Finding Access Control Bugs in Web Applications with CanCheck[*]

Ivan Bocić and Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara, USA
{bo,bultan}@cs.ucsb.edu

ABSTRACT

Access control bugs in web applications can have dire consequences since many web applications store private and sensitive data. In this paper we present an automated verification technique for access control in Ruby on Rails (Rails) applications. Our technique starts by automatically extracting a model that captures 1) the ways the data is accessed and modified by the application, 2) the access control policy of the application, and 3) the authorization checks used for access control policy enforcement. Then, it automatically translates this model to first order logic and uses automated theorem provers to check whether the declared access control policy is correctly enforced by the implementation. We implemented our technique in a tool called CanCheck. Using CanCheck on open source Rails applications, we found numerous previously unknown exploitable access control bugs as well as several deficiencies in access control policies.

CCS Concepts

•**Software and its engineering → Access protection; Model checking; Software verification; Automated static analysis; Formal software verification;** *Software verification and validation;*

Keywords

Access Control, Logic-based Verification, Web Applications

1. INTRODUCTION

Due to the increasing amount of digital information that individuals produce, and the convenience of cloud-based data stores, people increasingly trust their private and sensitive data to web applications. Web applications are becoming the digital banks that store people's data currency. This makes access control one of the most critical concerns for modern web applications.

Web applications execute *actions* in order to respond to user requests. Actions typically modify or read data from the data store and present it to the user in some format. Access control in web applications ensures that access to the data store (both in terms of reads and writes) is restricted according to the role of the current user. The *access control policy* specifies the permissions for each role. When executing actions, before an operation (read or write) is executed on a data item, the access control policy should be enforced using *authorization checks* in order to prevent unauthorized access. Errors in the implementation of authorization checks (i.e., errors in the enforcement of the access control policy) result in access control bugs.

In order to increase programmer productivity and to prevent common design and implementation errors, modern web applications are written using web application frameworks such as Ruby on Rails (Rails) and Django. Rails does not natively support access control. Instead, access control is implemented either in an ad-hoc manner or by using third party libraries (gems). Web applications written in Rails most commonly use gems called CanCan [7], CanCanCan [8] and Pundit [33] for access control.

In this paper we present an automated verification approach for access control policy enforcement in applications that use role-based access control. We first automatically extract an access control model from the application, containing information about the database schema, actions, and authorization checks implemented inside actions. We infer a formal access control policy from the informal access control specification developers have do specify to implement access control. We translate the extracted access control model to a set of first order logic (FOL) formulas. After this logic translation, using off-the-shelf FOL theorem provers, we can identify actions that violate the access control policy by not enforcing it correctly, or we can confirm to the developer that the access control policy is correctly enforced at all times.

Verification of access control in Rails applications has been studied [31, 9, 21, 29]. The contributions in this paper are:

- Using instrumented execution [4] to extract access control models from Rails applications that use CanCan.
- Translating an access control policy and authorization checks for policy enforcement to first order logic.
- Evaluating our verification technique on open source Rails applications.

In Section 2 we introduce Rails and overview our method. In Section 3 we define data store models with regards to access control policies. In Section 4 we explain extraction of access control models from Rails applications that use

[*]This work is supported in part by the NSF grant CCF-1423623.

```
1   class User < ActiveRecord::Base
2     devise :database_authenticable
3     enum role: [:admin, :nonadmin]
4     has_many :articles, foreign_key: :author_id
5   end
6   class Article < ActiveRecord::Base
7     belongs_to :author, class: User
8   end
9
10  class Ability
11    def initialize(user)
12      can [:index, :show], Article
13      if user.admin?
14        can :manage, :all
15      else
16        can :manage, User, id: user.id
17        can :manage, Article, author_id: user.id
18        can [:like, :dislike], Article
19      end
20    end
21  end
22
23  class ArticlesController < ActionController::Base
24    def destroy
25      article = Article.find params[:id]
26      raise unless can? :destroy, article
27      article.destroy!
28      respond_to(...)
29    end
30
31    def create
32      @article = Article.new(params)
33      @article.author = current_user
34      @article.save!
35      respond_to(...)
36    end
37  end
38
39  class UsersController < ActionController::Base
40    def index
41      @users = User.all
42      respond_to(...)
43    end
44  end
```

Figure 1: Rails application example.

CanCan. In Section 5 we define authorization properties and translate them to FOL. In Section 6 we experimentally evaluate our approach. In Section 7 we discuss related work and in Section 8 we conclude the paper.

2. OVERVIEW

Let us examine a segment of a Rails application presented in Figure 1. We split this application into three sections: the list of model classes (lines 1-8), the Ability class (lines 10-21), and three actions (lines 24-29, 31-36, and 40-43).

Model classes define the types of objects that are stored by the application. Users typically use a web application to browse and manage these objects. In this example we have two classes: the User class (lines 1-5) and the Article class (lines 6-8). Objects of the User class correspond to users of the application, and whenever a user logs in, an instance of the User class that corresponds to that user is loaded from the database (declaration in line 2). Users have two possible roles, `admin` and `nonadmin` (line 3), and can have articles associated with them as authors (line 4). The Article class represents articles users can write, and they can be authored by users (line 7). We omit all other details for brevity.

In CanCan, the access control policy is specified using Ability objects. The developer has to implement the Ability class in order to use CanCan. Every time an action is invoked, an ability object that corresponds to that user request is created. Typically, the way this object is initialized depends on the role of the user making the request. Once this object has been created, it can be queried during the action's execution to check if certain operations are permissible for the current user.

In our example, the Ability class is defined in lines 10-21 and the access control policy is specified in the constructor (lines 11-20). The constructor takes a single argument, the User object representing the current user (line 11). Inside the constructor we see branches and `can` statements. These statements whitelist certain operations on certain objects. For example, in line 12, the current user is permitted to execute operations `index` and `show` on all Article objects. These operations by convention correspond to action names. If the user has the `admin` role (line 13), then in line 14 he is permitted to execute all operations (special keyword `manage`) on all objects of all types (special keyword `all`). If the user does not have the `admin` role (line 15), he is permitted to manage his own User object (line 16), all Articles he authors (line 17), and to like and dislike any Article object (line 18).

Finally, we see three actions, two in the Articles controller and one in the Users controller. The first action is used to delete an article object (lines 24-29). It reads an article object from the data store using the unique identifier supplied with the request parameters (line 25), then it aborts the action unless the current user can execute the `destroy` operation on said article (line 26). This check will implicitly query the ability object generated for the current user before the action started executing. Afterwards, assuming the operation was permitted, the article is deleted (line 27). Line 28 synthesizes the response and is omitted for brevity.

The second action is used to create an Article object (lines 31-36). This action creates an Article object from request parameters (line 32), then assigns the author of this new article to be the current user (line 33), and saves the article (line 34). Note that, even though authorization is not checked anywhere in the action, this action cannot violate the policy: either the current user is an `admin` and is permitted to create Articles (line 14), or the current user is the author of this new article (line 17).

