Symbolic Model Extraction for Web Application Verification

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

***Abstract*—Modern web applications use complex data models and access control rules which lead to data integrity and access control errors. One approach to find such errors is to use formal verification techniques. However, as a first step, most formal verification techniques require extraction of a formal model which is a difficult problem in itself due to dynamic features of modern languages, and it is typically done either manually, or using ad hoc techniques. In this paper, we present a technique called *symbolic model extraction* for extracting formal data models from web applications. The key ideas of symbolic model extraction are 1) to use the source language interpreter for model extraction, which enables us to handle dynamic features of the language, 2) to use code instrumentation so that execution of each instrumented piece of code returns the formal model that corresponds to that piece of code, 3) to instrument the code dynamically so that the models of methods that are created at runtime can also be extracted, and 4) to execute both sides of branches during instrumented execution so that all program behaviors can be covered in a single instrumented execution. We implemented the symbolic model extraction technique for the Rails framework and used it to extract data and access control models from web applications. Our experiments demonstrate that symbolic model extraction is scalable and extracts formal models that are precise enough to find bugs in real-world applications without reporting too many false positives.**

***Keywords*-Formal Verification; Model Extraction; Web Applications**

I. Introduction

Web applications are used in all aspects of life. Due to the convenience of cloud-based data stores, many web applications store private and sensitive user data, loss or leakage of which could be disastrous for individuals and organizations. The complexity of data models and access control rules used by modern web applications leads to programming errors that can compromise both integrity and privacy of data. Hence, eliminating data integrity and access control errors from web applications is a critical problem.

In recent years, dynamically typed, interpreted languages such as JavaScript, Python and Ruby have become commonly used for web application programming. These languages, compared to statically typed languages, offer more flexibility and typically require less source code to implement the same program. Moreover, web application frameworks (such as Rails and Django) utilize dynamic features of these languages to provide a rich set of tools to developers. The increased expressiveness, however, comes at a cost. Certain compile-time guarantees of correctness become impossible to achieve in general. These include guarantees that are taken for granted in statically typed languages, such as the guarantee that every invoked function or method exists.

There exists a significant body of work on model based verification of web applications [29], [11], [27], [3], [6], [8], [17], [19], [23], [24], [22]. These approaches rely on extraction of formal models, where the extracted model is an abstraction of the program, focusing on a particular feature or behavior of the program. These models are then verified using model checkers or theorem provers [12], [9], [10], [31], [30], [14].

Some of this research specifically targets applications written in dynamically typed languages [17], [18], [3], [4]. Extracting a formal model from programs written in dynamically typed languages is a challenging problem in itself. Because of the difficulties in model extraction, web applications verified in most research papers tend to be simple. Third party libraries that augment the default behavior of the development frameworks are generally avoided. To statically verify such applications, verification tools require either that the applications do not use these features [21], [22], or they pre-process applications to make declarative constructs explicit [13]. Even with these limitations that are due to challenges in model extraction, model based verification techniques were able to find various previously unknown security and data integrity bugs in open source web applications[17], [20], [21].

In this paper, we present *symbolic model extraction*: a technique for automatic extraction of formal models from applications written using dynamically typed, interpreted languages. Symbolic model extraction takes a program in a source language as input and generates a model in a target modeling language as output, such that the generated model abstracts the behavior of the input program. Symbolic model extraction is based on the following ideas: 1) We instrument code such that executing the instrumented code returns the model that corresponds to the original code; 2) We implement the instrumentation function in the source language itself such that we can instrument newly dynamically generated code as it is encountered; 3) The instrumentation function replaces branches with code that handles *both* paths of the branch in a single execution in order to achieve full path coverage.

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

```
1  class Article < ActiveRecord::Base
2    acts_as_paranoid
3  end
4  class ArticlesController < ApplicationController
5    load_resource
6    before_action :destroy do
7      redirect_to :back unless current_user.verified?
8    end
9    def destroy
10     @article.destroy!
11   end
12 end
```

Fig. 1: Ruby on Rails Example.

Symbolic model extraction has its own limitations. Our approach may not correctly handle code dynamically generated from user input. This is not a significant problem in our experience, as directly evaluating user supplied code is unsafe and slow and, hence, is not common practice. Moreover, under certain conditions, the presence of mutually conflicting code generation under different program paths may result in extraction of inaccurate models. We did not experience this in practice, although it is a theoretical possibility.

To validate our approach, we implemented symbolic model extraction for data model verification of Rails applications. Our tool is open source and available online[1].

Our experiments show that symbolic model extraction can be used to extract models from real-world web applications very efficiently. We extracted models from 19 open source Rails applications, spanning up to 84 KLoC (Ruby source only). Extracting the entire model of most applications took under 10 seconds, and 62 seconds for the largest application. Automated verification of the extracted models took 0.069 seconds per property on average. On these 19 applications, we verified 18,490 properties. 404 properties failed and we manually confirmed that these are due to real bugs in the application code. We also observed 56 false positives. These results demonstrate that symbolic model extraction is fast and precise enough to be useful for finding bugs in practice.

The rest of the paper is organized as follows. Section II presents a portion of a Rails application, demonstrating why model extraction from applications written in dynamically typed web applications can be difficult. In Section III we discuss the foundations of symbolic model extraction. Section IV explains how we implemented symbolic model extraction to extract models of Rails applications that focus on data integrity or access control. Section V presents the experimental evaluation of our extraction. Section VI discusses related work, and in Section VII we conclude the paper.

II. A Model Extraction Example

Consider the excerpt of a Rails application in Figure 1. Lines 1-3 declare a *model class*, which is a class whose objects can be stored in the database. This particular class, `Article`, defines articles that are managed by this web application. This class does not contain any fields or additional methods for the sake of clarity. Lines 4-12 define the `ArticlesController`.

[1]http://bocete.github.io/adsl/

```
1  class ArticlesController < ApplicationController
2    def destroy
3      @article = Article.find(params[:id])
4      redirect_to :back unless current_user.verified?
5      @article.deleted_at = Time.now
6      @article.save!
7    end
8  end
```

Fig. 2: Static Equivalent to Action in Figure 1.

