Finding Security Bugs in Web Applications using a Catalog of Access Control Patterns

Finding Security Bugs in Web Applications using a Catalog of Access Control Patterns

Joseph P. Near
University of California, Berkeley
jnear@berkeley.edu

Daniel Jackson
Massachusetts Institute of Technology
dnj@mit.edu

ABSTRACT

We propose a specification-free technique for finding missing security checks in web applications using a *catalog of access control patterns* in which each pattern models a common access control use case. Our implementation, SPACE, checks that every data exposure allowed by an application's code matches an allowed exposure from a security pattern in our catalog. The only user-provided input is a mapping from application types to the types of the catalog; the rest of the process is entirely automatic. In an evaluation on the 50 most watched Ruby on Rails applications on Github, SPACE reported 33 possible bugs—23 previously unknown security bugs, and 10 false positives.

Categories and Subject Descriptors

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

Keywords

web application security, access control, bug finding

1 Introduction

Web application security holes represent a large and growing class of bugs. Programming frameworks and static analysis tools have begun to address bugs that violate generic, cross-application specifications (for example, injection, cross-site scripting and overflow vulnerabilities). However, application-specific bugs (like missing security checks) have received less attention. Traditional solutions, such as verification and dynamic policy-enforcement techniques, ask the user to write a specification of the intended access control policy.

We propose a specification-free technique for finding application-specific security bugs using a *catalog of access control patterns*. Each pattern in our catalog models a common access control use case in web applications. Our catalog is based on the assumption that developers usually intend for a particular pattern to be applied uniformly to all uses of a given resource type—an assumption supported by our experience with real-world applications.

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

We have implemented this technique in SPACE (Security PAttern CheckER). SPACE uses symbolic execution to extract the set of *data exposures* [25] from the source code of a Ruby on Rails application. The constraints associated with these exposures and the user-provided mapping are passed through a constraint specializer, which re-casts the constraints in terms of the types in our pattern catalog. Then, SPACE uses the Alloy Analyzer to perform automatic bounded verification that each data exposure allowed by the application is also allowed by our catalog.

We applied SPACE to the 50 most watched open-source Rails applications on Github. We found that 30 of the 50 implement access control, and SPACE found bugs in 8 of those 30—a total of 23 unique bugs. Both the symbolic execution and bounded verification steps of our technique scale to applications as large as 45k lines of code—none of our analyses took longer than 64 seconds to finish.

This paper makes the following contributions:

- A highly automated technique for finding security bugs based on the idea of matching extracted access patterns to known safe patterns;
- An initial catalog of common security patterns, based on those we encountered in an analysis of real-world applications;
- An open-source implementation,[1] SPACE, that has been used to find previously unknown bugs in open-source Rails applications.

In section 2, we describe SPACE from the user's perspective and demonstrate its use to find a bug. In section 3, we formalize our catalog of patterns and the technique used to check them against an application. Section 4 describes our implementation of SPACE, and section 5 details our evaluation of its effectiveness. In section 6, we discuss related work, and section 7 contains our conclusions.

[1] http://www.cs.berkeley.edu/~jnear/space

ICSE '16, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-3900-1/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884836

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

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

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

Figure 1: Controller Code for MediumClone

2 Description & Example

Space (Security PAttern CheckEr) finds security bugs in Ruby on Rails[2] web applications, and requires only that the user provide a mapping from application-defined resource types to the object types of the standard role-based model of access control (RBAC) [30, 15]. Space is otherwise completely automatic: it analyzes the target application's source code and returns a list of bugs.

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

2.1 Example Application: MediumClone

MediumClone[3] is a simple blogging platform designed to mimic the popular web site Medium. Using MediumClone, users can read posts and log in to create new posts or update existing ones. The site requires users to sign up and log in before writing posts, and prevents users from modifying others' posts.

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

Figure 1 contains a part of the controller code for MediumClone's UserController, which provides a RESTful API for user profiles. Following the REST convention, the show action is for displaying profiles, the edit action displays an HTML form for editing the profile, and submitting that form results in a POST request to update action, which actually performs the database update (using update_attributes). The call to before_filter installs a *filter*—a procedure that runs before the action itself—for the show and edit actions. The filter checks that the logged-in user has permission to perform the requested action, and redirects them to an error page if not. This use of filters to enforce security checks is common in Rails applications, and helps to ensure that checks are applied uniformly.

MediumClone's Security Bug. The code in Figure 1 fails to apply the correct_user filter to the update action, so an attacker can send an HTTP POST request directly to the update URL of the MediumClone site to update *any* user profile. The filter *is* properly applied to the edit action. In our experience, this kind of mistake is common: the developer assumes the user will use the interface provided by the site (in this case, the edit page), and will not craft malicious requests to constructed URLs. Developers often fail to consider alternative paths not accessible through the interface, and therefore omit vital security checks.

Finding the Bug Using Space. Space compares the checks present in the application code against its built-in catalog of common security patterns. When the application code is missing security checks required by the patterns, Space reports a bug in the code.