Finally, the third action is used to present all User objects to the user (lines 40-43). It does so by loading all user objects from the database and storing them in an instance variable, denoted by the prefix "`@`" (line 41). Because this is an instance variable and not a local variable, its value is exposed to the view. This action may violate the access control policy by giving a non-admin user the ability to read all user objects. This action is problematic even if the web interface does not expose this action to non-admins, as it is easy to create HTTP requests that invoke actions with any set of parameters.

Our Approach.

In order to detect access control bugs in Rails applications we implemented the approach outlined in Figure 2 in a tool called CanCheck[1]. This is an extension of our previous work [4, 5, 6] to include access control.

[1]The tool is available at https://github.com/Bocete/adsl

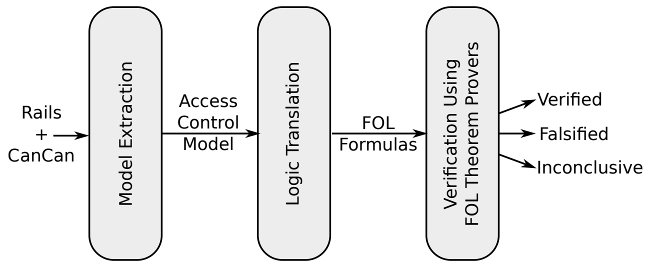

Figure 2: CanCheck tool architecture.

First, we automatically extract an *access control model* from a given Rails application. This model is specified in an intermediate specification language that is based on earlier work in data model verification [4, 6]. It includes information about the model classes, the set of possible roles and the access control policy, as well as actions defined in a high level imperative language. This model is an abstraction of the analyzed application focusing only on how model objects are read and modified, and how authorization checks are implemented in the application.

In addition to extracting the model, we extract *authorization properties* from the application. These properties represent the correct enforcement of the access control policy. The follow the form: *for every action, for any object possibly manipulated by this action, and for every operation done on this object, this operation must be authorized by the access control policy.* We translate these authorization properties, the model schema and the action (including authorization checks) to first order logic (FOL). We use off-the-shelf FOL theorem provers to check whether the resulting formulas are valid (or satisfiable, depending on the encoding and the prover). If yes, then access control is correctly enforced. If no, the access control policy is incorrectly enforced.

FOL theorem provers respond in one of three ways: either they validate access control (in which case it is guaranteed that the inferred authorization property is correctly enforced by the application), or they invalidate access control (in which case there exists an execution that violates access control). Alternatively, since FOL is in general undecidable, the theorem prover might not terminate. If the theorem prover does not produce a result in 5 minutes we halt its execution, and mark the result as inconclusive. However, we have not observed any inconclusive results in our experiments on open source Rails applications. In all cases, a conclusive result was produced in a matter of seconds.

3. ACCESS CONTROL MODEL

In this section we overview the access control model that CanCheck extracts from a web application. The access control model consists of the classes and associations between them, actions which can be used to update the state, user roles which are used to distinguish different types of users, and the access control policy. Figure 3 shows the access control model that corresponds to the Rails application in Figure 1. Section 3.1 defines data stores, and Section 3.2 defines the correct enforcement of access control in data stores.

3.1 Data Store

A *data store* DS is a tuple $\langle C, L, A, R, P\rangle$ where C is a set of classes, L is a set of associations, A is a set of actions, R is a tuple defining user roles, and P is a set of permissions.

```
1  authenticable class User {
2    0+ Article articles
3  }
4  class Article {
5    1 User author inverseof articles
6  }
7
8  roles admin, nonadmin
9  axiom inusergroup(admin) xor inusergroup(nonadmin)
10
11 permit admin, nonadmin read allof(Article)
12 permit admin read, create, delete allof(User)
13 permit admin read, create, delete allof(Article)
14 permit nonadmin read, create, delete current_user
15 permit nonadmin read, create, delete current_user.articles
16
17 action Articles_destroy {
18   article = oneof(allof(Article))
19   if not (
20       inusergroup(admin) or
21       (inusergroup(nonadmin)
22         and article in current_user.articles)
23     ) {
24     raise
25   }
26   delete article
27 }
28 action Articles_create {
29   @article = create(Article)
30   @article.author = current_user
31 }
32 action Users_index {
33   @users = allof(User)
34 }
```

Figure 3: An access control model.

A *data store state* is a tuple $\langle O, T, U\rangle$ where O is the set of objects, T is the set of tuples denoting associations among objects, and U is the set of role assignments for users. We define $\overline{DS}$ to be the set of all data store states of DS.

Classes and Objects.

Given a data store $DS = \langle C, L, A, R, P\rangle$, C is the set of classes, and it identifies the types of objects that can be stored in the data store. Each class has a set of superclasses (superclass$(c) \subset C$) and, transitively, the superclass relation cannot contain cycles.

One class $c_U \in C$ is called the *authenticable class.* The authenticable class is the class whose objects represent users of the application. Typically it is called just `User`. The concept of the user class is used in Rails whenever authorization and/or authentication take place, and it makes it straightforward to associate users with data that belongs to them.

Given a data store state $\langle O, T, U\rangle \in \overline{DS}$, O is the set of objects that are stored in a data store at some point in time. Each object $o \in O$ is an instance of a class $c \in C$. We use the notation O_c to encapsulate all objects in O whose class is c or any subclass of c. We define $\overline{O}$ to be the set of all sets of objects that appear in $\overline{DS}$.

In each data store state $\langle O, T, U\rangle \in \overline{DS}$, there exists a special object $o_U \in O_{c_U}$ that represents the *current user object.* The current user is the currently logged in user who is executing an action. In the example specification in Figure 3, C is defined in lines 1-6: $C = \{\text{User}, \text{Article}\}$ with User denoted as the authenticable class: $c_U = \text{User}$.

Associations and Tuples.

Associations define how objects of particular classes can be related to one another. An association $l = \langle name,\ c_o,\ c_t,$

$card\,\rangle \in L$ contains a unique identifier $name$, an origin class $c_o \in C$, a target class $c_t \in C$ and a cardinality constraint $card$ (such as one-to-one, one-to-many etc.).

Similarly to how objects are instances of classes, tuples are instances of associations. Each tuple $t \in T$ is in the form $t = \langle l, o_o, o_t \rangle$ where l is an association $l = \langle name, c_o, c_t, card \rangle \in L$ and $o_o \in O_{c_o}$ and $o_t \in O_{c_t}$. For a tuple $t = \langle l, o_o, o_t \rangle$ we refer to o_o as the origin object and o_t as the target object.

In the example specification in Figure 3, L is defined in lines 2 and 5. L contains just one one-to-many association between Users and Articles:

$$L = \{\langle \text{“User.articles”}, \text{User}, \text{Article}, \text{one-to-many} \rangle\}$$

Even though lines 2 and 5 both declare associations, line 5 shows that “Article.author” is an inverse of “User.articles”. As such, lines 2 and 5 refer to the same association.

Actions.

Given a data store $DS = \langle C, L, A, R, P \rangle$, A denotes the set of actions. Actions are used to query and/or update the data store state. Each action $a \in A$ is a set of *executions* $\langle s, s', \alpha \rangle \subseteq \overline{DS} \times \overline{DS} \times \overline{O}$ where $s = \langle O, T, U \rangle$ is the pre-state of the execution, $s' = \langle O', T', U' \rangle$ is the post-state of the execution, and $\alpha \subseteq O'$ is the set of objects shown to the user as the result of this action's execution.