Controllers define *actions* which are executed in response to user requests. The one action in this example is called `destroy` (lines 9-11). This action seemingly deletes the object stored in the `@article` variable using the `destroy!` method (line 10).

Due to dynamic features that are often used in Rails, the source code of the action is deceptive. The action does much more than deleting an object.

First, it is not clear which article object is being deleted. In line 5 of Figure 1 we see the `load_resource` declaration, defined by the CanCan gem [7]. This declaration will ensure that, before an action executes, the framework will preload an object and store it in a variable, to be accessed from inside the action. The specifics of this preload operation are subject to a number of conventions such as the name of the controller, the name of the action, and configurations.

Second, the `before_action` declaration in lines 6-8 prepends a *filter* to the action. Filters execute before or after an action and are usually used to prepare data for an action, or to conditionally prevent an action from executing any further. In this case, if the current user is not verified (line 7), the filter will redirect to a different page. This redirection will prevent the action from executing.

Finally, in line 10, the action invokes the `destroy` method on the object in order to delete it. However, in line 2, the `acts_as_paranoid` declaration (provided by the ActsAsParanoid gem [26]), overrides the `destroy!` method for the Article class. Instead of deleting an object, the object is simply marked as deleted but not removed from the database. This allows for Article objects to be restored later if need be.

Figure 2 contains a destroy action that is semantically equivalent to the action in Figure 1, but with its semantics directly understandable from the source code.

This is a simple example of how actions can be enhanced using dynamic features of the Ruby language. There exists a rich set of libraries that Rails developers can use to similarly enhance the framework. Some of these libraries, such as ActiveAdmin [2], can even generate entire actions that are not present statically. This way of developing applications makes static analysis and model extraction difficult, as the semantics of an application are fully defined only at runtime, after libraries have had the opportunity to augment them.

We explain symbolic model extraction and its application to data model verification in the following sections, but here we wish to demonstrate the result of symbolic model extraction when applied to the example in Figure 1. As the target modeling language, we focus on Abstract Data Store Language (ADSL) which has been defined in our earlier work [3].

```
1  action destroy
2    article = oneof(Article)
3    if *
4      return
5    end
6  end
```

Fig. 3: Model Extracted from Figure 1.

We discuss features of ADSL later in Section IV. When applied to the example in Figure 1, symbolic model extraction extracts the ADSL model in Figure 3. This model is an abstraction of the original method. In line 2, an object is read from the database but there are no specifics on which object is loaded. Lines 3-5 will abort the action, depending on a nondeterministic condition. Finally, the model will correctly omit the delete operation that was seemingly present in the original source code. This model is a sound abstraction of the full semantics of the action, including its dynamic features.

III. Symbolic Model Extraction

We explain symbolic model extraction on an abstract programming language L. Let us assume that L is an interpreted, dynamically typed, imperative programming language with functions as first-class citizens (e.g. functions can be assigned to variables, passed as arguments to function calls etc.).

For simplicity, we will represent a program written in L as a statement s. Since a sequence of statements is itself a statement, this perspective is accurate. At runtime, programs written in L are executed using the interpreter I where I evaluates statements to migrate the program from one state to another state.

Because this is a dynamically typed language, the types of objects assigned to variables may change over time. In addition, the type system in the program can change in any number of ways: Classes can be defined at runtime, methods can be added or even replaced at runtime.

Let $\overline{L}$ be the set of program states in L. These states include the program counter, the stack and heap memory states. This lets us define a statement s as a set of state transitions:

$$s \subseteq \overline{L} \times \overline{L}$$

In words, given an initial state $l \in \overline{L}$, executing a statement s will migrate the program state to some state l' such that $\langle l, l' \rangle \in s$. Furthermore, we constrain the definition of statements to have at least one state transition from any program state. Let S be the set of all statements in language L.

For model-based verification, in order to verify a program (statement) $s \in S$, we need to extract $s^\sharp$: the model of a statement s in some modeling language $L^\sharp$. This model is an abstraction of the original statement, meaning that if there are behaviors in s, they can be detected in $s^\sharp$.

Let $\overline{L^\sharp}$ be the set of abstract program states. Each abstract program state $l^\sharp \in \overline{L^\sharp}$ is a set of concrete program states:

$$l^\sharp \subseteq \overline{L}$$

Similarly, an abstract statement $s^\sharp$ is a set of transitions between abstract states that abstracts a concrete statement s.

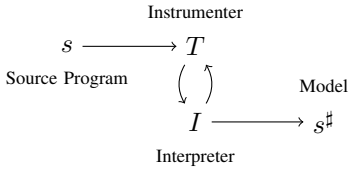


Fig. 4: Overview of symbolic model extraction

More precisely, for every state transition in s, $s^\sharp$ contains the transition of corresponding abstract states:

$$\forall \langle l, l' \rangle \in s \colon \exists \langle l^\sharp, l^{\sharp\prime} \rangle \in s^\sharp \colon l \in l^\sharp \wedge l' \in l^{\sharp\prime}$$

We can see that $s^\sharp$ simulates the behavior of s, i.e., for each behavior in s there exists a corresponding behavior in $s^\sharp$. Hence, $s^\sharp$ is an abstraction of s.

Note that this definition of statements does not account for expression return values. We are using this set of definitions to simplify the presentation.

A. Symbolic Model Extraction Rules

Symbolic model extraction uses the interpreter for the source language, and an instrumentation function that is accessible during runtime, to execute the input program in an instrumented, path-insensitive environment to explore static as well as dynamically generated code and extract the model for the given program in the target modeling language.

We illustrate the high level information flow in the symbolic model extraction in Figure 4 where the input program (s) in the source language is passed to the instrumenter function (T). The instrumenter will instrument the given program and pass it to the interpreter for execution. When new code is encountered or dynamically generated, the interpreter will pass this new code to the instrumenter for immediate instrumentation. The execution of the instrumented program returns the extracted model ($s^\sharp$) in the target modeling language.