Space's security patterns are formalized in terms of role-based access control (RBAC), which defines sets of *users, roles, operations, objects*, and *permissions*. Space's formal models of security patterns enforce security checks by constraining relations between these sets. In order to check these patterns against a piece of code, the user provides a mapping of the application's resources to the sets of objects defined in the RBAC model. MediumClone's User type represents the *User* type in the RBAC pattern, and Posts are special RBAC objects that have an owner. The *owns* relation is defined by our model of ownership, and user: owns means that the user field of a Post specifies its owner. The user provides this mapping for MediumClone as follows:

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

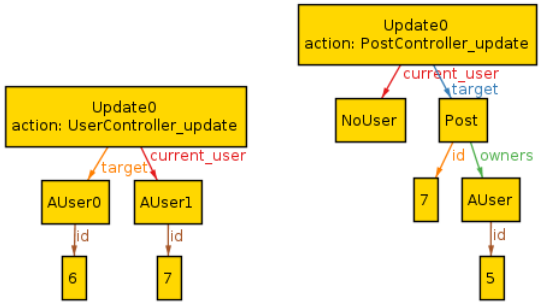
Space requires the mapping above and MediumClone's source code—it needs no further input or guidance from the user. The mapping provided by the user translates between the RBAC objects constrained by the pattern catalog and the resource types defined in the application code. Space uses this mapping to specialize the constraints derived from the checks present in the code to the set of RBAC objects, so that the two sets of security checks can be compared.

Space compares the specialized constraints against the constraints required by the pattern catalog. When Space finds a missing check, it builds a counterexample demonstrating the security vulnerability caused by the missing check. The counterexample is specific to the application code under analysis: it involves resource types defined in the code, rather than RBAC concepts.

For MediumClone, Space produces the counterexample shown in Figure 2(a). The "Update0" box indicates that an update is possible on the profile of "AUser0" (the "target"—a

[2]http://rubyonrails.org/

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



(a) (b)

Figure 2: SPACE Counterexample Showing Medium-Clone Security Bugs: (a) user can update another user's profile; (b) unauthenticated user ("NoUser") can update any post

renaming of "User" to avoid clashes) by the distinct "AUser1" (the "current_user", or currently logged-in user). This counterexample demonstrates a user updating another user's profile using the update action—a situation that is not captured by our security pattern catalog. The *User Profile Pattern* explicitly requires that users can update their own profiles, but no pattern exists that allows updating another user's profile. The bug can be fixed by adding :update to the list of actions for which the : correct_user filter is applied.

More Bugs in MediumClone. Fixing the bug and running SPACE again yields a new counterexample, shown in Figure 2(b). This counterexample suggests that Post resources in MediumClone can be updated not only by users who did not create them, but by users who are not logged in ("NoUser") at all! The code for PostController checks that the user is logged in before performing the new, create, and edit actions, but omits this check for the update action:

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

This bug is interesting because the intended security policy—that users only modify their own posts—is forgotten across the entire application codebase. This mistake exemplifies a class of bugs that are difficult to detect with testing or with our previously-developed consistency checker, Derailer [25].

3 Formal Model

This section formalizes our catalog and SPACE's method for comparing it to a web application's code. The first step is to derive a list of *data exposures* from the application code, representing the different ways users of the application can read or update data stored in the database. Second, SPACE uses the *mapping* defined by the user to specialize each exposure's constraints to the objects constrained by the catalog. Finally, SPACE verifies that each data exposure allowed by the application code is also allowed by the catalog.

3.1 Web Applications

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

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

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

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

The *exposures* relation characterizes the conditions (ϕ) under which the application code allows a particular resource to be read or written. The formula $(db, req, res) \in \mathit{response} \Rightarrow \phi$ means that if the application exposes resource res, it *must* be under conditions ϕ.

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

SPACE builds the *exposures* relation by symbolically evaluating each action of the web application under analysis, passing symbolic values for the database and user-supplied parameters. SPACE enumerates the symbolic values that can appear on a rendered page and constructs an exposure for each one. These exposures have the form $(req, \phi, \mathsf{read}, res)$, where req is the original symbolic request, ϕ is the path condition derived from the symbolic execution, and res is the symbolic expression representing the exposed resource. Similarly, when SPACE finds a database update, it constructs an exposure of the form $(req, \phi, \mathsf{write}, res)$, where req is the request, ϕ is the current path condition, and res is a symbolic expression representing the resource being updated.

Example. As a running example, we take MediumClone's UserController bug from section 2. For this controller's buggy update action, SPACE generates the exposure:

$$(\mathsf{update}(\mathit{user.profile}), \mathit{true}, \mathsf{write}, \mathit{user})$$

meaning that when the application receives a request to update a user profile of *user*, it performs the modification without enforcing any constraints. The exposure that *correctly* enforces the desired security policy (by enforcing that the modified profile is the current user's) is instead:

$$(\mathsf{update}(\mathit{user.profile}), \mathit{user} = \mathsf{current\_user}, \mathsf{write}, \mathit{user})$$

3.2 Role-Based Access Control

As a basis for representing security policies, we adopt the role-based access control model of Sandhu et al. [30] (specifically, $RBAC_0$). The standard model of role-based access control consists of sets *Users*, *Roles*, *Permissions*, and *Sessions*, over which the following relations are defined to assign permissions and users to roles:

$$\begin{aligned} \mathit{permission}_a &\subseteq \mathit{Permissions} \times \mathit{Roles} \\ \mathit{user}_a &\subseteq \mathit{Users} \times \mathit{Roles} \end{aligned}$$

We adopt the extension of this model by Ferraiolo et al. [15] with *Objects* and *Operations*, where *Objects* contains the

targets of permissions (which will correspond to the web application's resources) and *Operations* (for our applications) contains the operation types read and write already used in the definition of the *exposures* relation. This model defines *Permissions* to be a set of mappings between operations (either read or write) and objects allowed by that permission:

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

3.3 Security Pattern Catalog

The patterns in our catalog constrain the relations of the generic RBAC model to specifically allow certain behaviors. Each one of these pattern definitions corresponds to a standard use case for resources in web applications; in effect, the patterns "whitelist" common kinds of correct data accesses.

