iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 6 **Issue:** IX **Month of publication:** September 2018

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 6.887
Volume 6 Issue IX, Sep 2018- Available at www.ijraset.com

Lightweight Authentication and Authorization Framework for Ruby-on-Rails Web Application Development

Mr. Aniket Patil[1], Mr. Harish Barapatre[2]
[1, 2]*Yadavrao Tasgaonkar Institute of Engineering and Technology*

Abstract: ***In a Web application framework suitable for a code-centric development approach, maintaining the fault-lessness of the security features is an issue because the security features are dispersed throughout the code during the implementation. In this paper, we propose a method and develop a static verification tool for Web applications that checks the completeness of the security features implementation. The tool generates a navigation model from an application code while retaining the security properties and then checks the consistency of the security properties on the model since access control is relevant to the application behavior. We applied the proposed tool to various Ruby on Rails Web application source codes and then tested their authentication and authorization features. Results showed that the tool is an effective aid in the implementation of security features in code-centric and iterative Web application development.***
Keywords: Static security analysis, access control, agile development, modeling Web application.

I. INTRODUCTION

Securing Web applications is an essential requirement because anything published on the Internet is exposed to various attacks. When using a Web application framework such as Ruby on Rails (shortened to Rails) [1], the frame-work itself takes countermeasures against typical Web application vulnerabilities. *Secure by default* became a trend in Rails as a security measure. For example, a string escape at the template became the default setting from version 3.0 for cross-site-scripting vulnerability. Another well known vulnerability specific to Rails is *Mass Assignment*. The configuration flag to guard against this vulnerability was introduced in version 3.1 (at that time, the flag was *false* (no protection) as a default, but it was changed to *true* from version 3.2.3). In this way, the framework provides a safe side default state and reduces potential vulnerabilities caused by coding bugs. The framework also provides a security guide that describes the best secure coding and setting practices.

However, security features such as authentication and authorization are part of the application design and are implemented by the application source code. In other words, mistakes made when coding the security features are at risk of becoming a vulnerability. To address this, many model-based approaches have been proposed for http://rubyonrails.org

Web application development [5][6][14]. However, code-centric development is still popularly practiced due to the immaturity and limitations of modeling tools [12].

Testing the implemented security features is therefore crucial in terms of developing a secure Web application. However, there are three primary challenges when it comes to such testing.

1) There is no adequate test oracle due to the lack of a written security design.
2) It is not suitable for test-driven development (TDD) since a security test must be exhaustive.
3) A conventional static and dynamic security verification tool cannot support application design tests.

A design documentation of distinct security features cannot be performed in the development of most Agile and code-centric Web applications, so it is difficult to test the security features due to the lack of a test oracle. The implementations tend to go ahead without a formal threat analysis or consideration of a security model. In general, simply incorporating a proven module that provides authentication or authorization is an easy way to develop the application.

TDD and behavior-driven development (BDD) are also being used in Agile Web application development [7][15]. There is no problem with testing the typical authentication and authorization behavior using the existing testing framework of TDD. However, to validate the security features comprehensively and exhaustively (e.g., to identify exceptional behaviors against attack), we need to prepare a large number of test cases. The problem here is that it is difficult to develop and maintain a huge amount of test cases because this kills the agility of the development.

Static security verification for the application code of Rails has been proposed. Chaudhuri et al. proposed a method to detect vulnerabilities such as SQL injection and Cross Site Scripting (XSS) by symbolic execution analysis of the application source code [10]. Brakeman is a practical tool to verify vulnerability by static analysis [9]. It presents CVE[2] information from the version of the pack-age used by the application and reports any vulnerabilities after inspecting the source code. Static verification is very fast compared to most Web application vulnerability scanners. In addition, it identifies the problem at the source [2]Common Vulnerabilities and Exposures, http://cve.mitre.org/code level, which makes it easy to check the error report on the code and fix the bug. However, the conventional static verification tool only detects vulnerabilities caused by known coding bugs or defects; it does not correspond to the confirmation of security features that are part of the design of the application. To test the security design, we need a document that describes security requirements and policy as a test oracle.

We feel that a static verification tool is potentially effective for testing the implementation of security features from the viewpoint of ensuring the comprehensiveness of the test. As noted above, the absence of documentation of security design is a problem when testing. Therefore, we take two novel approaches for security testing without a clear test oracle. First, we leverage the completeness of the implementation of security features as a test oracle. Inconsistency of the implementation occurs because the security features are dispersed in the source code and increasingly implemented. This can be detected by static analysis. This is the main contribution of this paper. Second, through the static analysis, we create a documentation of the security design. This document can be used for manual static analysis and is more efficient than reviewing all of the code. In addition, we complement the documented features to define the implicit requirement that now appears in the code and reuse this as the test oracle. This alleviates the false-positive problem of static verification.

Along with Agile and code-centric development, our tool detects inconsistencies that arise in the incremental implementation of security features. In addition, it is now easy to verify that the intended implementation is being carried out by the generated security feature documentation.

The rest of this paper is organized as follows. Section II gives background information on Web applications and the implementation of security features. Section III presents the details of our method and its implementation. In Section IV, we discuss the results of an evaluation of the tool being used with various applications. Section V discusses related works, and we conclude the paper in Section VI with a brief summary and a mention of our future work.