In order for this approach to work without developing a custom interpreter, the instrumenter has to be implemented in the source programming language itself. Newly generated code can then be investigated and instrumented using metaprogramming.

Key to our approach is the *symbolic model extraction instrumentation function* T, or the *instrumenter* in short. T is a function $T \colon S \to S$ that, given a statement $s \in S$, returns the instrumented statement $T(s)$. When executed by the interpreter I, $T(s)$ evaluates to the model of s:

$$I(T(s)) = s^\sharp$$

In words, the instrumenter transforms a statement such that executing the transformed statement using the source language interpreter returns the model of the original statement.

After implementing T in the source language, we can use it to instrument and extract a model of a dynamic program using the source language interpreter. We instrument and invoke the program's entry point within the same runtime. Our instrumentation ensures that instrumentation will propagate as new code is discovered or even generated.

Additionally, we override all methods that are relevant to our abstraction to make them execute symbolically. For

Rule #	s	$T(s)$	$I(T(s))$
1	$\alpha_1;\alpha_2;\ldots;\alpha_n$	`ins_block(`$T(\alpha_1)$`, `$T(\alpha_2)$`, ..., `$T(\alpha_3)$`)`	$\alpha_1^\sharp;\alpha_2^\sharp;\ldots;\alpha_n^\sharp$
2	`fn(`$\alpha_1 \ldots \alpha_n$`)`	T`(fn)(`$T(\alpha_1)\ldots T(\alpha_n)$`)`	$fn^\sharp(\alpha_1^\sharp \ldots \alpha_n^\sharp)$
3	`if` α `then` β `else` γ	`ins_if(`$T(\alpha)$`, `$T(\beta)$`, `$T(\gamma)$`)`	**`if`**$^\sharp$`(`$\alpha^\sharp$`, `$\beta^\sharp$`, `$\gamma^\sharp$`)`
4	`while` α β	`ins_while(`$T(\alpha)$`, `$T(\beta)$`)`	**`while`**$^\sharp$`(`$\alpha^\sharp$`, `$\beta^\sharp$`)`
5	$\alpha\, op\, \beta$	`ins_op(`op`, `$T(\alpha)$`, `$T(\beta)$`)`	$\alpha^\sharp\ op^\sharp\ \beta^\sharp$, if $op^\sharp$ exists within the abstraction $*$ otherwise

6	`var =` α	`result =` $T(\alpha)$
		`var = SymVar.new(result.sym_type, 'var')`
		`ins_assignment('var', result)`
			`var =`$^\sharp$ $\alpha^\sharp$

7	`var`	`var`	`var`$^\sharp$

TABLE I: Statement Instrumentation

example, if we aim to extract a model that focuses on database operations, methods that communicate with the database will be overridden such that, instead of communicating with the database, they return a model that describes the nature of communication that was attempted.

The instrumenter's behavior is not obvious when it comes to dynamic language features, control flow, and data flow features such as scoping and assignments that appear in the source program. Table I demonstrates how the instrumenter could be implemented with regards to basic language constructs.

1) Sequences of statements: Rule 1 in Table I demonstrates how a sequence of statements is instrumented in order to extract the model of the sequence.

Let us extract the model of a sequence of statements $s = \alpha_1;\alpha_2;\ldots;\alpha_n$ using symbolic model extraction. We instrument the sequence in such a way that executing the instrumented sequence returns the model of the sequence. To achieve this, we replace each statement α_k with its instrumented version $T(\alpha_k)$ and pass these instrumented statements as arguments to an `ins_block` function. As such,

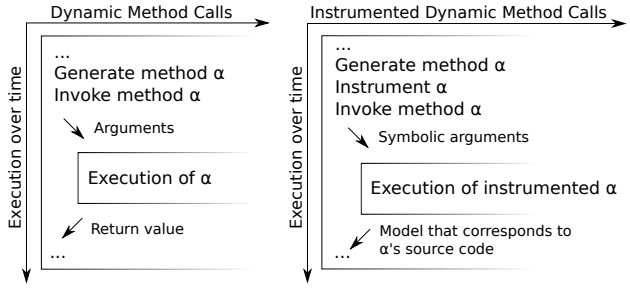
$$T(s) = \texttt{ins\_block}(T(\alpha_1), T(\alpha_2), \ldots, T(\alpha_n))$$

Note that this instrumented statement is still a valid statement in the source language. `ins_block` is a method provided by our instrumentation library that merges a sequence of models of statements into a block model.

When executing $T(s)$, the interpreter will first evaluate each argument for `ins_block` in order. The result of each argument $T(\alpha_k)$ will be $\alpha_k^\sharp$, the model of the statement α_k. Finally, these models will be merged by `ins_block` into the sequence of statements in the modeling language.

2) Method calls: Rule 2 in Table I refers to how function (or method) calls are treated by the instrumenter. Any call to a function $fn(\alpha_1, \ldots, \alpha_n)$ in the source program is replaced with a call to $T(fn)(T(\alpha_1), \ldots, T(\alpha_n))$. In words, the instrumented function gets executed instead of the original function, with arguments having been instrumented as well. The result of this execution, as defined by the instrumenter, will be the model of the function's body.

To illustrate this approach, consider Figure 5(a). It illustrates the execution of a program that dynamically generates a method α and subsequently invokes it. Since code generation is done at runtime, it poses a problem for standard model extraction techniques. Figure 5(b) demonstrates how symbolic



(a) Normal execution (b) Instrumented execution

Fig. 5: Symbolic model extraction from dynamically generated methods.

model extraction extracts the model of a dynamically generated method. After generating the method, the interpreter instruments the generated method at runtime. The instrumentation alters the method such that, after it is invoked with symbolic arguments, the instrumented method returns its own model. This is how instrumentation is propagated through the program: all functions are instrumented just before they are invoked, allowing us to extract statically available source code as well as source code that might not exist statically.

