| author.name.trust?} end
```

Since we will reject any unsanitized strings, we can assume in save, which writes to the database, that name is trusted (line 80). (Note that we do not check the regular expression that name is tested against; Rubyx assumes it is correct.) This maintains an automatically generated invariant of the database: the name fields of all author objects in it are trusted (line 82). In turn, this invariant implies that any names retrieved from the database are trusted.

Once we have established the trust of strings, we can reason about XSS. In our proxy implementation, any strings displayed in a response are concatenated to the @output field of the response. In Rubyx, concatenating string x with string y sets the trust of x to the minimum of the trusts of x and y. We also assume that any strings passed as parameters in a request are untrusted (not shown). Then the specification of *No XSS*, shown on line 61 of Figure 1, is simply that the @output field of a response must be *trusted* unless the output is received by an attacker.

This technique captures the key principle behind defenses against XSS: that inputs must be sanitized before they flow to outputs. By assuming that inputs may be untrusted and requiring that outputs be trusted, we effectively force inputs that flow to outputs to pass through some sanitization mechanisms, which in turn "bless" the untrusted strings as trusted. (We handle SQL injection similarly, but do not mention it further since we have not seen SQL injection vulnerabilities in Rails—the applications we've studied always use secure Rails APIs for database queries and updates.)

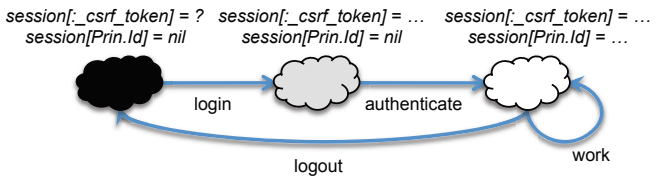

Figure 2: State machine for session

4.3 CSRF

Next, we consider CSRF attacks. Recall that for any interaction with the web application, we consider principal $\mathcal{P}_s$, the sender, and $\mathcal{P}_r$, the receiver. CSRF is possible because $\mathcal{P}_s$ and $\mathcal{P}_r$ may be different. Thus, our specification of CSRF safety is simply:

(CSRF safety) For any non-idempotent request, $\mathcal{P}_s \geq \mathcal{P}_r$.

Informally, *any principal that controls behaviors of the web application must be at least as trusted the principal that views those behaviors*. In particular, if $\mathcal{P}_r$ is an honest user, then $\mathcal{P}_s$ cannot be the attacker, although it may be $\mathcal{P}_r$ itself or another honest user $> \mathcal{P}_r$ (or perhaps even the web application).

We specify CSRF safety in Rubyx with two assertions. On line 65 of Figure 1, we require that POST requests have the right relationship between sender and receiver, and on line 66 we specify that any requests that change the session or database must be POST.

To reason about CSRF protection, we must track which principal each string came from. Hence, similarly to trust levels in Section 4.2, we extend class String with a secrecy attribute, which contains a Lattice element describing the string's principal.

Recall from Section 2.2 that defense against CSRF requires employing several different countermeasures: ensuring only POST requests can change state; including and checking for a secret value session[:_csrf_token] in legitimate requests (achieved by calling **protect_from_forgery**); and calling **reset_session** at the right point to create fresh tokens for new users.

To understand how these countermeasures provide CSRF safety, we argue abstractly about an application. Consider Figure 2, which shows a state machine describing the life cycle of session[:_csrf_token] and the currently logged-in principal when CSRF protection is used correctly. Here, the user is initially at the black state, where session[Prin.Id] is **nil** and session[:_csrf_token] is irrelevant. We move to the gray state when a login request is received, at which point a fresh session[:_csrf_ token] is generated. We then move to the white state when password authentication succeeds, which also sets session[Prin.Id]. We stay in that state doing work, and eventually return to the black state by logging out.

Thus, $\mathcal{P}_r$ may change only in the black state. If an implementation matches the state machine in Figure 2, then session[:_csrf_token] must change whenever $\mathcal{P}_r$ may have changed. More precisely, the following assertion, which we include in forgery_safe? (Section 2.2), will hold, since form_authenticity_token (Section 2.2) returns session[:_csrf_token] after generating a token if required:

```
83 class Controller::Base
84   def forgery_safe?