Given an action $a \in A$ and an execution $\langle s, s', \alpha \rangle \in a$, we can define the sets of objects this execution created, deleted, and read as follows:

$$o \in created(\langle s, s', \alpha \rangle) \Leftrightarrow o \notin s \wedge o \in s'$$
$$o \in deleted(\langle s, s', \alpha \rangle) \Leftrightarrow o \in s \wedge o \notin s'$$
$$o \in read(\langle s, s', \alpha \rangle) \Leftrightarrow o \in \alpha$$

In access control models, an action is a sequence of statements. Statements are state transitions specified using a combination of boolean and object set expressions. Boolean expressions have the usual semantics, and object set expressions represent a set of objects of a common class.

To illustrate how statements correspond to state transitions, let us show the semantics of the `delete(expr)` statement. Note that this statement operates with an object set expression. Assuming that α is the set of objects that the `expr` argument evaluates to, and that $\langle s, s' \rangle \in \overline{DS} \times \overline{DS}$ are the pre- and post-states of the statement, this statement transitions from $s = \langle O, T, U \rangle$ to $s' = \langle O', T', U' \rangle$ if and only if all three constraints below hold:

$$\forall x \colon x \in O' \Leftrightarrow x \in O \wedge x \notin \alpha \quad (1)$$
$$\forall x \colon x \in T' \Leftrightarrow x = \langle l, x_o, x_t \rangle \in T \wedge x_o \notin \alpha \wedge x_t \notin \alpha \quad (2)$$
$$U' = U \quad (3)$$

In other words, (1) x is an object in the post-state if and only if it is an object in the pre-state that is not in α, (2) x is a tuple in the post-state if and only if it is a tuple in the pre-state, and neither the origin or the target object of x are in α, and (3) role assignments do not change.

Roles and Role Assignments.

In a data store $DS = \langle C, L, A, R, P \rangle$, R is a tuple $\langle \overline{R}, R_x \rangle$ where $\overline{R}$ is the set of *roles*, and R_x is a *role constraint.*

Roles are used to distinguish different types of users. For example, a role could be defined to distinguish administrators from other users, or employees, etc.

In a data store state $\langle O, T, U \rangle$, $U \subseteq O_{c_U} \times \overline{R}$ is a set of role assignments. Each role assignment $\langle o, r \rangle \in U$ defines that user object o has user role r in this data store state.

The role constraint R_x is a function that maps a data store state to a boolean: $R_x \colon \overline{DS} \to \{true, false\}$. The rule constraint ensures a data store state's role assignments are semantically correct. This is application dependent and can be an arbitrary condition, but in practice it often declares that each user has to have exactly one user role.

For example, line 8 of Figure 3 defines the set of roles $\overline{R}$ to be $\{\texttt{admin}, \texttt{nonadmin}\}$. The constraint in line 9 defines that roles `admin` and `nonadmin` are mutually exclusive:

$$R_x(\langle O, T, U \rangle) \Leftrightarrow$$
$$\forall o \in O_{c_u} \colon (\langle o, \texttt{admin} \rangle \in U \Leftrightarrow \langle o, \texttt{nonadmin} \rangle \notin U)$$

Permits.

Given a data store $DS = \langle C, L, A, R, P \rangle$, P is the set of permits. Permits are used to whitelist operations on a data store, depending on the role of the user executing the operation and the object the operation is executed on.

Each permit $p \in P$ is a tuple $\langle g, ops, e \rangle$ where $g \subseteq \overline{R}$ is a non-empty set of roles to which the permit applies, $ops \subseteq \{create,\ delete,\ read\}$ is a non-empty set of operations permitted by this permit, and e is an expression $e \colon \overline{DS} \to \overline{O}$ that maps a data store state to a set of objects: $e(\langle O, T, U \rangle) = \alpha$ such that $\alpha \subseteq O$. The expression e is used to determine the set of objects to which the permit applies to. Note that, for simplicity, we treat update operations as an atomic sequence of delete and create operations.

For example, line 11 of Figure 3 defines that all `admin`s and `nonadmin`s are permitted to *read* all objects of the Article class. This corresponds to the following permit:

$$p = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, e \rangle$$
$$\text{where } o \in e(\langle O, T, U \rangle) \Leftrightarrow o \in O_{\text{Article}}$$

Similarly, line 15 of Figure 3 defines that all `nonadmin`s can do any operation on all Articles for which the current user is the author. Let l be the “User.articles” association, and o_U the current user object. Formally, this corresponds to the following permit:

$$p = \langle \{\texttt{nonadmin}\}, \{create, read, delete\}, e \rangle$$
$$\text{where } o \in e(\langle O, T, U \rangle) \Leftrightarrow \langle l, o_U, o \rangle \in T$$

3.2 Access Control Correctness

Authorization is fundamentally about ensuring that users can view and modify only data that they have been permitted to view and modify, using a set of methods that are permitted to them. The goal of our tool is to check whether all operations that could be executed by any action and for any user are permissible with respect to the access control policy. We formalize this below.

Given a data store $DS = \langle C, L, A, R, P \rangle$, a permit $p = \langle g, ops, e \rangle \in P$ *accepts* an operation op on an object o in state s, denoted as $p[op, s, o]$, if and only if the current user o_U has at least one role from g, the operation op is in ops, and o is inside the set that e evaluates to in data store state s. Formally:

$$p[op, s, o] \Leftrightarrow (\exists r \in g \colon \langle o_U, r \rangle \in U) \wedge op \in ops \wedge o \in e(s)$$

Now that we defined how to check permissions for a given operation in a given state for a given set of objects, we need to extend this check to cover all possible behaviors.

One question we need to answer before we can do that is: In which state of an action's execution should we check for permissions? If we choose the pre-state, it becomes impossible to check permissions for object creation because the created objects do not yet exist in the pre-state, as well as tuples that might be necessary to check the access control policy correctly. Similarly, it is impossible to evaluate permissions for the delete operation in the post-state of an execution. In order to handle all possible scenarios, we chose to evaluate creation permissions in the post-state (once all the objects and tuples have been created), deletion permissions in the pre-state (before any objects or tuples have been deleted), and read permissions in the post-state (as this is the state shown to the user of the application).

We define whether an action $a \in A$ of a data store $DS = \langle C, L, A, R, P \rangle$ correctly enforces the access control policy as follows. An action correctly enforces the access control policy if and only if, for every execution in a:

- There exists a permit that accepts the creation of every object created by this execution,
- There exists a permit that accepts the deletion of every object deleted by this execution, and
- There exists a permit that accepts the read operation on every object read by this execution.