3) Control Flow: Dynamic program analysis is typically subject to the problem of path explosion. Symbolic model extraction bypasses this problem by exploring all paths of the program at the same time. We transform branches into a sequence of three symbolic executions: one for the condition, one for the then block, and one for the else block. We execute all these in an instrumented environment in order to extract models of the condition and both branches. Finally, we combine the resulting three models into a branch statement in our modeling language.

Rule 3 in Table I summarizes our approach for extracting models of branches. Given a branch where α is the condition and β and γ are the then and else block respectively, the instrumenter replaces the branch with a call to `ins_if(`$T(\alpha)$`, `$T(\beta)$`, `$T(\gamma)$`)` which will consecutively instrument and execute the condition and both paths. The results of executions of instrumented elements are the models of each element, which are combined into a model representation of the branch itself. Note that, contrary to intuition, this approach can handle some often encountered situations where different paths have seemingly conflicting side effects, such as assigning different

```
1 do
2   a = Article.new
3   a.destroy!
4 end
```

(a) Rails code

```
1 ins_block(
2   (
3     result = ins_call(Article, :new);
4     a = SymVar.new(result.sym_type, 'a');
5     ins_assignment('a', result)
6   ),
7   ins_call(a, :destroy!)
8 )
```

(b) Instrumented Rails code

```
1 {
2   a = create(Article)
3   delete a
4 }
```

(c) Extracted Model

Fig. 6: Symbolic model extraction example.

values to the same variable. This will be made clear in Section III-A5.

Loops are handled analogously (Rule 4 in Table I). Instead of executing the loop body a number of times, the loop can be instrumented and executed only once to extract the model of the loop body.

4) Expressions: Rule 5 in Table I explains how our approach handles expressions. Even though this discussion assumes that the expression is a binary operator, the principle generalizes to any number of arguments.

Given an expression α *op* β, we first extract the model of α and β by instrumenting and evaluating them. Then, depending on the operation *op* itself and $\alpha^\sharp$ and $\beta^\sharp$, using the `ins_op` function, we return either the model that correctly abstracts the expression ($\alpha^\sharp op^\sharp \beta^\sharp$), or $*$, representing any possible value.

The specifics of the treatment of expressions is highly dependent on the specifics of the modeling language. Any expression that can be represented in our modeling language should be instrumented so that executing the instrumented code evaluates to the corresponding model. Otherwise, the instrumented expression should result in a symbol representing any possible value. For example, if the abstraction handles integer addition but not string concatenation, `ins_op(:+,` $\alpha^\sharp$, $\beta^\sharp$`)` would check whether $\alpha^\sharp$ and $\beta^\sharp$ are both integers. If they are, `ins_op` should return the model of an addition operation with appropriate models for arguments. Otherwise, the result of `ins_op` would be a symbol representing any possible value.

5) Variables and scoping: Rule 6 in Table I refers to how variable assignments are treated by the instrumenter and Rule 7 explains how variable reads are treated by the instrumenter. These two operations are closely tied to each other, as we actually instrument variable reads as a side effect of instrumenting assignments.

Given an assignment $var = \alpha$, the instrumenter should generate code that, when executed, returns the model representation of an assignment operation. Similarly, when a variable var is read in the original program, the corresponding model should represent the variable reading operation.

As shown in Rule 6 in Table I, the instrumenter replaces the assignment with a sequence of three statements. The first statement instruments and extracts the model of the assigned expression, storing it in a temporary, local variable `result`.

The second statement creates a model of the variable read operation and stores it in the assigned variable, along with the type of the assigned expression and the name of the variable. That way, whenever this variable is subsequently read by the interpreter, the interpreter will identify the correct variable using the scoping rules of the language and return the proper variable read model. This approach not only reduces the amount of work needed to implement symbolic model extraction as we need not worry about variable scoping rules in the source language, but this is especially useful in Ruby, where it is not trivial to distinguish variable reads from method calls made without parenthesis and arguments.

Finally, the third statement constructs and returns the model of the assignment operation itself.

Since variables do not contain a value that is tied to the expression that was assigned to the variable, symbolic model extraction does not have a problem with the source program assigning different values to the same variable in different program paths. During symbolic model extraction from such a program, although a different models will be extracted from each assignment, all assignments will assign the very same value to the variable in question.

6) Dynamic Code Generation: Symbolic model extraction is built on the assumption that the dynamic features of the source language are used input-independently within the abstraction of choice. In other words, all dynamically generated code has an identical abstraction in every execution of the program. This assumption holds in Rails very often as code generation is almost exclusively used to make the developers work easier, generating code that's identical every time the web application starts.

During instrumented extraction, code generation is executed concretely, with all symbolic values concretized into arbitrary values within their abstraction.

B. Symbolic Extraction Example

We will proceed to demonstrate how symbolic model extraction can be used for model extraction on an example. This example is designed to demonstrate key features of the technique, and how the approach deals with difficulties more straightforward techniques could not handle easily.

Assume that we wish to extract a model from the Ruby block in Figure 6(a). This block creates a new Article object (`Article.new`) and assigns it to a variable called `a` in line 2. In line 3 the `destroy!` method is invoked on the previously created object, deleting the object from the database. These statements are wrapped in a block (lines 1-4).

The target modeling language is ADSL, which we will discuss in more detail in Section IV. The model we will eventually extract should be understandable regardless, and is presented in Figure 6(c)).

1) Program Transformation: The instrumenter will automatically transform the block in Figure 6(a) to the block in Figure 6(b). We refer to the code in Figure 6(b) as *instrumented code* to distinguish it from the original code. Instrumented code is generated as a transformation of the abstract syntax tree of the original block, and is executed instead of the original code. This instrumented code follows the instrumentation rules previously discussed in Table I.

The block presented in lines 1-4 of Figure 6(a) corresponds to the `ins_block` statement that spans lines 1-8 of Figure 6(b). This is a direct application of Rule 1 in Table I. The two arguments of `ins_block`, spanning lines 2-6 and 7 of Figure 6(b), directly correspond to the two statements in lines 2 and 3 of Figure 6(a) respectively. When executing the instrumented code, arguments of `ins_block` will be evaluated one at a time, evaluating to their models. The models of these statements will be conjoined into the extracted block by the `ins_block` call.