85     assert (form_authenticity_token.secrecy == Prin.receiver)
86     ... # see line 33
87   end
88 end
```

Thus we can conclude session[:_csrf_token].secrecy $= \mathcal{P}_r$.

Now, for any POST request, forgery_safe? (Section 2.2) establishes session[:_csrf_token] = params[:authenticity_token]. Putting these two equalities together, we have params[:authenticity_token].secrecy $= \mathcal{P}_r$.

Finally, we assume that any string sent with a request must have secrecy level $\leq \mathcal{P}_s$ (*i.e.*, that senders can only send messages at their secrecy level or below). Then we have params[:authenticity_token].secrecy $\leq \mathcal{P}_s$, and thus $\mathcal{P}_r \leq \mathcal{P}_s$, which is CSRF safety for POST requests.

Note that in the state machine in Figure 2, we generate a new CSRF token for a login request. Traditionally CSRF attacks have focused only on work transitions, hence it seems we could have generated fresh tokens after authentication. However, recently *login CSRF* attacks have been studied [4], which focus on authentication transitions. It is easy to see that generating CSRF tokens *before* authentication, as we have done, prevents login CSRF attacks also.

Furthermore, note that for authentication transitions, we may reasonably assume that $P_s \not> \mathcal{P}_r$, *i.e.*, $\mathcal{P}_s$ will never aid any less trusted $\mathcal{P}_r$ in authentication. For instance, an honest user $\mathcal{P}_s$ will not try to authenticate for a dishonest user $\mathcal{P}_r$, or for another honest user $< \mathcal{P}_s$. Thus, for authentication transitions that are CSRF safe we actually have $\mathcal{P}_s = \mathcal{P}_r$.

Finally, note that key to our proof above was the assertion that the secrecy of the CSRF token must be equal to the receiver. We propose this as a design principle for CSRF safety. Indeed, it provides the main insight behind eliminating CSRF attacks, embodied in this combined use of **protect_from_forgery** and **reset_session**—the secrecy of the CSRF token must always be related to the principal viewing the behaviors of the web application, and since such a principal may change between logout and login, the CSRF token must also change between logout and login. Otherwise, the attacker may learn the token for an honest user, or the token for an attacker may serve an honest user—either of which can break CSRF safety.

4.4 Authentication and Access Control

Next, we show how Rubyx can assert the correctness of password authentication, meaning that sends and receives after authentication can be assumed to come from the logged-in user (or a more trusted principal). Formally, our specification is as follows (see lines 68 and 69 in Figure 1).

(Password authentication) Suppose that $\mathcal{P}_i \neq$ **nil** (i.e., a user is logged in). Then:

$$\text{For any POST request: } \mathcal{P}_s \geq \mathcal{P}_i, \text{ and} \quad (1)$$

$$\text{For any POST or GET request: } \mathcal{P}_r \geq \mathcal{P}_i \quad (2)$$

To authenticate correctly, web applications usually store and check passwords using a combination of cryptographic hashing and "salting" to avoid known attacks. In our proxy Rails API, we include a basic model of string concatenation and hashing. Our implementation ensures that for any strings x, y, and z, if $x \neq y$ then Crypto.hash(x) $\neq$ Crypto.hash(y) and $x + z \neq y + z$. (Here Crypto is a placeholder for the name of the relevant Ruby class.)

Given this API, Rubyx can reason about typical authentication strategies. As a concrete example, consider the following code:

```
89  class UsersController
90    def authenticate # POST
91      u = User.find(params[:id])
92      if (u.hashed_pwd == Crypto.hash(params[:password] + u.salt))
93        session[:id] = u.id; render :logout_form
94      else redirect_to :abort end
95    end
96  end
97  class User
98    def password() @password end
99    def password=(x)
100     @password = x; hashed_pwd = Crypto.hash(x + salt)
```

```
101   end
102 end
103 def User.invariant
104   @db.forall {|u|
105     (u.password.secrecy == u.id) &&
106     (u.hashed_pwd == Crypto.hash(u.password + u.salt)}
107 end
```

Notice that the programmer has supplied an invariant (lines 103–106) that specifies that the password's secrecy level is the id of the user, as well exactly how the hashed password is computed. This invariant describes the stored password information abstractly, and lets us leave it otherwise as an unknown.