Formally, an action a correctly enforces the access control policy if and only if:

$$\begin{aligned}
&\forall \langle s = \langle O, T, U \rangle, s' = \langle O', T', U' \rangle, \alpha \rangle \in a\colon \\
&\quad \forall o \in \mathit{created}(\langle s, s', \alpha \rangle)\ \exists p \in P\colon p[\{\mathit{create}\}, s', o] \\
&\quad \wedge \forall o \in \mathit{deleted}(\langle s, s', \alpha \rangle)\ \exists p \in P\colon p[\{\mathit{delete}\}, s, o] \\
&\quad \wedge \forall o \in \mathit{read}(\langle s, s', \alpha \rangle)\ \exists p \in P\colon p[\{\mathit{read}\}, s', o]
\end{aligned}$$

For example, let us take a look at the `Articles_destroy` action in Figure 3. This action will read one article and store it in a variable in line 18. Then, lines 19-25 will attempt to check whether deleting this variable is authorized for the current user. If the user is neither: 1) an admin (line 20), nor 2) a normal user and `article` is a member of the articles of the current user, then the action is aborted (line 24). Finally, said object is deleted in line 26. Logically, if any object is deleted, then the action has not been aborted in line 24. Therefore, the current user is either an `admin` (in which case the permit in line 12 accepts the object and the operation), or the current user is a `nonadmin` but is also the author of the article (which is covered by the permit in line 15). Since there exists no execution in which an article is deleted without proper authorization, this action correctly enforces the access control policy.

Now let us examine the `Articles_create` action in lines 28-31 of Figure 3. This action creates an Article object (line 29) and assigns the current user to be the author of the new article (line 30). In every execution of this action, the created article's author must be the current user. Therefore, if the current user is an `admin`, the permit in line 12 accepts this operation. Otherwise, if the current user is a `nonadmin`, the permit in line 15 will accept this operation. Hence, this action does not violate the access control policy.

Finally, the action in lines 32-34 of Figure 3 reads all User objects in line 33 and stores them in a variable. In the case the current user is a normal user, and if there is more than one user in the database, no permit accepts this read operation. Hence, this action violates the access control policy.

4. EXTRACTION FROM CANCAN

In this section we describe the extraction of access control models from Rails applications that use CanCan or CanCanCan. The extraction technique we use is based on instrumented execution [4] which has been developed for model extraction from dynamically typed languages such as Ruby. In this paper, we extend the instrumented execution technique to specifically address extraction of access control models. For access control verification, three components related to access control need to be extracted: 1) the authenticable class and the user role set, 2) the access control policy, and 3) the authorization checks implemented in actions.

4.1 User Role Extraction

Neither Rails nor CanCan implement authentication by default. Instead, they rely on third party libraries to define the authenticable class and roles. CanCan is usually paired with Devise [11] for this purpose. Extracting the authenticable class and roles is straightforward from an application that uses Devise. If an application does not use Devise, we rely on the convention that the authenticable class is called `User` and define roles according to branch conditions in the Ability object (see Section 4.2).

4.2 Access Control Policy Extraction

In CanCan, the access control policy is declared in the Ability class (for example, lines 10-21 in Figure 1). Every time an action is executed, an Ability object is implicitly generated with the current user in mind.

A typical Ability class constructor is defined with a sequence of `if/elsif/else` branches where branch conditions query the user role of the current user. `can` statements outside these branches apply to all roles. Each `can` statement permits the current user to execute a set of operations on a class type, with optional qualifiers that restrict the set of objects the `can` statement applies to.

The process of access control policy extraction is essentially no different from the instrumented execution we use to extract actions, just altered for the purpose of extracting CanCan policies. For example, any interactions that the constructor might have with the database will be extracted and modeled instead of executed concretely, making this process insensitive to the state of the database.

In order to extract the policy, we instrument the Ability constructor as follows. All branches are instrumented for the runtime to execute *both* branches, associating encountered `can` statements with the user groups from the branch condition. We override the `can` directive to generate permits in the access control model based on its arguments. Then, we initiate instrumented execution by creating an Ability object. As for the current user argument that the constructor requires, we use a symbolic value that corresponds to the `current_user` object set expression.

In order to extract the set of roles a `can` statement applies to, during instrumented execution of the Ability constructor, each branch is associated with a set of roles. Any `can` statement executed under a branch is assigned roles that correspond to the branch condition (or branch conditions, in case branches are nested). The root block of the constructor applies to the entire set of roles.

For example, the `can` statement in line 12 of Figure 1 is in the root block of the constructor and, as such, applies to both roles. In line 13 we have a branch condition that only accepts `admins`, meaning that the statement in line 14 only applies to `admins`. Finally, in line 15 we have an else statement, which means that the statements in lines 16-18 refers to all roles associated with the root block but that were not expected by the branch condition: $\{\texttt{admin}, \texttt{nonadmin}\} \setminus \{\texttt{admin}\} = \{\texttt{nonadmin}\}$.

The second piece of information that needs to be extracted is the set of operations. The operation symbols used in `can` statements, by convention, correspond to action names. Moreover, Rails has a strong convention on action names that correspond to create/read/update/delete (CRUD) operations [35]: each model class typically has a separate action for each CRUD operation. Therefore, if we recognize that a user is permitted to execute an action that by convention corresponds to a CRUD operation on objects of a model class, then we infer that the user is permitted to execute the corresponding CRUD operation on objects of the corresponding model class.

Table 1 presents our mapping of action names to operations. For example, `new` and `create` actions by convention serve to create a new object of a given model class. If a user is permitted to execute these actions, we infer that the user is permitted to create objects of corresponding type in general. Note that we treat object updates as a composite operation of deleting an object and subsequently creating a new one. This is sufficient because we abstract basic types away, so objects contain no internal state. Modifying foreign keys corresponds to deleting a tuple and creating another.

Table 1: Mapping Actions to CRUD operations

Operation symbols	Implied CRUD operations
`:manage`	create, delete, read
`:create`, `:new`	create
`:destroy`	delete
`:index`, `:show`	read
`:update`	create, delete

Finally, extracting the expression of a permit is straightforward. If a `can` statement refers to a class without any additional constraints, it applies to all objects of said class in a given data store state. In case there are additional constraints (such as in lines 16 and 17 in Figure 1), considering that we use instrumented execution [4], we can simply evaluate the condition to extract the symbolic representation of the set of objects the `can` statement refers to.

For example, the `can` statement in line 12 of Figure 1 is executed in the root block and as such applies to all user roles (`admin` and `nonadmin`). It lets users of these roles to execute `:index` and `:show` operations, implying the *read* operation in the resulting permit. It refers to the Article class without any additional limitations. Therefore, this `can` statement translates to the permit in line 11 of Figure 3.

The `can` statement in line 14 of Figure 1 is executed in a code block that refers to the `admin` role. It allows the user to `:manage` (do any CRUD operation) to all objects. It translates to two permits in lines 12 and 13 of Figure 3.

The two statements in lines 16 and 17 of Figure 1 directly translate to the two permits in lines 14 and 15 of Figure 3. Finally, the `can` statement in line 18 of Figure 1 does not translate to a CanCheck permit as `like` and `dislike` are not CRUD operations. However, as we explain later, CanCan authorization checks may still refer to this `can` statement.

4.3 Authorization Check Extraction

Extracting the access control policy is not enough for verification of access control in a web application. Runtime enforcement of the policy via authorization checks is an integral part of access control in Rails. Our goal is to ensure that actions, considering implemented authorization checks, correctly enforce the access control policy at all times. There are two ways to check authorization using CanCan: 1) explicitly, using the `can?` method inside action code, and 2) implicitly, using automatically generated authorization checks.