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

$$owns \quad \subseteq \quad Users \times Objects$$

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

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

$$PublicObjects \quad \subseteq \quad Objects$$
$$\forall u, r, o, p \,.\, o \in PublicObjects \wedge (u, r) \in user_a \Rightarrow \\ ((\mathsf{read}, o), r) \in permission_a$$

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

$$logged\_in \quad \subseteq \quad Users$$

$$\forall u, r, o, op, p \,. \\ (u, r) \in user_a \wedge ((op, o), r) \in permission_a \Rightarrow \\ (op = \mathsf{read} \wedge o \in PublicObjects) \vee \\ u \in logged\_in$$

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

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

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

$$profile \subseteq Users \times Objects$$
$$\forall u, p \,.\, \exists r \,.\, (u, r) \in user_a \wedge (u, p) \in profile \wedge \\ ((\mathsf{write}, p), r) \in permission_a$$

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

$$\mathsf{Admin} \quad \in \quad Roles$$
$$\forall o \,.\, ((\mathsf{read}, o), \mathsf{Admin}) \in permission_a \wedge \\ ((\mathsf{write}, o), \mathsf{Admin}) \in permission_a$$

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

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

3.4 Mapping Application Resources to RBAC Objects

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

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

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

We also need to provide definitions from the application for the new concepts introduced in our pattern definitions: *PublicObjects*, *PermissionObjects*, and the *owns* and *permits*

relations. The *map* relation can be used to define public and permission objects, but we must define a separate mapping from field names of resources to the corresponding relations they represent in our security patterns. We use map_{fields} for this purpose.

$$\begin{array}{rcl} FieldNames & = & \textit{application-defined field names} \\ RelationNames & = & \{\mathsf{owns}, \mathsf{permits}, \mathsf{logged\_in}, \\ & & \mathsf{object\_roles}, \mathsf{user\_roles}\} \\ map_{fields} & \subseteq & FieldNames \times RelationNames \end{array}$$

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

$$session \quad \subseteq \quad Requests \times Users$$

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

Example. Based on the mapping provided for MediumClone in section 2, SPACE populates the mapping relations as follows:

$$\begin{array}{rcl} \tau_{Resource} & = & \{\mathsf{MUser}, \mathsf{MPost}\} \\ map & = & ((\mathsf{MUser}, \mathsf{User}), \\ & & (\mathsf{MPost}, \mathsf{OwnedObject})) \\ FieldNames & = & \{\mathsf{author}, ...\} \\ map_{fields} & = & ((\mathsf{author}, \mathsf{owns})) \end{array}$$

3.5 Finding Pattern Violations

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

To check exposure (req, ϕ, op, res):

1. Build a *specialized constraint* ϕ' from ϕ by substituting RBAC objects for application resources using the *map* relation supplied by the user.
2. Determine whether or not some pattern allows the exposure by attempting to falsify the formula:

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

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

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

- Replace each reference res to an application resource in ϕ with an expression $\{\tau | (res, \tau) \in map\}$ representing the corresponding set of possible RBAC objects.
- Replace each field reference $o.fld$ in ϕ with an expression $\{o' | (fld, r) \in map_{fields} \wedge (o', o) \in r\}$ representing a reference to the corresponding relation defined by our security patterns.

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

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

- Let ϕ' be the result of performing substitution on ϕ using the user-supplied *map*.
- Let *obj* be the RBAC objects corresponding to the application resource res, so that if $(res, o) \in map$ then $o \in obj$.
- Let u be the current user, so that $(req, u) \in session$.
- Let $\mathcal{C}$ be the conjunction of all of the constraints defined by our pattern catalog.
- Then the following formula must hold:

  $\mathcal{C} \wedge \phi' \Rightarrow \exists r.\ (u, r) \in user_a \wedge ((op, obj), r) \in permission_a$

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

Example. To check the buggy exposure in our running example, SPACE sets $op = \mathsf{write}$, $\phi' = true$, $obj = user$, and $u = \mathsf{current\_user}$. Then, SPACE attempts to falsify the formula:

$$\begin{array}{l} \mathcal{C} \wedge\ true \Rightarrow \exists r.\ (\mathsf{current\_user}, r) \in user_a \wedge \\ \qquad\qquad ((\mathsf{write}, user), r) \in permission_a \end{array}$$

This formula is falsified by any assignment under which $user \neq \mathsf{current\_user}$ and $user_a$ and $permission_a$ do not give the current user permission to update *user*'s profile, since the user profile pattern requires only that the current user be able to update his or her *own* profile. An example of such an assignment is the one that SPACE reports for this bug, shown earlier in Figure 2.