Using this invariant, Rubyx can assert that the code implements correct password authentication as follows. Suppose that $\mathcal{P}_i \neq$ **nil**. For some user u = User.find(params[:id]), the code above establishes the following conditions:

$$\begin{aligned} \text{u.hashed\_pwd} &= \text{Crypto.hash(params[:password] + u.salt)} \\ \text{u.id} &= \text{session[:id]}\ (= \mathcal{P}_i) \\ \text{u.password.secrecy} &= \text{u.id} \\ \text{u.hashed\_pwd} &= \text{Crypto.hash(u.password + u.salt)} \end{aligned}$$

Combining these equations with those for cryptographic hashing and string concatenation above, we have params[:password].secrecy $= \mathcal{P}_i$. Furthermore, as in Section 4.3, we assume any string sent with a request must have secrecy level $\leq \mathcal{P}_s$. Then we have $\mathcal{P}_i =$ params[:password].secrecy $\leq \mathcal{P}_s$. This establishes (1). But since we are performing authentication, by CSRF safety we have $\mathcal{P}_r = \mathcal{P}_s$, and hence (2) also holds.

After authentication has occurred, $\mathcal{P}_r$ and $\mathcal{P}_i$ remain constant. (Recall that $\mathcal{P}_r$ can change only in the black state, and $\mathcal{P}_i$ does not change in the white state.) Thus $\mathcal{P}_r \geq \mathcal{P}_i$, from (2) at authentication time, continues to hold. Moreover, by CSRF safety we have $\mathcal{P}_s \geq \mathcal{P}_r$, and so (1) also continues hold.

Of course, authentication is useful only if we implement some access control, for which we need to track dynamic conditions on $\mathcal{P}_i$ itself, of the form $\mathcal{P}_i \geq \mathcal{P}_a$ for some access privilege $\mathcal{P}_a$. By the above theorem, these conditions will imply $\mathcal{P}_s \geq \mathcal{P}_a$ for write access privileges, and $\mathcal{P}_r \geq \mathcal{P}_a$ for read access privileges.

4.5 Secrecy

Finally, note that the above properties rely on the preservation of the secrecy lattice, *i.e.*, principals do not eventually know secrets of unrelated principals. We specify this in line 63 of Figure 1, by requiring that any strings received in a response must have secrecy levels $\leq \mathcal{P}_r$, so that the receiver could have already known them.

5. EXPERIMENTS AND RESULTS

We used Rubyx to analyze seven Rails applications with a range of non-trivial security and correctness requirements. Two applications were developed within our research group, but independently of this project (so they did not take any particular advantage of Rubyx's strengths or weaknesses). The remaining applications were obtained from external sources.

5.1 Applications and Properties

We begin by describing each web application in our experiments, along with their application-specific access control and correctness properties. For all of the applications, we also checked for the presence of session manipulation, XSS, CSRF, and authentication attacks, using the specifications in Section 4.

As discussed earlier, the *pubmgr* application, which we used for examples throughout the paper, was developed by one of the authors to manage publications of our research group. In addition to the common security properties, we also checked that users are always properly authorized, as outlined in Section 2.4.

The *coffee* application was developed by another member of our research group to track use of a shared coffee machine. The application maintains an inventory of coffee capsules available; a count of each user's tokens, which are exchanged for coffee; and the consumption of coffee by users. Some users have administrative privileges and can refill tokens for other users and adjust the inventory of capsules. We checked to ensure that administrative privileges cannot be circumvented, and that counts of capsules, tokens, and user consumption all match up correctly.

The *depot* application is used as the main running example in a popular book for Rails developers [27]. Since its code is freely available, we expect that many developers use that code as a starting point for their own applications. The *depot* application maintains a database of products, and records orders of products by customers. It also maintains accounts of administrators, who are able to edit information on products, including their prices. We checked to ensure that administrative privileges are required to edit product information, and to introduce other administrators to the system. We also checked to ensure customers are charged the correct price for any product they order.

The *chuckslist* application is a classified ad system inspired by craigslist. The application maintains a database of ads and their authors. It also maintains accounts for users, some of whom may have administrative privileges. Authors manage ads, users manage authors, and users with administrative privileges manage other users. The system relies on a sophisticated access control module, which we included as part of the application. We checked for several access control properties, such as: authors cannot edit ads of other authors, and administrative privileges cannot be circumvented.