The assignment in line 2 of Figure 6(a) is transformed into the sequence of statements in lines 3-5 of Figure 6(b), as described by Rule 6 in Table I.

The statement in line 3 of Figure 6(b) instruments and evaluates the `new` method, storing the result model in a temporary variable. We implemented `ins_call` to facilitate instrumentation and invocation of the instrumented method (line 2 of Table I. In this case, the expression is invoking the `new` method on the `Article` class. This is a core Rails method that creates a new model object, which we overrode to return the model of the object creation operation: `create(Article)`.

The statement in line 4 assigns a model of a variable read operation to `a`. This value mimics the type that was assigned to the variable in the original program, but otherwise has no state. Whenever any subsequent statement reads variable `a`, the value it reads will be the model of the operation of reading `a`.

The statement in line 5 creates a model representation of an assignment operation. This model will be returned to the `ins_block` call in lines 1-8 of Figure 6(b) as the first argument.

The method call in line 3 of Figure 6(a) is transformed into the method call in line 7 of Figure 6(b). This is in concordance with Rule 2 presented in Table I. The `ins_call` statement will execute a method in an instrumented environment in two steps:

1) Find the method that will be invoked on the provided object with any provided arguments, and instrument it.
2) Invoke the method with any provided arguments.

By default in Rails, the `destroy!` method deletes an object from the database. During instrumented execution, calling `destroy!` on an Article object returns the model of a `delete` operation on the called object. In this case, since variable 'a' contains the model of the variable read operation, from `destroy!` we extract the model statement present in line 3 of Figure 6(c).

C. Limitations

Symbolic model extraction has limitations. For one, the approach assumes that the application does not utilize input dependent dynamic features. If it does, for example if a user is given the ability to enter code that will be executed by the application, symbolic model extraction will not extract a sound abstraction of the original source code. In practice, generating code from user input is avoided for performance and security reasons, so we did not encounter this issue.

Our treatment of branches is designed under the assumption that different paths in the program will not use mutually conflicting code generation. Consider a branch statement that executes statement A if the condition holds true and statement B if the condition holds false. Let A and B generate a method under the same name with different source codes. Inside A you would see A's method during both concrete and symbolic execution, inside B you would see B's method during both executions. However, after the branch, symbolic model extraction would only consider B's implementation. Although this is a problem in theory, in practice, we did not encounter such programs. We believe that this problem can be avoided by keeping track of every generated method and using aliasing to access different versions of the same method.

IV. Symbolic Extraction for Data Model Verification

We used symbolic model extraction to models from Ruby on Rails applications. We used these models for two purposes: to verify data integrity [3] and access control [6] properties.

In this section we present the *Abstract Data Store Language* (ADSL), the target modeling language we used for verification of data integrity in Rails applications. We also discuss the extension of ADSL called ADSL+ that enhances ADSL with access control information, although for brevity, ADSL+ is discussed in less detail. In addition to presenting the modeling language, we show how we employed symbolic model extraction to extract models in these target modeling languages.

A. Data Model Verification Models

Since our goal is to verify data integrity or access control, the models we extract need to encompass operations the application does on its data. For data integrity, we need to focus on the way data can be modified by the application, as well as automated validators and constraints that prevent data invalidation. For access control, we additionally need to extract the specifics of the permissions system, as well as observe which objects might be read or modified by different users.

At the core of ADSL is a *data store*. A data store DS is a tuple $\langle C, R, A\rangle$ where C is a set of classes, R is a set of associations, and A is a set of actions. Classes and associations define the types of entities stored in the data store, while actions define possible ways to modify or query the data store. *Data store states* define the exact set of data being stored in the data store. We define $\overline{DS}$ to be the set of all possible data store states of DS. Formally, a data store state is a tuple $\langle O, T\rangle \in \overline{DS}$ where O is the set of objects and T is the set of tuples denoting associations among objects in O.

1) Classes and Associations: Given a data store $DS = \langle C, R, A\rangle$, 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 ($\text{superclass}(c) \subset C$) and, transitively, the superclass relation cannot contain cycles.

Given a data store state $\langle O, T\rangle \in \overline{DS}$, 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 denote all objects in O whose class is c or whose superclass is c (directly or transitively). We define $\overline{O}$ to be the set of all sets of objects that appear in $\overline{DS}$.

Associations define how objects of particular classes can be related to one another. An association $r = \langle name,\, c_o,\, c_t\, \rangle \in R$ contains a unique identifier $name$, an origin class $c_o \in C$, and a target class $c_t \in C$. We omit the definition of cardinality constraints for brevity.

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

Notice that this model does not include fields as part of the class definition. ADSL abstracts basic types away for several reasons. First, Rails natively supports validators that effectively ensure that only valid basic type data can be saved in the database. As such, verification is unlikely to yield useful results. Second, verification of basic types is difficult, especially in case of strings, which are still unsupported by most SMT solvers that we use for verification. This would limit our choice of theorem provers. In addition, there is nothing that prevents symbolic model extraction from extracting basic type field operations. If necessary, this would be a simple addition to our implementation.

2) Actions: Given a data store $DS = \langle C, R, A\rangle$, 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* $\langle q, q', \alpha\rangle \in \overline{DS} \times \overline{DS} \times \overline{O}$ where $q = \langle O, T\rangle$ is the pre-state of the execution, $q' = \langle O', T'\rangle$ 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.

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

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

In ADSL, 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.

To illustrate how statements correspond to state transitions, let us define 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 $\langle q, q'\rangle \in \overline{DS} \times \overline{DS}$ (where $q = \langle O, T\rangle$ and $q' = \langle O', T'\rangle$) are the pre- and post-states of the statement, this statement transitions from q to q' if and only if:

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

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.

ADSL includes statements for creating and deleting objects, creating and deleting tuples between objects, variables and assignments, branches and for-each loops. Updating objects and tuples is outside our abstraction as objects do not contain basic type fields, and as such, have no state that can be updated.