The exposure corresponding to the *correct* security policy, in contrast, yields $\phi' = (user = \mathsf{current\_user})$, so the formula to be falsified is:

$$\begin{array}{l} \mathcal{C} \wedge\ user = \mathsf{current\_user} \Rightarrow \\ \quad \exists r.\ (\mathsf{current\_user}, r) \in user_a \wedge \\ \quad ((\mathsf{write}, user), r) \in permission_a \end{array}$$

Since this formula requires that the profile being updated is the current user's, any assignment that falsifies it also falsifies the user profile pattern. Substituting the (specialized) user profile pattern for $\mathcal{C}$ results in a tautology:

$$\begin{array}{l} \exists r\ .\ (\mathsf{current\_user}, r) \in user_a \wedge \\ \quad ((\mathsf{write}, \mathsf{current\_user}), r) \in permission_a \wedge \\ user = \mathsf{current\_user} \Rightarrow \\ \quad \exists r.\ (\mathsf{current\_user}, r) \in user_a \wedge \\ \quad ((\mathsf{write}, user), r) \in permission_a \end{array}$$

4 Implementation

Our implemented tool, SPACE (Security PAttern CheckER), finds security bugs in Ruby on Rails web applications. Figure 3 contains an overview of SPACE's architecture. SPACE extracts the data exposures from an application using symbolic execution, specializes the constraints on those exposures to the types of role-based access control using the mapping provided by the user, and exports the specialized constraints to an Alloy specification. Then, SPACE uses the

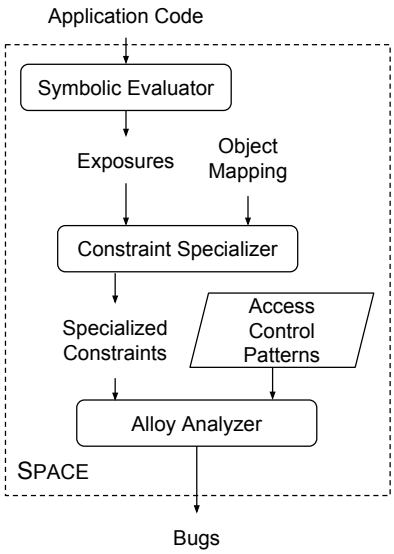


Figure 3: Summary of the Architecture of SPACE

Alloy Analyzer—an automatic bounded verifier for the Alloy language—to compare the specialized constraints to our pattern catalog (which is also specified in Alloy).

4.1 Symbolic Execution

Implementing a standalone symbolic evaluator is difficult for any language, but especially so for Ruby. Ruby is dynamic, has no formal specification, and changes quickly. Since Rails is a large library using nearly all of Ruby's features, however, a symbolic evaluator for Ruby on Rails applications must implement all of those features too.

Rather than attempt to build a standalone symbolic evaluator, we used a symbolic execution *library* to coerce Ruby's standard interpreter into performing symbolic execution. We developed this approach for Rubicon [24], which performed bounded verification of user-defined properties on Rails applications, and Derailer [25], which interacts with the user to find mistakes in the uniform application of a security policy.

We present a brief explanation of the library-based approach to symbolic execution here for completeness. Detailed descriptions of our approach to symbolic execution is available as part of our earlier work; a formal description of the technique, including an embedding in a simplified Ruby-like language with a formal semantics and a proof that it produces the same results as the standard approach to symbolic execution, is available in [26].

Symbolic Values. Our library defines Ruby classes for symbolic objects and expressions. When a program invokes a method on a symbolic object, Ruby's method_missing feature is used to produce a new symbolic expression.

```
def method_missing(m,*a)
  Exp.new(m, [self]+a)
end
def ==(o)
  Exp.new(:==,[self,o])
end
end
```

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

These class definitions define symbolic values, and can be used to perform simple symbolic execution. For example, the following three statements evaluate to (written ⇒) the result of symbolically evaluating the given method call:

```
x = SymbolicObject.new
y = SymbolicObject.new      ⇒ Exp(foo, [x, y])
x.foo(y)
```

Similarly, since Ruby's infix operators are interpreted as method calls, the symbolic evaluator we have defined works for these operators as well:

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

Conditionals. Symbolic execution runs both branches of each conditional. Ruby does not allow redefining conditionals, so we rewrite the source code of the target application to call our own definition of **if**.

```
x = SymbolicObject.new
if x.even? then
  (x+1).odd?                ⇒ Exp(if,[Exp(even?,[x]),
end                             Exp(odd?,[Exp(+,[x,1])])])}
```

Side effects. Programs may perform side effects inside of conditionals; the results should be visible only if the condition holds. We split the current state into two versions, recording the condition and values using a Choice object.

```
x = SymbolicObject.new
if x.even? then x = 1       ⇒ Choice(Exp(even?,[x]) ,1,2)
else x = 2 end
x
```

4.2 Analyzing Rails Applications

SPACE symbolically evaluates each action in a Rails application and collects the set of symbolic objects appearing on a page or written to the database. These symbolic objects represent the set of possible exposures of the application.

Wrapping ActiveRecord. Rails applications interact with the database using ActiveRecord, an object-relational mapper (ORM) that provides methods like "find" and "all." These methods are wrappers around database queries: they allow the specification of queries (including features of SQL like `where` and `group by`) directly in Ruby, and are translated by the ORM into SQL queries. SPACE provides a new implementation of the ActiveRecord API that ignores the concrete database and instead returns symbolic values representing the set of possible results for *all* databases. Given a "User" class extending ActiveRecord, for example, the following code would normally find users with the name "Bob," but it returns a symbolic expression representing the query under our redefinition of ActiveRecord.

```
User.find :name => "Bob"
      ⇒ Exp[User, [Exp(query, [name => "Bob"])]]
```

Rendering. The Rails renderer evaluates a *template*—which contains both HTML and Ruby expressions—to produce an HTML string. SPACE wraps the Rails rendering API, recording the symbolic objects referenced during evaluation of the Ruby expressions contained in the template. This strategy allows the existing renderer to run unchanged, which is valuable because Rails developers often replace the standard Rails renderer with other versions.