The *boxroom* application is a secure file sharing system that maintains access control metadata for files. Users are members of groups, and permissions are associated with groups and files. We checked that several expected access control properties hold, such as non-administrators cannot modify the permissions for groups and files, and users cannot access a file they do not have permission to access.

The *mystic* application is a trouble ticket system. The application maintains a database of (outstanding and resolved) tickets, as well as accounts for customers, technical-staff members, and administrators. Customers post tickets, technical-staff members address those tickets, and administrators manage their accounts. The developers claim a clear separation of these duties in the system, and we check to ensure that this is indeed the case.

Finally, the *rtplan* application is a planning system for project tasks. The application maintains a database of projects, each of which has several tasks. It also maintains accounts for users, who are allocated various tasks, and records the work those users have put into those tasks. Apart from expected access control properties, we checked that the project and user databases have a consistent record of the total work done.

5.2 Attacks

Figure 3 summarizes the results of our experiments. We group our results by property, shown across the top of the figure: *No XSS*, *No CSRF*, *Authentication*, *Secrecy*, access control, and other application-specific correctness properties. For each property, a check mark indicates no vulnerabilities found, a cross mark indicates some exploitable security vulnerability, and a question mark indicates that a potential security vulnerability exists, but it may not be exploitable. We indicate that some vulnerabilities are due to replay attacks with an r. For most exploitable or potential vulnerabilities, we attempted to fix the code so that the vulnerability was

	XSS	CSRF	Auth.	Secr.	Acc.	Corr.
pubmgr	$\surd$	$\times(1)$	$\surd$	$\surd$	$\surd$	N/A
coffee	? (2)	$\times(1)$	$\surd$	$\surd$	$\times$ (2)	$\surd$
depot	$\times(2)$	$\times(10)$	$\surd$	$\surd$	$\surd$	$\times^r$ (7)
chuckslist	$\surd$	$\times(1)$	$\surd$	$\surd$	$\times$ $(-)$	N/A
boxroom	$\surd$	$\times(5)$	$\surd$	$\surd$	$\surd$	$?^r$ (2)
mystic	$\times(17)$	$\times(8)$	$\surd$	$\surd$	$\times^r(6)$	N/A
rtplan	$\surd$	$\times(1)$	$\times(16)$	$\surd$	$?^r$ $(-)$	$\surd$

$\surd$ = no vuln. found $\times$ = vuln. found ? = potential vuln. found
$(n) = n$ lines of fixes $(-)$ = did not fix r = replay attack

Figure 3: Experimental Results and Fixes

eliminated (as checked by Rubyx). The number of fixes is listed in the figure; we show a dash where we chose not to attempt fixes because doing so might require pervasive changes.

As we can see, Rubyx detected many vulnerabilities, and every application had at least one. At the same time, except for three cases we were able to fix the vulnerabilities with a small number of changes. Since Rubyx explores all possible program paths from a given symbolic state, once we have eliminated all detected vulnerabilities from that state we can guarantee the security of that application from any instantiation of that state.

Next, we discuss the vulnerabilities we found in more detail.

XSS. Rubyx found XSS attacks in three applications. In *coffee*, the attributes of a capsule (including a text description) are not sanitized, and hence can be used for XSS attacks. We categorized this issue as a potential, rather than definite, vulnerability because only administrators can modify attributes. However, this still reflects poor security practice because administrators could use XSS attacks to steal passwords from other users.

In *depot*, a product's information includes information on orders that were placed on it, and the text of such orders is not sanitized. Thus, any customer that places an order can mount XSS attacks on other customers and administrators who view this data. Similarly, in *mystic*, users can mount XSS attacks on other users.

CSRF. Rubyx found CSRF attacks against all applications. In *pubmgr*, the vulnerability exists because the controller for logging in a new user does not use **reset_session**—instead it simply resets session[:user] to **nil**, but does not also set session[:_csrf_token] to **nil**. In *coffee*, the developer does use **reset_session**, but in the controller for logging out a user instead of logging in—hence CSRF attacks are enabled by simply not logging out. Similar problems occur in *chuckslist*, *rtplan*, *depot*, *boxroom*, and *mystic*. Moreover, the latter three also do not require that all state-changing requests are POST, enabling yet more CSRF attacks.

Authentication. Only one application, *rtplan*, failed to correctly authenticate users. Strangely, here users do not require passwords to log in, although users do have password attributes in the model.