ADSL+ expands on the above definition of data models by modeling authentication, different user roles, the access control policy as well as runtime access control checks that are implemented in the source web application. The formal definition of these models is discussed in prior work [6].

B. Implementation of Symbolic Model Extraction for Rails and ActiveRecord

For both ADSL and ADSL+, in order to run instrumented execution, we have to first install and configure the analyzed application on our own computer. Once we can start the web server, we add to it our own symbolic model extraction library that overrides core Rails methods with their symbolic versions. Finally, we start model extraction of each action by generating http requests that will execute them. The set of requests the server responds to are dynamically extracted from the configuration of the instrumented server. This process is fully automated.

ActiveRecord is the Object-Relation Mapping (ORM) library employed by Rails. It provides the methods by which data can be managed or stored in the data store. As such, both our models heavily rely on overriding ActiveRecord methods with their symbolic counterparts. The configuration and usage of ActiveRecord is highly relevant to our models.

Table II shows parts of the target modeling languages that are common to the models we used to verify data integrity or access control. Table II a) contains class (static) methods, and Table II b) contains instance (object) methods. The first column represents various Ruby on Rails methods. The second column explains the semantics of the corresponding method. Finally, the third column defines the expression in the target modeling language (ADSL) that is extracted from said method. This list is not exhaustive because many methods in Ruby on Rails have multiple aliases (different names that achieve the same functionality) for developer convenience.

For example, `Model.create(attrs)` is a constructor. Developers can use this method to create a new object of type *Model*, setting the newly created object's fields corresponding to the *attrs* argument. Similarly, `Model.all` will return a collection of all objects of type *Model* that exist in the database.

Our library will, when the Rails application is booting up, replace core ActiveRecord methods with their instrumented

ActiveRecord method	Semantics	ADSL expression
Class.create(attrs)	Creates an object with provided attributes (basic type values)	create(*Class*)
Class.all	Load all model objects of this type from the database	allof(*Class*)
Class.where(...)	Load all model objects in the database that satisfy some criteria	subset(*Class*)
Class.find(id)	Finds an object using the provided unique identifier	oneof(*Class*)

a) Class Methods

ActiveRecord method	Semantics	ADSL expression
expr.select(...)	Returns all objects in *expr* that meet some criterion	subset(*expr*)
expr.association	Returns object(s) related to *expr* via the association	*expr*.association
expr.association << expr2	Associates object *expr* with *expr2* via association *association*	createTuple(*expr*, association, *expr2*)
expr.association = expr2	Mutates an association	*expr*.association = *expr2*
expr.delete!	Deletes the object	delete(*expr*)
expr.destroy!	Deletes the object, propagating deletion to associated objects	delete(*expr*.assoc); delete(*expr*)
expr.destroy_all!	Deletes all objects in a collection expression	delete(*expr*)
expr.each(block)	Executes *block* once for each element in *expr*	foreach v in *expr*: *block*
expr.nil?	Checks whether *expr* is null or not	isempty(*expr*)
expr.any?	Checks whether *expr* has at least one object	not(isempty(*expr*))

b) Instance Methods

TABLE II: Table of various ActiveRecord methods and corresponding ADSL expressions

versions. Other libraries that build on top of ActiveRecord do not need to be manually specified and overridden, as when they implement their functionality on top of core ActiveRecord, they become implicitly prepared for instrumentation.

After overriding the core ActiveRecord methods, we identify the set of actions that the Rails application contains. We instrument them and execute them one at the time. Each action will return the model of itself, and these models make part of the entire model we extract from the applications.

For data integrity verification, we make an additional step to extract invariants [3]. These invariants are written in Ruby using a library we developed for this purpose. Adapting our extraction method to extract invariants was straightforward. For example, we added quantification to ActiveRecord objects that can be used to quantify over sets.

ADSL+ similarly requires an additional extraction step, and even though it is more involved than invariant extraction, it is essentially identical to extraction of other model features. We extract access control information from CanCan configuration [6]. Similar to how we handled ActiveRecord, we have manually overridden a few key methods for policy declaration and runtime checks of access control.

V. Experiments

We used symbolic model extraction to extract models from Rails applications in order to verify their data integrity [3], [4], [5] and access control [6] properties. The result of our experiments are summarized in Table III.

We analyzed a total of 19 open source Rails applications. We found these applications from various sources. We looked at the 25 most starred open-source Rails applications on Github according to the OpenSourceRails.com website [25], a compilation of open source Rails applications categorized by domain [1], and applications investigated by related work.

Our implementation of symbolic extraction does not support all versions of Rails, as that would require a substantial engineering effort. Our tool supports Rails 3, up to and including Rails 4.2. Furthermore, since we focused on how applications employ ActiveRecord, we did not extract models from applications that bypass ActiveRecord: for example, if they are not backed by a relational database.

Column *LoC (Ruby)* shows the number of Ruby lines of code in these applications. This number does not include JavaScript, html, dynamic html generation through irb files, or configuration files. Columns *Classes*, *Actions* and *Invariants* show the number of model classes, actions, and invariants respectfully. As invariants are not part of the core Rails framework, we wrote them manually for each application after investigating their source code. We did not write any invariants for Bootstrap and Illyan because their models were too simple to warrant any non-trivial invariants.

Column *Access Control* shows whether the application employs access control through CanCan [7]. For applications that do, we verified access control in addition to data integrity.

Column *Extraction Time* shows the total amount of time it took to extract models from these applications. This includes booting up the Rails application, instrumenting it, and executing each action as described in Section IV. We obtained these results on a computer with an Intel Core i5-2400S processor, running 64bit Linux. Memory consumption was typical for starting and running a Rails application. We manually confirmed that the extracted models are correct.

We find the performance and the quality of our model extraction to be more than acceptable for real world use, potentially both as part of verification of applications during the quality assurance process, and for daily developer use.