4.3 Constraint Specializer

Using the user-provided mapping from application resources to role-based access control objects, the constraint specializer substitutes one for the other in each exposure's constraints and translates the result into the Alloy language.

User-provided Mapping. As in the example in Section 2, the user provides the mapping between application resources and role-based access control objects using a SPACE-provided embedded domain-specific language. The mapping specification is written according to this grammar:

```
<map_spec>   ::=  <Clause>*
   <Clause>  ::=  <Resource>: <Object>
             |    <Resource>: <Object>(<FMap>*)
    <FMap>   ::=  <FieldName>: <MapRelation>
```

where *Resource* is the set of resource names in the application under analysis, *Object* is the set of object names in the role-based access control model, *FieldName* is the set of resource field names defined by the application code, and *MapRelation* is the set of relation names defined by our security patterns (in section 3.4).

Resource/Object Substitution. As in Section 3.5, the specializer replaces all references to names in the *Resource* set with references to their corresponding RBAC objects in *Object*. A field reference *r.f* is replaced by a reference to one of the relations defined by our patterns using the mapping from *FieldName* to *MapRelation* (if one is present). Some field references are not security-related; since no mapping is provided, the specializer leaves these unchanged.

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

4.4 Bounded Verification

Alloy [17] is a specification language based on first-order relational logic with transitive closure. SPACE uses Alloy to represent both the complete pattern catalog and the specialized constraints derived from an application's exposures. Due to the kinds of quantifier alternation present in our pattern catalog, analysis of first-order formulas is required, and so SPACE's verification task is not decidable. Unbounded tools such as SMT solvers are therefore less suited than a bounded verifier like Alloy. As detailed in Section 5.4, our experience suggests that Alloy's analysis is a good fit for the kinds of bugs SPACE is designed to find: it scales well to the bounds required to find the bugs we discovered in our experiments.

We model web applications, role-based access control, and our security pattern catalog in Alloy after the formal description given in Section 3. Each pattern consists of a set of Alloy constraints on RBAC objects. The patterns that define new subsets of the set of RBAC objects are modeled Alloy's *extends* keyword, and the relations defined by the pattern catalog are represented using Alloy's global relations.

SPACE uses the Alloy Analyzer to compare the specialized constraints to the security pattern models and find bugs. The Alloy Analyzer is a tool for automatic bounded analysis of Alloy specifications; it places finite bounds on the number of each type of atom present in the universe, then reduces the specification to propositional logic.

SPACE builds a single Alloy specification containing the model of role-based access control, the definitions of security patterns, and for each exposure, an *Operation* definition and a predicate imposing its specialized constraints. Finally, SPACE invokes the Analyzer to check that each operation predicate is implied by the pattern catalog.

4.5 Limitations

SPACE's analysis sacrifices soundness for compatibility and scalability, and is therefore capable of analyzing real-world applications. This compromise leads to several limitations:

Dynamic environment changes. No static analysis can provide guarantees about a Ruby program without making assumptions about the environment, since Ruby allows replacing any method's implementation at any time. SPACE assumes that the environment at analysis time is the same as in production.

Unexpected API calls. SPACE detects only information flowing from ActiveRecord calls to rendered pages. It therefore misses leaks through native code, direct network connections, or non-ActiveRecord data sources. SPACE relies on the application developer to ensure that these mechanisms do not introduce security bugs.

Bugs outside the finite bound. SPACE uses a bounded model finder to find violations of its pattern library. If the counterexample exposing a bug contains a large number of objects, the model finder may not be able to represent it within its finite bound. In this situation, the model finder may report no bugs even when one exists. In practice, all of the bugs we encountered require only a handful of objects to represent a counterexample.

5 Evaluation

In evaluating SPACE, we examined three questions:

Is SPACE effective at finding bugs? We applied SPACE to the 50 most watched open-source Ruby on Rails applications on Github. 30 of those implement access control; SPACE found bugs in 8 (a total of 23 bugs). SPACE reported 10 false positives in addition to these actual bugs. These results suggest that SPACE is effective at finding security bugs and does not produce excessive false positives.

Does SPACE's analysis scale to real-world applications? We recorded analysis times for the 30 analyses performed in the experiment above; every analysis finished within 64 seconds, and the average time was 38 seconds. We also tested SPACE's symbolic execution step on the 1000 most watched Rails applications on Github; every execution finished within one minute. These results suggest that SPACE's symbolic evaluator scales to applications of many sizes.

How does the bound selected for verification affect the number of bugs found and the scalability of the analysis? For the bugs found in the experiment described above, we re-ran the bounded verification step at progressively lower finite bounds until the verifier no longer detected the bug. A finite bound of 3 was sufficient to find these bugs; SPACE uses a bound of 8 (thus considering 8 objects of each type) by default—a bound well in excess of that required to find the bugs we discovered. For a bound of 8, the verification step finished in under 10 seconds for all applications. These results suggest that SPACE's default bound of 8 scales well and is large enough to catch most bugs.