Both explicit and automated authorization checks are extracted during action extraction via instrumented execution, as described below.

Explicit checks.

The `can?` method takes two arguments: an *operation* symbol and the *subject* of the authorization check. For example, line 26 of Figure 1 checks if the current user can execute operation `destroy` with the subject being `article`. The operation is a symbol that matches an operation in the Ability object. The subject of the authorization check might be an object, a set of objects, or a class (denoting all objects of that class). The `can?` method queries the current Ability object to check if at least one `can` statement covers the operation symbol and the subject.

We extract `can?` checks into boolean expressions in our model as follows. We can extract the subject of the check directly using instrumented execution.

Next, we inspect the Ability class to identify all `can` statements that are relevant to the authorization check at hand. Let *ops* be the list of operations checked. For each role r, we identify all `can` statements that apply to role r and whose operations correspond to *ops* and whose expression type corresponds to the type of the subject of the `can?` check. Assuming that $e_1, e_2 \ldots e_k$ are expressions of these `can` statements, we extract `(inusergroup(r) and subject in union(`e_1`, ..., `e_k`))`. We generate this conjunction for each role and disjoin the results to get the complete boolean expression of the `can?` check.

For example, let us extract the `can?` check in line 26 of Figure 1. The extracted boolean expression should evaluate to true if and only if the current user is permitted to execute operation `destroy` on the `article` object.

For each role, we identify `can` statements that are relevant to the check and union their expressions if there are multiple. First, for the `admin` role, there is only one `can` statement that could permit this operation on an Article: the `can` statement in line 14. This `can` statement permits all operations on all objects, so its expression trivially covers all objects.

Then, for the `user` role, we again union the expressions of all `can` statements that might permit this operation. Again, there is only one `can` statement that could permit this operation, in line 17. Since there is only one such `can` statement, the union of all the expressions is equal to the expression of this `can` statement. This expression translates to CanCheck as `current_user.articles`.

At this point we have defined the set of objects each role is permitted to operate on. If every object we are checking authorization on is inside one of the above sets of objects, and the user has the appropriate roles, then the check passes. In our example, the resulting translation of this check is in lines 19-23 of Figure 3: the user is permitted to execute the `destroy` operation if he is an `admin` (line 20), or if he is a `nonadmin` (line 21) and if the article in question is a member of `current_user.articles` (line 22).

Automated checks.

CanCan can automatically generate access control checks using controller-level declarations that prepend authorization checks to actions. In our experience, most access control checks are created using these declarations. Since these checks are "automagical," we found bugs related to misuse or misunderstanding of these automated checks.

Since we use instrumented execution for model extraction, we do not need to treat automated checks differently from explicit checks. We extract automated checks by allowing CanCan to follow its own logic and heuristics to determine which checks need to be executed, then we extract these generated checks as they are executed.

4.4 Limitations

The extraction method we presented is not sound in general [4]. It extracts a CanCheck specification from a Rails application fully automatically and does so precisely in all applications we experimented on. However, considering the excessive flexibility Ruby allows and the amount of dynamic features libraries tend to employ, imprecision in extraction is possible. In addition, our tool relies on heuristics to extract user role information in case Devise is not used. Similarly, we assume that the Ability class constructor will investigate the role of the current user in branch conditions. Considering that the Rails community strongly promotes convention over configuration, basing heuristics on conventions often yields the correct result, and it always did in our experiments. Our tool lets developers manually alter the extracted model in case conventions are not followed.

5. AUTHORIZATION PROPERTIES

We want to use FOL theorem provers to verify that, regardless of the role of the current user, all objects created, deleted or read by any possible execution of an action are permitted for creation, deletion and reading, respectively. We generate a set of *authorization properties* to express this expectation.

An authorization property is a tuple $\langle a, op, c\rangle$ that defines the expectation that action $a \in A$ correctly enforces the access control policy with respect to the operation $op \in \{create, read, delete\}$ for all objects of type $c \in C$. We generate an authorization property for every action, every operation, and every class where the action might possibly execute that operation on objects of that class. For example, we will not generate an authorization property for creating Articles in an action that never creates an Article, as this authorization property is vacuously valid.

Our goal is to, given an authorization property $\langle a, create, c\rangle$, generate a formula that verifies whether the access control policy is correctly enforced with regards to this property. To do this we need to first express the set of objects created, deleted or read by an action in FOL. We also need to translate permits to FOL. With these two, we can express the expectation that all objects created, deleted or read by an action are accepted by at least one permit in FOL.

Create, Delete and Read Sets of an Action.

As a consequence of translating an action to FOL using existing techniques [4], we have access to a predicate $s_c(o)$ that determines whether object o of class c exists in the action's pre-state s. Similarly, we have access to a predicate $s'_c(o)$ that determines whether object o of class c exists in the action's post state s'. We can use these predicates to define new predicates $created_c$ and $deleted_c$ that identify objects of class c that have been created or deleted by an action:

$$\forall o \in O_c\colon created_c(o) \Leftrightarrow \neg s_c(o) \wedge s'_c(o)$$
$$\forall o \in O_c\colon deleted_c(o) \Leftrightarrow s_c(o) \wedge \neg s'_c(o)$$

In order to identify all objects read by an action, we look for objects stored in any variable whose name begins with `@`. In Rails, these variables are referred to by the view in order to synthesize the response. When translating an action to FOL, after static single assignment, every variable v corresponds to a predicate $v(o)$ that determines whether an object is stored in the variable. Formally, given an action a, class c, and a set of variables $v_1 \dots v_i \in V$ that are of type c and whose name starts with `@`:

$$\forall o \in O_c\colon read_c(o) \Leftrightarrow s'(o) \wedge (v_1(o) \vee \cdots \vee v_i(o))$$

Translation of Permits and Permit Acceptance.

Given a permit $p = \langle g, ops, e\rangle$, a specific operation op and state predicate $s(o)$, we can generate a predicate $p_{op,s}(o)$ that accepts objects o if and only if $p[op, s, o]$. This predicate should be satisfied by o if and only if three conditions hold: $op \in ops$, the current user has a role in g, and o belongs to the result of expression e in state s.

The first condition is the easiest to translate. If $op \notin ops$, this permit does not accept o. We can at this point stop translating $p_{op,s}(o)$ to first order logic and declare that it will accept no object o.

The second condition checks the role of the current user. In our FOL translation, each role $r \in \overline{R}$ corresponds to a predicate $r(o)$ where $o \in O_{c_U}$. Therefore, if $g = \{r_1 \dots r_k\}$, the second condition is $r_1(c_U) \vee \cdots \vee r_k(c_U)$.

Finally, we need to translate the expression e to FOL. All expressions e, in actions and permits alike, translate to formulas with a single free variable x (denoted as $f(x)$) such that $f(x)$ is satisfied by all x that correspond to objects that the expression evaluates to [4]. Therefore the third condition is directly defined as $e(o)$.

With that in mind, we can define the permit predicate $p_{op,s}$. If $op \notin ops$:

$$\forall o \in O_c\colon \neg p_{op,s}(o)$$

Otherwise:

$$\forall o \in O_c\colon p_{op,s}(o) \Leftrightarrow (r_1(o_U) \vee \cdots \vee r_k(o_U)) \wedge e(o)$$

Translation of Authorization Properties.