A. Verification Results

For each model that was extracted, we used data model integrity verification and access control to verify each application. For data integrity verification, our tool generated set of first order logic (FOL) formulas for each action/invariant pair. We refer to these sets of formulas as *verification queries*. A verification query can be used to verify whether an action

Application	LoC (Ruby)	Classes	Actions	Invariants	Access Control	Extraction Time (sec)	Verification Queries	Avg Time per Query (sec)	Verified	Bugs	Timeouts	False Positives
Avare	1137	6	26	3	✓	3.708	111	0.015	75	36	0	0
Bootstrap	785	2	4	/	✓	2.861	4	0.005	3	1	0	0
Communautaire	753	5	28	6	✓	3.236	216	0.004	208	2	6	0
Copycopter	3201	6	11	6		3.534	66	0.014	66	0	0	0
CoRM	7745	39	163	32	✓	33.868	5671	0.090	5537	91	25	18
FatFreeCRM	20178	32	120	8	✓	14.383	1120	0.030	1077	40	3	0
Fulcrum	3066	5	40	6		4.966	246	0.009	239	7	0	0
Illyan	1486	3	24	/	✓	5.099	25	0.008	24	1	0	0
Kandan	1535	5	25	6	✓	5.395	177	0.009	162	15	0	0
Lobsters	5501	17	86	9		7.576	819	0.203	798	13	4	4
Obtvse2	828	2	13	1		6.266	13	0.001	13	0	0	0
Quant	4124	9	38	4	✓	5.688	203	0.006	203	0	0	0
Redmine	84770	74	264	21		62.295	5796	0.054	5771	19	6	0
S2L	1334	9	44	4	✓	3.913	266	0.022	209	53	4	0
Sprintapp	3042	15	120	8	✓	14.899	1105	0.032	1058	47	0	0
Squash	15801	19	46	18		8.251	828	0.031	824	4	0	0
Tracks	17562	11	117	9		16.349	1188	0.127	1171	11	3	3
Trado	10083	33	66	10	✓	12.094	709	0.081	638	64	5	2
WM-app	2425	18	95	4	✓	6.095	414	0.019	414	0	0	0
Totals	185356	310	1330	155		220.476	18977	0.069	18490	404	56	27

TABLE III: Experimental Results

could potentially invalidate an invariant. Given an action a and invariant i, assuming a pre-state in which all invariants hold, is it necessarily true that i will hold in the post-state? If not, verification fails and reports an error.

If the application employs access control, we generate verification queries that verify access control enforcement in the application. This verification technique is detailed in prior work [6]. Given an action a that may execute an operation (such as create or read) over objects in set α, assuming that the action is invoked by a user with any role, is α necessarily a subset of the set of objects that users are permitted to operate on? If not, verification fails and reports an error.

Verification queries are translated both for Z3, an SMT solver, and Spass, a FOL theorem prover. We ran Z3 and Spass concurrently for each verification query. If either prover reaches a conclusive result with 60 seconds we accept it, otherwise, we halt both provers and mark the result as inconclusive. We consider this an acceptable time limit as very few queries for which we reach a conclusive result take more than a second. Spass outperformed Z3 in only 7 queries out of 18977.

Using our tool to verify models extracted from real world web applications using symbolic extraction, we found numerous bugs. These applications are publicly available and have been developed over years by multiple developers. Neither the original developers or verification by related approaches caught these bugs before us.

Most bugs we found are access control bugs. In particular, 303 invalidated verification queries relate to access control. For example, CoRM (a customer relationship manager) allows administrators to import data into the database by accessing a special admin panel. Our tool found that the access control check for managing imports is incorrect: anybody can import and export data if they know the correct URL.

Data integrity bugs were present as well, having caused 101 invalidated verification queries. For example, in Redmine (a project management web application), when a user is deleted, associated data is not properly cleaned up in the database. This leads to crashes and strange patterns in the user interface when this invalid data is accessed.

Column *Verification Queries* in Table III shows the number of verification queries that were generated for each application. Column *Avg Time per Query* shows the average time provers took per verification query to reach conclusive results.

Column *Verified* shows the number of verification queries that were proven correct, showing that an action preserves an invariant or correctly enforces some aspect of access control. Column *Bugs* shows the number of queries that reported an error, and we manually confirmed the error is caused by a bug. These queries demonstrate either a way to invalidate an invariant, or to expose restricted information to the user.

Column *False Positives* shows the number of queries that reported an error, but that were caused by a deficiency in our implementation instead of an application bug. All false positives we found were detected by an application creating objects that would invalidate the database, but never saving them in the database. Our implementation does not distinguish between objects that were saved in the database from those that are not, which we plan to improve on. Finally, column *Timeouts* lists the number of verification queries that caused our theorem provers to time out.

B. Verifiability

Our tool is open source and available online[2]. We also uploaded our experimental set online[3] so that our results can be reproduced. This includes all configuration and changes we made to run our tool. This will still require installation of ruby, rails, corresponding gems and database configuration.

To extract a model from an application, the developer needs to first install and setup the application. This process is different for each application, but typically requires installing an appropriate version of Ruby, associated gems, and setting

[2]http://bocete.github.io/adsl/

[3]http://tinyurl.com/symext

up the database. Even though we do not communicate with the database during symbolic model extraction, the application still has to establish a connection and validate the schema to start up. Finally, the model is extracted using the command:

```
rake adsl_translate
```

As for verification, our library has an executable called `bin/adsl-verify`. In addition to installing ruby and dependencies of our library, theorem provers Spass [31] and Z3 [10] need to be installed and added to the system path variable. After extracting a model, assuming it is stored in a file, it can be verified using command `bin/adsl-verify <file>`. Additional command line options can be used to get a more detailed verification report.

VI. Related Work

Near et al. [17] developed Rubicon, a web application verification tool that adds quantification to unit tests, for verification using Alloy [14]. Their symbolic execution is fully explained in a technical report [19]. Like them, we use the dynamic features of an unmodified Ruby runtime to override concrete methods with their symbolic counterparts. They both override methods with their symbolic counterparts only once, before symbolic execution has started. Considering that they use classical symbolic execution in a Ruby interpreter, there are some key differences between our methods: most importantly, we extract models from dynamically generated source code, and don't depend on SMT solvers for branch condition resolution. Recently, they used their symbolic execution technique to extract access control signatures from Rails programs [20]. Their experiments are limited to applications much smaller and simpler than ours. We suspect this is caused by their extraction method not supporting dynamic code generation.