5.1 Experimental Setup

We considered the 50 most-watched open-source Ruby on Rails applications hosted by Github, assuming that the number of "watchers" a repository has is proportional to its popularity. These applications fall into two categories: applications intended for installation by end-users and fragments of example code intended to help programmers build new ap-

Author / Application	LOC	Exposures	Exposure Generation Time	Verification Time	Bugs Found	False Positives	Finite Bound Required
RailsApps/rails3-bootstrap-devise-...	1103	16	45.02 s	7.66 s	0	0	-
nov/fb_graph_sample	655	87	41.13 s	8.31 s	**2**	**1**	**2**
tors/jquery-fileupload-rails-paperclip...	727	15	28.71 s	7.11 s	0	0	-
heartsentwined/ember-auth-rails-demo	742	28	28.94 s	6.48 s	0	0	-
kagemusha/ember-rails-devise-demo	619	24	29.84 s	3.25 s	0	0	-
openbookie/sportbook	3568	94	49.67 s	7.16 s	**4**	0	**3**
shageman/rails_container_and_engines	778	39	23.94 s	7.20 s	0	**3**	-
OpenFibers/TOT2	1470	54	34.07 s	5.21 s	**2**	0	**2**
Eric-Guo/bootstrap-rails-startup-site	389	17	41.00 s	4.72 s	0	0	-
IcaliaLabs/furatto-rails-start-kit	293	16	44.69 s	4.95 s	0	**1**	-
ngauthier/postgis-on-rails-example	206	22	21.56 s	4.14 s	0	0	-
johnbender/jqm-rails	340	43	24.66 s	3.84 s	**3**	0	**2**
GAFuller/rails-4-landing	800	21	41.67 s	3.09 s	0	0	-
netxph/redport	142	9	22.68 s	3.73 s	0	**2**	-
PaulL1/league-tutorial	833	31	22.50 s	3.14 s	0	0	-
jwhitley/requirejs-rails-jasmine-...	197	11	23.36 s	8.17 s	0	0	-
danneu/grinch	1059	24	32.27 s	5.44 s	0	0	-
phaedryx/rails-angularjs-example	423	17	23.31 s	8.18 s	0	0	-
brobertsaz/railscrm-advanced	2373	132	59.75 s	4.63 s	**5**	0	**3**
Ask11/dreamy	243	13	22.14 s	8.84 s	0	**1**	-
m1k3/tada-ember	200	8	26.60 s	6.67 s	0	0	-
tomgrim/ribbit	473	41	24.53 s	3.35 s	**2**	0	**2**
myitcv/test-signet-rails	282	20	27.05 s	4.51 s	0	0	-
hjhart/yelp	1077	48	21.78 s	6.55 s	0	**1**	-
geraldb/world.db.admin	866	63	27.95 s	7.22 s	0	0	-
seankwon/MediumClone	797	46	38.43 s	3.43 s	**4**	0	**2**
yakko/permissions	523	58	22.82 s	7.82 s	0	0	-
theef/rails-devise-backbone-auth	296	14	26.19 s	7.69 s	0	**1**	-
asterite/geochat-rails	9596	89	49.32 s	3.99 s	0	0	-
OAGr/rails-api-authentication	1003	51	32.22 s	4.56 s	**1**	0	**2**

Figure 4: Results of Running Space on Access Control-Enabled Open-Source Rails Applications on Github

plications. Security bugs are a serious issue in both types of application: in end-user applications, they make user data vulnerable, while in example code, they can propagate to applications that have copied the code.

Selecting Applications. We used the Github Archive database[4] to make a list of the most-watched Rails-associated repositories. We developed a Ruby script to automatically clone each repository in the resulting list, check that it contains a Rails application, install its dependencies, and run the application. We used this script to filter the original list of 4000 repositories down to the 1000 most-watched repositories containing actual Rails applications. We used the first 50 of these to perform our bug-finding experiments, and the full list of 1000 applications for our scalability tests.

Building Mappings. We first eliminated those applications with no access control through manual inspection, leaving 30 applications with some access control checks. For each of these, we constructed a mapping from application resources to RBAC types as detailed in section 3. We used application documentation and the code itself as a guide; in most cases, it was a straightforward process, involving mapping the "User" type to RBAC users, a "Permission" type to permission objects, a subset of resources to owned objects, and perhaps a role type to role objects. This process took us approximately 15 to 25 minutes per application.

[4]https://www.githubarchive.org/

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

Limitations. The 50 applications we considered contained a number of abandoned and "toy" projects that likely have few real-world users. Our experimental design—sorting applications by number of watchers—was an attempt to mitigate this factor in an objective way; we speculate that a large number of real-world Rails codebases are not open source, causing a bias in open-source applications towards example and toy projects.

5.2 Bug Finding

The results of the experiment are summarized in Figure 4. In total, Space reported 23 bugs and 10 false positives. The longest analysis took 64 seconds, while the average time was 38 seconds. In most cases, the symbolic execution step (generating exposures) took most of the time.

Bugs Found. Figure 5 classifies the 23 bugs we found according to the pattern we believe the programmer *intended* to implement. The largest category is Authentication, indi-

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

Figure 5: Classification of Bugs Found by Pattern Violated

cating that the check programmers forget most often is the one ensuring that the user is logged in. The next largest, Ownership and Explicit Permission, indicate situations in which a user is allowed to view or modify a resource owned by someone else without permission. The User Profiles category includes situations where a user can modify another user's profile, and the Public Objects category includes bugs where a resource was marked public when it should not have been. We did not find any bugs violating the Administrator or Explicit Roles patterns; developers, it seems, tend to use these patterns correctly.