Finally we have all the tools we need to synthesize FOL formulas that correspond to authorization properties. In order to check whether an action is valid with regards to authorization property $\langle a, create, c\rangle$, we use the theorem prover to check whether the following formula is implied from the definition of the action. Let $P = \{p_1 \dots p_k\}$:

$$\forall o \in created_c(o)\colon p_{1\ create,s'}(o) \vee \cdots \vee p_{k\ create,s'}(o)$$

In other words, an action is valid with regards to authorization property $\langle a, create, c\rangle$ if and only if every object of class c created by a is accepted by at least one permit p: $p[create, s', o]$.

We similarly define the condition for whether an action is valid with regards to authorization properties $\langle a, read, c\rangle$ and $\langle a, delete, c\rangle$ as:

$$\forall o \in read_c(o)\colon p_{1\ read,s'}(o) \vee \cdots \vee p_{k\ read,s'}(o)$$
$$\forall o \in deleted_c(o)\colon p_{1\ delete,s}(o) \vee \cdots \vee p_{k\ delete,s}(o)$$

Note that, in accordance with our discussion in Section 3.2, the condition for $deleted_c(o)$ refers to the pre-state, while the conditions for $created_c(o)$ and $read_c(o)$ both refer to the post-state.

6. EXPERIMENTS

To evaluate CanCheck, we conducted experiments on a set of Rails applications. We looked for largest Rails applications we could find that are supported by the current implementation of the extraction module. Our implementation is restricted to applications that: 1) use Rails 3.2 or up to 4.2; 2) use CanCan or CanCanCan for authorization using hash-based conditional authorization; 3) use ActiveRecord as the Object-Relational-Mapping (ORM).

In order to find these applications we examined two collections of Rails applications: the list of 25 most starred open-source Rails applications on github according to the OpenSourceRails.com website [32] and a compilation of open source Rails applications categorized by domain [1]. In total we found 5 applications that satisfy all constraints listed above: CoRM [10] (a customer relationship manager), Kandan [26] (an online chat service), Quant [34] (a personal health tracker), SprintApp [36] (project manager and time tracker), and Trado [38] (E-commerce platform).

In addition we included 6 applications were used to evaluate previous work [29]. These are small and tutorial applications. We included only 6 for technical reasons: these applications were written before Rails 3.2 and used library versions that are no longer available. We managed to upgrade these 6 applications to 3.2 in working order.

Table 2 summarizes the results of our experiments for all applications. We obtained these results on a computer with an Intel Core i5-2400S processor and 32GB RAM, running 64bit Linux. Memory consumption never exceeded 200Mb. The memory and time performance indicates that CanCheck is usable on commonly available hardware.

As we described above, the first phase of our approach is instrumented execution for extracting CanCheck specifications from the given applications. During the extraction phase, in several cases we had to manually alter implementation of a method when the developers deviated from the Rails coding style. These modifications were necessary only in applications that were seemingly designed by developers not versed in Rails coding style (Trado and CoRM). In addition we had to exclude one action of CoRM which is used for logging in. This action attempted to communicate with a third party server over a network during extraction and was blocking the process. The column labeled "Extraction Time (sec)" presents the time it took to completely extract the model from the application.

As described in Section 5, for each extracted action, for each class and for each operation done on objects of this class (read, create or delete), CanCheck generates an authorization property and translates it to a FOL theorem that is examined by theorem provers. The "Authorization Properties" column in Table 2 represents the number of generated authorization properties. Whenever access control is proven to be invalid for a given authorization property, CanCheck outputs a human readable message that describes the action, the unauthorized operation, and the type of the object that this unauthorized operation is executed on. When access control is proven to be valid, CanCheck outputs a similar human readable message that confirms validity for an action, operation and object type.

CanCheck generates formulas both for Z3, an SMT solver, and Spass, a FOL theorem prover. We express our FOL theorems in SMT using problem group UF, which includes free quantification, free sorts and uninterpreted functions. Z3 checks satisfiability, so when CanCheck constructs a formula to be sent to Z3, if a satisfying model exists for the formula, then there exists an execution of the application that violates the access control policy. If, on the other hand, Z3 reports that the formula is unsatisfiable, then we can conclude that no execution that violates the access control policy exists. Spass, on the other hand, checks whether a conjecture implies from a set of axioms. This conjecture is the authorization property itself. If Spass reports that the conjecture implies from the axioms, then we can conclude that no execution that violates the access control policy exists. However, if Spass reports that the conjecture does not always imply from the axioms, then we can conclude that an execution that violates the access control policy exists.

Note that FOL is undecidable in general and CanCheck generates formulas without restrictions on quantification nesting, without a bound on the number of arguments for predicates, and without a bound on the domains. So, the formulas generated by CanCheck are not in a known subset of FOL that is decidable. This implies that Z3 and Spass may not be able to produce a conclusive result for some of the formulas generated by CanCheck.

CanCheck uses Z3 and Spass concurrently, waiting for either theorem prover to produce a result, after which the other prover is terminated. In our experiments we observed that Z3 is faster and is more likely to report conclusive results for the formulas CanCheck generates. In fact, Z3 always produced results first, and produced conclusive results for every set of formulas generated by CanCheck. As such, Table 2 reports performance results for Z3 only. Column "Avg. Time per Check" shows the average amount of verification time per authorization property. Verification time was about a second or less for all applications other than CoRM. For CoRM average verification time was about 16 seconds per authorization property. This is likely due to the large number of roles and permits in the CoRM data model.

In Table 2, the "Correct" column lists the number of cases where the access control policy was correctly enforced with respect to an action, an operation and a class type. The "Bugs" column lists the number of instances where a user is allowed to execute operations that are not allowed by the policy. For example, in CoRM, any user is allowed to delete entire chunks of the database that administrators have imported into the database. Non-admins are explicitly forbidden from doing so in Ability object, however, the policy is incorrectly enforced. Similarly, in SprintApp, anyone can change user data of any other user, including their password. This is explicitly forbidden by the policy, and the option to do so is not visible in the user interface, but the policy is incorrectly enforced and it is possible to target these restricted pages by manually entering a url in a web browser.

The "Policy errors" column lists the number of theorems that certain possible operations are not covered by the access control policy. These errors were caused by the policy not mentioning certain operations, and in all cases, the policy was not checked in the corresponding action. When we investigated these applications manually we concluded that it is OK for these applications to allow these operations, and, they implicitly do so by not checking the policy. We cate-

Table 2: Experiments application summary.