Access control. Rubyx found critical violations of access control in several applications. In *coffee*, non-administrators can change their token counts or even grant themselves administrator privileges. The problem is that the code to update a user's information can write to *all* of the attributes of that user, even in cases when only some information should being updated. The developer of *coffee* tried to prevent the token count and administrator bit from being updated in non-administrative mode by rendering one of them as read-only and not rendering the other in the relevant form. Unfortunately, this is not enough: a simple browser setting can make read-only parameters writable, and any missing parameters can be passed by extending the form manually. This is an interesting example, because it shows how even a developer who pays attention to security can get it wrong.

	LoC	Running time		Yices stats	
		orig. (s)	re-run(%)	% time	# calls
pubmgr	7,450	32.6	37.2%	91.4%	352
coffee	7,928	67.5	4.0%	97.8%	142
depot	5,338	32.5	16.6%	94.8%	265
chuckslist	12,554	175.2	10.8%	95.7%	519
boxroom	13,727	123.1	32.0%	95.6%	347
mystic	20,350	28.2	30.3%	71.8%	169
rtplan	9,849	195.3	1.5%	99.1%	163

Figure 4: Performance

In *chuckslist*, administrative privileges are not protected—anybody can sign up as an administrator, and any existing user can obtain administrative privileges. Additionally, any author can trivially become a user and manage other authors in the system. For example, an author and all its ads may be removed without an existing user logging in. Finally, any author can edit any ad in the system. In fact, *chucklist* does not actually maintain any relationship between users and authors, which is surprising, because at the database level they have very similar attributes.

In *mystic*, the claimed separation of duties is violated. Anybody can sign up as an administrator and can modify others' accounts. Furthermore, privileges granted to a logged-in user are stored in the session, and so session replay can be used to defeat revocation. For example, even if a technical-staff member's privileges are revoked by an administrator, it may replay a stale session to keep viewing information on outstanding tickets.

In *rtplan*, any user can promote anyone to an administrator. Furthermore, the isadmin attribute of a user is stored in the session, so a user whose administrative privileges have been revoked can keep acting as an administrator by replaying a stale session. Strangely, however, administrator privileges do not seem to have any function in this application, so we must consider these attacks benign.

Application-specific security properties. Finally, Rubyx found violations of application-specific correctness properties in two cases. In *depot*, Rubyx found attacks in which the price in an order does not match the price of the ordered products. The problem is that the shopping cart of a customer—which is used to place the order—is stored in the session, which can be replayed. Thus, a customer may add some products to the cart, save the session, and empty the cart. Later, the administrator may increase those products' prices. But then the customer can replay his session and check out, paying the lower prices. Indeed, the database is not consulted before placing an order—there is complete trust in the session.

In *boxroom*, Rubyx found a potential vulnerability in a clipboard implementation that could be exploited by a replay attack. However, the clipboard is not yet enabled, and so the attack is latent until the clipboard is actually used.

5.3 Performance

Finally, we briefly discuss Rubyx's performance, summarized in Figure 4. The applications we analyzed ranged from 5k to 20k lines of code. The running times for the original applications were between half a minute to 3 minutes. (Note that if an assertion fails, Rubyx reports a bug and continues, assuming that the assertion actually succeeded.) On average, around 90% of the time was spent on calls to Yices; indeed, one such call took 75 seconds out of 195 seconds of total running time. Caching such calls not only improved performance (we observed a high cache hit rate), but also allowed us to analyze the applications "incrementally." Indeed, during our experiments we often ran partial analysis scripts on an

unfamiliar application to get intermediate results, and then incrementally added assumptions and assertions to get the final results; caching prevented the cumulative analysis times from blowing up quadratically across runs. The resulting experience was much like testing. To measure this effect, note that re-running the analyses took only 2–40% of the original running times, and the fraction went down as the time spent on calls to Yices went up.

6. RELATED WORK

Some necessary background on important threats and defenses for web-application security can be found in [26, 35], while an excellent resource for Rails security in particular is [36].

Much research on web-application security has focused on settings where applications are untrusted, and users must be protected from applications [16, 38, 20]. In contrast, applications are not considered inherently malicious in our setting—we assume that some users may be malicious, and we care about verifying that applications and other users are protected from such malicious users.