The vast majority of vulnerabilities we discovered fell outside the standard use cases of the applications. To exploit these, an attacker must craft a URL or form submission manually; exposing the bug using the application's exposed user interface is impossible. This is the kind of security bug that is difficult to detect using testing, because the programmer writing the test cases is biased towards the standard use cases of the application, and often ignores access paths outside of those use cases.

False Positives. In our experiment, SPACE produced a total of 10 false positives. All of these were situations in which the application exposed a particular field of a resource in a different way from the other fields of that resource. For example, the redport photo manager allows users to provide "feedback" on a photo that another user owns, but does not allow changing any of that photo's other fields. Because SPACE maps whole resource types to RBAC objects, it does not allow per-field permissions to be specified.

However, this limitation is not inherent in our approach—it is a conscious decision to make the resource mapping easier to construct. A finer-grained mapping would eliminate these false positives, but would require more effort to construct.

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

For example, an e-commerce web site may implement administrative access, but may disallow even administrators from viewing customers' saved credit card numbers. Our catalog, on the other hand, includes a pattern allowing administrators to view everything. SPACE would therefore miss a bug allowing administrators to view card numbers, since the intended policy is actually more restrictive than the pattern in our catalog.

We examined the code of the applications in our experiment for precisely this situation—security policies intended (based on evidence in the code itself) to be more restrictive

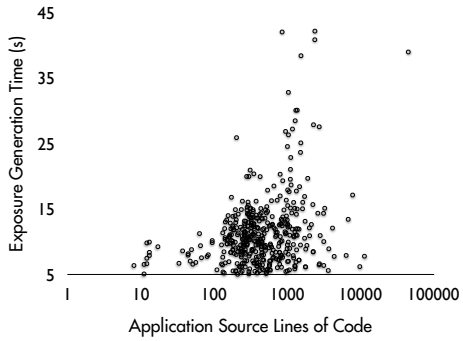


Figure 6: Scalability of Exposure Generation: Generation time vs application lines of code for 1000 applications

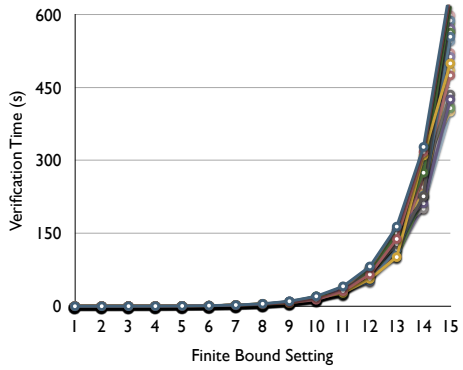


Figure 7: Effect of Finite Bound on Verification Time

than the corresponding patterns in our catalog—and found none. Given the correct user-provided mapping, the patterns applied by SPACE were always at least as restrictive as those enforced by the target applications. However, the number of applications we considered was limited: we do not consider this enough evidence to conclude that security policies are generally less restrictive than our patterns.

5.3 Scalability

Since exposure generation time dominates verification time, we performed an additional experiment to test the scalability of the exposure generator more thoroughly. We ran the exposure generation step *only* on the 1000 most-watched Rails applications on Github. Figure 6 plots exposure generation time against lines of source code for each of these applications. Exposure generation finished within one minute for every application, with most taking much less time.

5.4 Choice of Finite Bounds

SPACE uses the Alloy Analyzer, which requires the user to place finite bounds on the number of objects it will consider. Since these bounds both scalability and bugs found (because bugs requiring a counterexample larger than the finite bound will be missed), choosing the default bound carefully is vital. We ran the verification step against the applications in Figure 4 using finite bound settings from 3 to 15. The results, in Figure 7, indicate that the verification step runs in

an acceptable time at finite bound settings below 10. Above that bound, the verification time grows quickly.

We chose the default bound of 8 based on these performance results and the counterexample sizes required for the bugs we discovered. Bugs may exist beyond this bound, but the distribution of bounds required to find the bugs we discovered (100% of bugs with bound 3, and 0% of bugs from bounds 3-15) suggests that more bugs have small counterexamples than have large ones. Users of SPACE can easily select a different bound in order to tune the verifier.

6 Related Work

Modeling Security Patterns. Conceptually, our catalog of security patterns is similar to the built-in specifications of common bugs that static analysis tools have used for decades: memory errors, race conditions, non-termination, null pointer exceptions, and so on. Like SPACE, tools that attempt to find these classes of bugs can run without user input. However, these specifications represent a blacklist—a model of situations that should *not* occur—in contrast to our catalog of patterns, which whitelist only those situations that *should* occur. Our pattern catalog is similar in approach to the formal model of web security by Akhawe et al. [1], which defines a general model of the actors (browsers, servers, attackers etc.), and threats (XSS, CSRF, etc.) involved the larger picture of web security problems, then uses a constraint solver to discover attacks that potentially involve several different components; the model is not directly applicable to code. Our technique, by contrast, focuses on mistakes in access control policies, and compares the model against the code automatically.