Application	KLoC	Extraction Time (sec)	Actions	Roles	Permits	Authorization Properties	Avg. Time per Check (sec)	Correct	Bugs	Policy Errors	False Positives
CoRM	16.8	45.05	173	5	53	284	16.04	223	26	32	3
Kandan	9.6	7.08	35	2	8	20	0.07	12	0	8	0
Quant	14.6	4.84	44	1	9	51	0.09	51	0	0	0
SprintApp	8.6	17.65	127	2	15	110	0.28	103	4	3	0
Trado	53.9	13.66	75	2	36	55	1.26	15	0	40	0
Avare	1.2	3.51	39	6	12	25	0.10	2	23	0	0
Communautaire	0.8	3.54	41	2	10	48	0.07	46	2	0	0
Illyan	1.5	3.97	24	2	4	23	0.03	23	0	0	0
bootstrap	0.8	4.32	4	1	2	4	4.42	3	0	1	0
s2l	1.3	5.48	50	3	15	120	0.56	55	65	0	0
wm-app	2.4	2.48	94	2	36	123	0.00	123	0	0	0

gorize these errors separately as they do not cause a breach of privacy or elevate user privilege, however, they point out a deficiency in the access control policy itself. For example, Kandan's policy does not explicitly refer to Attachment and Activity classes, nor does it ever check permissions related to these two classes.

Finally, there are three false positives our tool reported in SprintApp. All three correspond to the same faulty behavior, as SprintApp automatically creates an administrator user account if there are no users in the database. This implies that anyone could potentially create a user account that would belong to someone else, and the policy only allows operations on one's own user object. This is a clear violation of the policy. However, this bug does not elevate ones privilege nor expose private information since there are no users in the database when this behavior occurs.

6.1 Discovered Bugs and Policy Errors

There are 11 actions in CoRM that violate the access control policy in a total of 26 different ways. In AccountsController and ContactsController, `destroy` actions can be used to, as a side-effect, delete Tasks and Aliases. The policy explicitly forbids the superuser role from deleting these objects, yet superusers are allowed to execute these actions. Furthermore, none of the actions in the Emails controller ever checks policies, accounting for several access control violations. The same holds for the Imports controller mentioned above.

Our tool found 4 access control violations in SprintApp, all ultimately caused by poor communication between CanCan and ActiveAdmin, a library SprintApp uses for action generation. In SprintApp, ActiveAdmin pre-loads data relevant to the action, and CanCan subsequently authorizes operations on this data. However, in four actions, CanCan fails to capture the pre-loaded objects and proceeds to authorize the action ignoring pre-loaded objects. ActiveAdmin then proceeds with the pre-loaded objects that were not correctly authorized. Using this exploit, any user can access the private information of any user and change their information including their password. What is specifically interesting about this bug is that it is not caused by a developer error, but by a configuration error. The action and the controller source code both look like they authorize data correctly.

There is a total of 8 policy violations in Kandan. The policy does not refer to Attachment and Activity classes, implicitly disallowing operations on them. However, the policy is never checked with regards to these classes. Semantically all operations should be allowed on them, and as such, this represents a deficient policy.

Quant uses a simple access control policy. There is only one user group, and every user can access only their own objects. This policy is simple enough to correctly be enforced using automated checks that are present in every controller. As such, this application has no access control violations.

Trado is an e-commerce platform, letting users browse items, add them to their carts and place orders. However, the policy only mentions a few of these classes, most notably not mentioning Orders and order-related classes at all. In addition, it attempts to enforce this policy in only a few controllers, and as such is enforced inconsistently. For the sake of consistency we categorized these errors as policy errors as the policy is too sparse to be usable, however, even a more complete policy wouldn't be correctly enforced on the application as-is. The policy and the enforcement of the policy are both amateurish, but show that our tool is useful for detecting basic errors in addition to esoteric ones.

The other 6 applications have numerous access control bugs in them, but these applications are typically developed by beginners and abandoned mid-development. Illyan, the only application still in development, has no bugs or policy errors. Wm-app also has no bugs or policy errors, and is a nice example of how automated access control can be used correctly and uniformly. All other applications had either policy errors or invalid access control enforcement.

7. RELATED WORK

Access control in Ruby on Rails has been examined before [31]. This work provides an interactive tool that let the developer traverse the tree of all data exposed by an application and interactively generate a desired policy, while noticing exceptions to this policy. Our method is fully automated. Their model represents actions as direct transitions from the pre- to the post-state, whereas we represent them in an imperative model language. They use Alloy to power verification while we use FOL theorem provers.

RubyX [9] is a tool for symbolic execution Rails that can be used to find access control bugs. It uses manually written scripts, each of which has to setup a database with symbolic values, execute an action, manually capture relevant output of the action, and check whether specific post-conditions hold. We require no manual effort from the developer both in terms of specifying expectations of correctness and scenarios under which these expectations should be met. Our extraction method does not rely on SMT solvers and a custom symbolic runtime. Furthermore, they use DRails [20] to make specific usages of Rails code explicit, whereas we capture metaprogramming natively.

SafeWeb [21] is a Rails middleware tool for access control. It stands as a layer between the web application and the data. At runtime, it tracks data items and propagates associated permissions in a way that is similar to dynamic taint analysis, raising errors if a user gains access to restricted information. We statically ensure that access control is implemented correctly, while SafeWeb incurs a runtime overhead. In addition, in order to use SafeWeb, existing Rails applications to be fundamentally overhauled. Our goal was to examine existing applications.

RailroadMap [29] is an automated tool for verification of access control in Rails using CanCan and Pundit. The similarities between our approach and theirs end at trying to achieve the same goal. Their program analysis is limited to parsing a few specific Rails files and examining the AST, not even taking file dependencies into account. They expect the Ability class to be declared in a limited manner, not allowing `elsif` branches or branch nesting or method calls in the constructor. They naively assume that not showing url links to actions is sufficient to prevent unrestricted execution of actions. Finally, they evaluated their method on small applications: all but a few had a single developer and were abandoned in weeks. We could not directly compare our results to theirs because their reported results are not specific enough to compare: they report the number of bugs but no specific description of bugs. In addition, we are confused by them reporting access control bugs in two applications (Artdealer and shiroipantsu) that, as far as we can see, had never in their history of development used access control.

Access control bugs are sometimes found with techniques not specifically tailored for finding access control bugs [30]. These methods typically require more effort than our automated method and may miss bugs.

The contributions in this paper extend our previous work on data model verification [4, 5, 6]. This work focuses on verifying whether all actions in a Rails application preserve a set of user specified invariants that refer to data integrity. We do this by extracting a data model from a Ruby on Rails application, translating the model to FOL, and using first order logic theorem provers to ensure that no execution of an action can violate a manually specified invariant. We use a similar approach for extraction of CanCheck specifications and translation to FOL. However, access control models are significantly different than data models used in these earlier approaches. Access control models, besides access control related constructs, support conditional branches and exceptions. The formal modeling of access control in Rails applications, inference of access control properties and the conditions under which access control is deemed correct or incorrect are all original contributions of this paper. Moreover, the approach presented in this paper does not require manual specification of invariants.

Our basic access control model maps each user, operation and object, to a boolean [27]. There are more elaborate access control systems [14, 22]. Our model can be considered a simplified version of role-based access control (RBAC). Our model of access control reflects the access control mechanisms used in real world Rails gems like CanCan, CanCanCan and Pundit.

There is an extensive body of research that focuses on verification of access control policies [12, 23], as well as aiding the creation of access control policies [15, 13]. This work describes and investigates policies specified using standards such as XACML. In our work, rather than focusing on policy correctness in isolation, we are focusing on inconsistencies between policy specification and policy enforcement. In most cases such inconsistencies point to bugs in policy enforcement, even though policy specification might itself be deficient. In addition, we focus on verifying access control policies in Rails applications, and XACML is not used in any Rails application we came across.