The need for end-to-end web application security has been previously argued [13]. Existing techniques for achieving end-to-end security for web applications include type systems [8, 5] and secure compilation [32, 9, 3]. Most of these techniques rely on lattice-based security with labels [12, 21, 10], as we do. An important difference between such techniques and ours is that we require almost no label annotations. We also ignore implicit information flows.

Many specialized techniques have been developed to analyze particular security properties of web applications [31, 23, 37, 18, 34]. In contrast, our approach can be used to check a wide range of security properties. Some recent tools also use symbolic execution for security analysis of JavaScript programs [2, 28].

Finally, most of the existing work on understanding web application security focuses on injection attacks [30]. In comparison, the other kinds of attacks we consider in this paper have received far less attention; notable exceptions include [25, 11] on access control and authentication, [17] on session integrity, and [4, 22] on CSRF.

7. CONCLUSION

We described a new approach that uses symbolic execution to reason about the security of Ruby-on-Rails web applications. Our symbolic executor, Rubyx, uses a simple assume/assert language to describe a wide range of properties. We use Rubyx to specify protection against XSS, CSRF, session manipulation, and unauthorized access, using basic notions such as principals, secrecy, and trust levels. We applied Rubyx to seven applications, and found a wide variety of vulnerabilities. Overall, our results suggest that symbolic execution is a promising approach for analyzing web applications for security vulnerabilities.

Acknowledgments. We wish to thank the anonymous reviewers for their helpful comments on this paper. This research was supported in part by DARPA ODOD.HR00110810073.

8. REFERENCES

[1] Jong-hoon An, Avik Chaudhuri, and Jeffrey S. Foster. Static typing for Ruby on Rails. In *ASE*, 2009.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst. Finding bugs in web applications using dynamic test generation and explicit state model checking. *IEEE Transactions on Software Engineering*, 2010.

[3] I.G. Baltopoulos and A.D. Gordon. Secure compilation of a multi-tier web language. In *TLDI*, 2009.

[4] A. Barth, C. Jackson, and J.C. Mitchell. Robust defenses for cross-site request forgery. In *CCS*. ACM, 2008.

[5] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

[6] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. In *ICFP*, 2010.

[7] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2), 2010.

[8] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *PLDI*, 2010.

[9] S. Chong, K. Vikram, A.C. Myers, et al. SIF: Enforcing confidentiality and integrity in web applications. In *USENIX Security*, 2007.

[10] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD*, 2009.

[11] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *USENIX Security*, 2009.

[12] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5), 1976.

[13] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *HOTOS*, 2007.

[14] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization in distributed systems. In *CSF*, 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[16] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *WWW*, 2009.

[17] M. Johns. SessionSafe: Implementing XSS immune session handling. *ESORICS*, 2006.

[18] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *S&P*, 2006.

[19] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[20] S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *S&P*, 2010.

[21] J. Magazinius, A. Askarov, and A. Sabelfeld. A Lattice-based Approach to Mashup Security. In *ASIACCS*, 2010.

[22] Z. Mao, N. Li, and I. Molloy. Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. *Financial Cryptography and Data Security*, 2009.

[23] M. Martin, B. Livshits, and M.S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

[24] J. McCarthy. Towards a mathematical science of computation. *Information Processing*, 62, 1962.

[25] G. Naumovich and P. Centonze. Static analysis of role-based access control in J2EE applications. *ACM SIGSOFT Software Engineering Notes*, 29(5), 2004.

[26] OWASP. The ten most critical web application risks, 2010. http://www.owasp.org/images/0/0f/OWASP_T10_-_2010_rc1.pdf.

[27] Sam Ruby, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Bookshelf, 2009.

[28] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript, 2010. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley.

[29] SRI. Yices: An SMT solver. http://yices.csl.sri.com/.

[30] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL*, 2006.

[31] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis for Java. In *PLDI*, 2009.

[32] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *CCS*, 2009.

[33] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[34] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, 2007.

[35] Web Application Security Consortium. Web application security statistics, 2008. http://projects.webappsec.org/Web-Application-Security-Statistics.

[36] Heiko Webers. Ruby on rails security, v2. OWASP report: http://www.owasp.org/images/2/26/Owasp-rails-security.pdf.

[37] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security*, 2006.

[38] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *POPL*, 2007.