Symbolic Execution. Symbolic execution was first proposed by King [20] and Clarke [11] in 1976, and recent symbolic evaluators [7, 18, 28, 33, 32, 16] have made improvements in scalability and applicability of the idea. Symbolic execution has also seen application to dynamic languages in recent years, with systems for evaluating Javascript [32, 13, 32], PHP [2, 19], Python [31], and Ruby [9]. In contrast to SPACE, all of these use standalone symbolic evaluators.

Other recent efforts implement symbolic execution as a library, like SPACE. NICE-PySE [8] and Commuter [12] both implement library-based symbolic execution in Python, but only for particular domain-specific languages. Yang et al. [36] and Köskal et al. [21] use the same technique in Scala, but to enforce security policies and perform constraint programming, respectively. Rosette [35] uses a library to symbolically execute Racket for verification and program synthesis. None of these systems allow symbolic execution of arbitrary programs, however.

Chef [5] produces symbolic evaluators for interpreted languages by symbolically executing *the standard interpreter itself* on the target program. Like our approach, Chef results in a system that is 100% compatible with the standard interpreter; however, the indirection of symbolically executing the interpreter incurs significant overhead (at least 5x over NICE-PySE [8], which is implemented like SPACE).

Static Analysis of Web Applications. Existing work on the application of static analysis to web applications focuses on modeling applications, and especially on building navigation models. Bordbar and Anastasakis [4], for example, model a user's interaction with a web application using UML, and perform bounded verification of properties of that interaction by translating the UML model into Alloy using UML2Alloy; other approaches ([22, 34, 29]) perform similar tasks but provide less automation. Nijjar and Bultan [27] translate Rails data models into Alloy to find inconsistencies, but do not examine controller code. Bocić and Bultan [3] and Near and Jackson [24] check Rails code, but require the user to write a specification. SPACE, in contrast, requires only that the programmer provide a simple object mapping.

Techniques that do not require the programmer to build a model of the application tend to focus on the elimination of a certain class of bugs, rather than on full verification. Chlipala's Ur/Web [10] statically verifies user-defined security properties of web applications, and Chaudhuri and Foster [9] verify the absence of some particular security vulnerabilities for Rails applications; unlike SPACE, neither approach can find application-specific bugs. Derailer [25] can find application-specific bugs but lacks SPACE's automation: it requires the user to interact with the tool to discover bugs.

Run-Time Approaches to Web Application Security. Resin [37] is a runtime system that enforces information flow policies attached to data objects; it has been successfully applied to web applications. Jeeves [36], a similar language for enforcing information flow policies, has also been applied to the web. Jif [23], an extension of Java, also supports checking policies at runtime. GuardRails [6] allows the programmer to annotate ActiveRecord classes with access control information, then performs source-to-source translation of the application's implementation, producing a version of the application that enforces, at run time, the access control policies specified by the annotations. Nemesis [14] is a similar effort for PHP applications: it is a tag-based system that enforces authentication and access-control at runtime. All of these approaches, in contrast to SPACE, require the programmer to write some form of specification to encode the application's security policy.

7 Conclusion

In this project, we explored the idea that a small set of formal models—our catalog of access control patterns—is enough to capture a large portion of the access control policies that web applications use in practice. We have presented a technique leveraging this idea to find security bugs in web applications, based on checking that if the application allows some data to be exposed, then some pattern in the catalog must also allow it. To be effective, this approach relies on the premise that real-world web applications share common patterns of access control.

Our implementation, SPACE, demonstrates that our catalog of patterns, plus a mapping from application types to the types of the catalog, is sufficient to find bugs in real-world applications automatically. The results of our evaluation indicate that this approach is effective: SPACE scales to large applications, produces few false positives, and found a significant number of previously unknown bugs, suggesting that our pattern-based approach is a promising strategy for eliminating application-specific security bugs.

Acknowledgements

We are grateful to the anonymous reviewers for their insightful comments. We also thank Eunsuk Kang, Aleksandar Milicevic, and Jean Yang for their helpful discussions. This research was funded in part by the National Science Foundation under grant 0707612 (CRI: CRD - Development of Alloy Tools, Technology and Materials)

8 References

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

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

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

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

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

[6] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. Guardrails: a data-centric web application security framework. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 1–1. USENIX Association, 2011.

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

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

[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*, pages 585–594. ACM, 2010.

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

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

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

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

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

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

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

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

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

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

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

[21] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In John Field and Michael Hicks, editors, *POPL*, pages 151–164. ACM, 2012.

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

[23] A.C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.

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

[25] Joseph P Near and Daniel Jackson. Derailer: interactive security analysis for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 587–598. ACM, 2014.

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

[27] Jaideep Nijjar and Tevfik Bultan. Bounded verification of ruby on rails data models. In Matthew B. Dwyer and Frank Tip, editors, *ISSTA*, pages 67–77. ACM, 2011.

[28] C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant

generation. *Model Checking Software*, pages 164–181, 2004.

[29] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer, editors, *ICSE*, pages 25–34. IEEE Computer Society, 2001.

[30] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[31] Samir Sapra, Marius Minea, Sagar Chaki, Arie Gurfinkel, and Edmund M Clarke. Finding errors in python programs using dynamic symbolic execution. In *Testing Software and Systems*, pages 283–289. Springer, 2013.

[32] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[33] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

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

[35] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 54. ACM, 2014.

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

[37] A. Yip, X. Wang, N. Zeldovich, and M.F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304. ACM, 2009.