The problem of access control policy enforcement has been tackled before. For example, there exists work that checks whether a user of a particular role could access a restricted webpage [37]. Their work requires manual specification of user roles and categorizing pages based on which roles can access them. We automatically extract this information from existing code. Instead of just ensuring authorized actions are accessible, we also ensure that all operations executed by accessing these actions conform to the policy. As an other example, access control can be verified on Java objects, given an appropriate policy [2]. Our problem domain, level of automation, and approach are all different.

Alloy [24, 25] is a formal language for specifying object oriented data models and their properties. Alloy Analyzer is used to verify properties of Alloy specifications. Alloy Analyzer uses SAT-based bounded verification techniques as opposed to the FOL based unbounded verification technique we use. DynAlloy [17, 18] is an extension of Alloy that adds support for actions, achieved by translating dynamic specifications onto Alloy. While they talk about actions in their work, those actions do not correspond to actions in web applications. Instead, they are more similar to statements in programming languages [19]. Their work has focused on verification of data structures, not behaviors in data models of web applications.

Verification of software using theorem provers has been explored before in projects such as Boogie [3], Dafny [28] and ESC Java [16]. These projects focus on languages such as C, C#, and Java, and typically require user guidance in the form of explicit pre- and post-conditions, explicit data structure constraints, and loop invariants. Our method is fully automated and focuses on access control in modern web applications.

8. CONCLUSIONS AND FUTURE WORK

We presented a technique for access control verification in Rails applications. We extract a formal model from the application that captures the schema, actions that update the state, user roles, the access control policy and the way the access control policy is enforced by the application. We automatically generate authorization properties from this formal model, which, if violated, point to access control bugs. We implemented this approach in a tool called CanCheck that can automatically check for access control bugs in Rails applications written using access control gems CanCan and CanCanCan. Using CanCheck we found numerous real bugs in multiple open source web applications, as well as deficiencies in access control policies.

Our results indicate that automated theorem provers can effectively reason about access control violations in real applications when the access control model is extracted appropriately and access control properties are encoded in logic. We would like to investigate Rails language extensions that support direct semantic encoding in logic, which would eliminate the need for extraction and logic translation, and enable direct application of the verification technology to web application development.

9. REFERENCES

[1] ekremkaraca/awesome-rails: A collection / list of awesome projects, sites made with Rails.

[2] A. Ali and M. Fernández. Static enforcement of role-based access control. In M. H. ter Beek and A. Ravara, editors, *Proceedings 10th International Workshop on Automated Specification and Verification of Web Systems, WWV 2014, Vienna, Austria, July 18, 2014.*, volume 163 of *EPTCS*, pages 36–50, 2014.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[4] I. Bocic and T. Bultan. Inductive verification of data model invariants for web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, May 2014.

[5] I. Bocic and T. Bultan. Coexecutability for efficient verification of data model updates. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 744–754, 2015.

[6] I. Bocic and T. Bultan. Efficient data model verification with many-sorted logic. In *30th IEEE/ACM International Conference on Automated Software Engineering ASE 2015, Lincoln, Nebraska, USA, November 9-13, 2015*, 2015.

[7] ryanb/cancan • GitHub, Nov. 2015. https://github.com/ryanb/cancan.

[8] CanCanCommunity/cancancan • GitHub, Nov. 2015. https://github.com/CanCanCommunity/cancancan.

[9] A. Chaudhuri and J. S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 585–594, 2010.

[10] CoRM - Enfin une solution CRM simple et puissante., Jan. 2016. http://www.corm.fr/.

[11] devise | RubyGems.org | your community gem host, Sept. 2013. http://rubygems.org/gems/devise.

[12] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[13] S. Egelman, A. Oates, and S. Krishnamurthi. Oops, i did it again: Mitigating repeated access control errors on facebook. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2295–2304, New York, NY, USA, 2011. ACM.

[14] D. Ferraiolo and R. Kuhn. Role-based access controls. *Proc. of 15th NIST-NSA National Computer Security Conference*, 1992.

[15] K. Fisler and S. Krishnamurthi. A model of triangulating environments for policy authoring. In J. B. D. Joshi and B. Carminati, editors, *SACMAT 2010, 15th ACM Symposium on Access Control Models and Technologies, Pittsburgh, Pennsylvania, USA, June 9-11, 2010, Proceedings*, pages 3–12. ACM, 2010.

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[17] M. F. Frias, J. P. Galeotti, C. L. Pombo, and N. Aguirre. Dynalloy: upgrading alloy with actions. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 442–451, 2005.

[18] M. F. Frias, C. L. Pombo, J. P. Galeotti, and N. Aguirre. Efficient analysis of dynalloy specifications. *ACM Transactions on Software Enginnering Methodology*, 17(1), 2007.

[19] J. P. Galeotti and M. F. Frias. Dynalloy as a formal method for the analysis of java programs. In *Software Engineering Techniques: Design for Quality, SET 2006, October 17-20, 2006, Warsaw, Poland*, pages 249–260, 2006.

[20] J. hoon (David) An, A. Chaudhuri, and J. S. Foster. Static typing for ruby on rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 590–594, 2009.

[21] P. Hosek, M. Migliavacca, I. Papagiannis, D. M. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch. Safeweb: A middleware for securing ruby-based web applications. In *Middleware 2011 - ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings*, pages 491–511, 2011.

[22] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (abac) definition and considerations. *NIST Special Publication*, 800:162, 2014.

[23] G. Hughes and T. Bultan. Automated verification of access control policies using a SAT solver. *STTT*, 10(6):503–520, 2008.

[24] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Enginnering and Methodology (TOSEM 2002)*, 11(2):256–290, 2002.

[25] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, Cambridge, Massachusetts, 2006.

[26] kandanapp/kandan, Sept. 2013. http://github.com/kandanapp/kandan.

[27] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, Jan. 1974.

[28] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 348–370, 2010.

[29] S. Munetoh and N. Yoshioka. Model-assisted access control implementation for code-centric ruby-on-rails web application development. In *2013 International*

Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013, pages 350–359, 2013.

[30] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *Proceedings of the ACM SIGSOFT 20th Int. Symp. Foundations of Software Engineering (FSE 2012)*, pages 60:1–60:11, 2012.

[31] J. P. Near and D. Jackson. Derailer: interactive security analysis for web applications. In I. Crnkovic, M. Chechik, and P. Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 587–598. ACM, 2014.

[32] Open Source Rails, Jan. 2016. http://www.opensourcerails.com.

[33] GitHub - elabs/pundit: Minimal authorization throught OO design and pure Ruby classes, Jan. 2016. https://github.com/elabs/pundit.

[34] jdjkelly/quant - GitHub, Jan. 2016. https://github.com/jdjkelly/quant.

[35] Rails Routing from the Outside In - Ruby on Rails Guides, Jan. 2016. guides.rubyonrails.org/routing.html#crud-verbs-and-actions.

[36] macfanatic/SprintApp, Sept. 2014. https://github.com/macfanatic/SprintApp.

[37] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[38] Jellyfishboy/trado - Ruby- GitHub, Jan. 2016. https://github.com/Jellyfishboy/trado.