RubyX [8] is a tool for symbolic execution in 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. Furthermore, symbolic model extraction does not rely on SMT solvers and a custom symbolic runtime. Because we do not use SMT solvers during model extraction, we are not limited to conditionals that can be specified in decidable logic fragments. We accomplish model extraction without a custom runtime that keeps track of symbolic values. Furthermore, they use DRails [13] to make specific usages of Rails code explicit, whereas we capture metaprogramming natively.

RailroadMap [16] is an automated tool for verification of access control in Rails using CanCan and Pundit. Their program analysis is limited to parsing a few specific Rails files and examining the AST, not even taking file dependencies into account. As such, their program analysis is difficult to make usable in practice. Their experimental evaluation focuses on small applications: all but a few had a single developer and were abandoned in weeks. We extracted semantically rich models from some of the applications they analyzed. We could not extract models from all of their applications because they were written in Rails 2, a long deprecated version of Rails, which our tool does not support.

This work relies on previous work for data model integrity verification [3], [4], [5] and access control verification [6]. These earlier contributions are on efficient detection of bugs and verification of Rails web applications using theorem provers. This is achieved by extracting models from Rails applications and translating them to first order logic. These earlier papers focus on translation of real world behaviors to first order logic in a way that will be analyzed by theorem provers quickly and with high probability of success. This paper, in contrast, focuses on model extraction. The extraction techniques we used previously were limited (e.g. they did not support branch conditions) and had a number of problems caused by an unprincipled solution that was implemented out of necessity. In addition, these previous papers did not elaborate on the extraction method they employed. This paper presents a novel model extraction method that is applicable to other programming and modeling languages.

Access control bugs are sometimes found with techniques not specifically tailored for finding access control bugs [17]. These methods typically require more effort than our automated method and may miss bugs.

Symbolic execution [15] is a well known technique for program analysis. Instead of executing source code in a normal runtime, symbolic execution will execute source code in an alternate runtime, operating on symbolic values instead of concrete values. These symbolic values are abstractions of concrete values. SAT and SMT solvers are used in branch conditions to determine if branch conditions are satisfiable, in order to guide path exploration for the purpose of testing. We use an unmodified Ruby runtime which makes our technique easier to implement, and we do not use solvers to resolve branch conditions as our purpose does not extend beyond extracting the model of a branch condition.

Concolic execution [28] extends on symbolic execution by keeping track of concrete values as well as symbolic. This is useful when solvers are not able to check satisfiability or find satisfying assignments to a branch conditions. One could look at our treatment of dynamic features as concolic, as we execute them concretely instead of symbolically.

VII. Conclusion

In this paper we presented an approach for model extraction from applications written in dynamically typed, interpreted languages. This is done by 1) using the source language interpreter in order to handle dynamic code generation, 2) instrumenting code so that executing it returns the model that corresponds to it, 3) implementing instrumentation in the source language in order to be able to instrument dynamically generated code, and 4) executing both branches of a branch statement in order to fully explore code. We experimentally demonstrate that symbolic model extraction can be used to

efficiently extract models of real world applications and to verify data integrity and access control properties.

REFERENCES

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

[2] activeadmin/activeadmin: The administration framework for Ruby on Rails applications., Aug. 2016. https://github.com/activeadmin/activeadmin.

[3] 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.

[4] 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.

[5] 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.

[6] I. Bocic and T. Bultan. Finding access control bugs in web applications with CanCheck. In *31st IEEE/ACM International Conference on Automated Software Engineering ASE 2016, Singapore*, 2016.

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

[8] 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.

[9] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[10] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[11] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of the 25th IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2010)*, pages 235–244, 2010.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[13] 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.

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

[15] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[16] 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.

[17] 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.

[18] 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.

[19] J. P. Near and D. Jackson. Symbolic execution for (almost) free: Hijacking an existing implementation to perform symbolic execution. Technical Report MIT-CSAIL-TR-2014-007, MIT, April 2014.

[20] J. P. Near and D. Jackson. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 947–958, 2016.

[21] J. Nijjar. *Analysis and Verification of Web Application Data Models*. PhD thesis, University of California, Santa Barbara, Jan. 2014.

[22] J. Nijjar, I. Bocić, and T. Bultan. An integrated data model verifier with property templates. In *Proceedings of the ICSE Workshop on Formal Methods in Software Engineering (FormaliSE 2013)*, 2013.

[23] J. Nijjar and T. Bultan. Bounded verification of Ruby on Rails data models. In *Proceedings of the 20th Int. Symp. on Software Testing and Analysis (ISSTA 2011)*, pages 67–77, 2011.

[24] J. Nijjar and T. Bultan. Unbounded data model verification using SMT solvers. In *Proceedings of the 27th IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2012)*, pages 210–219, 2012.

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

[26] ActsAsParanoid/acts_as_paranoid: ActiveRecord plugin allowing you to hide and restore records without actually deleting them., Aug. 2016. https://github.com/ActsAsParanoid/acts_as_paranoid.

[27] E. D. Sciascio, F. M. Donini, M. Mongiello, R. Totaro, and D. Castelluccia. Design verification of web applications using symbolic model checking. In *Proceedings of the 5th Int. Conf. Web Engineering (ICWE 2005)*, pages 69–74, 2005.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 05)*, pages 263–272, 2005.

[29] P. D. Stotts, R. Furuta, and C. R. Cabarrus. Hyperdocuments as automata: Verification of trace-based browsing properties by model checking. *ACM Transactions on Information Systems (TOIS 1998)*, 16(1):1–30, 1998.

[30] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[31] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *Proceedings of the 22nd Int. Conf. Automated Deduction (CADE 2009), LNCS 5663*, pages 140–145, 2009.