II. BACKGROUND

Access control features are implemented through the application code and configuration in code-centric Web application development. In this section, we provide a brief overview of the Web application framework we used in this paper, access control implementation, and common weaknesses using Rails.

A. Ruby on Rails

First, we briefly describe the Web application frame-work, Ruby on Rails, covered in this paper. Rails is one of the most popular Model-View-Controller (MVC) types of Web application framework and provides very efficient Web application development based on its various principles, including DRY: Don't Repeat Yourself, and

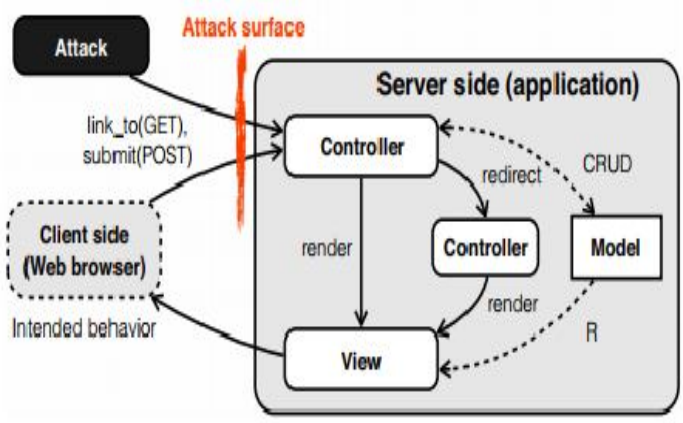


Figure 1 – Behavior of MVC-style Web applications.

CoC: By Convention over Configuration. All frameworks and applications are written in the Ruby programming language. This framework is also appropriate for test-driven development and code-centric Agile development.

Figure 1 shows the behavior of a typical MVC-style Web application. The content of the Web pages (HTML) are dynamically generated by the View (template code of Web page generation). Then, as a result of the browser side operation, a POST or GET HTML message is sent to the server side. The framework routes the request to the appropriate Controller based on the requested URL. The routed Controller processes the input message and then calls the View to generate the requested page. In some cases, the operation is redirected to another controller. The Model holds persistent data (connected to the Database) that is accessed from the View and Controller.

In general, the regular behavior of the Web application is defined by the Controller and View codes. However, a remote client can send any HTML message to the server side. Forced browsing is an attack in which a perpetrator tries to access restricted resources that are not referenced by the application itself. The "attack surface" of the Web application in such a case is all the controllers and their input messages. Therefore, the Controller must behave correctly and securely against any access (any HTML message from client).

B. Authentication and authorization

In this section, we discuss how to implement an access control on Rails. It is possible to implement authentication and authorization by the application's own code, but using the popular functional module is much more common. In this paper, we deal with Devise[3] for the authentication module and with CanCan[4] for the authorization module. When using them within Rails, we specify and install the packages and put the necessary commands and configuration into the application code.

Figure 2 shows a typical code example of the implementation when using Devise and CanCan. In the case of CanCan, the "app/models/ability.rb" code is equivalent to the Policy Decision Point (PDP). In this example, the administrator can only delete the entry of the User model. At the same time, the Policy Enforcement Point (PEP) is a command (authorize!) that is placed in each method in

https://github.com/plataformatec/devise/

https://github.com/ryanb/cancan

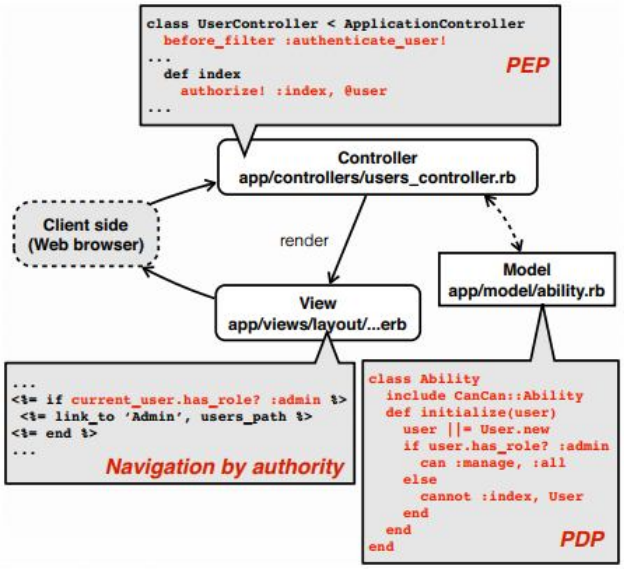


Figure 2 – Example of Devise and CanCan implementation.

The Controller class. In addition, the View has conditions that check the authority and generate the page based on the user's role.

The Devise also provides a variety of user interface codes (View and Controller) required for authentication features, including sign in, sign up (new user registration), and password update/reset. The programmer must add a hook (e.g., authenticate user! command) to the Controller code that requires authentication. When an un-authenticated client accesses the Controller that requires authentication, the access is automatically redirected to the sign in page (the flow is shown on detail in Fig. 4).

CanCan supports a Role-Based Access Control (RBAC)[16]. First, the programmer needs to provide a Model ("app/models/ability.rb") as a PDP and describe the access control rules by the code. The PEP is achieved by inserting a hook into the Controller in addition to the Devise hook. However, the CanCan supports three possible implementation patterns.

1) Class: Hook up all methods in the Controller class (same with Devise).
2) Method: Place the checks at each method.
3) Condition: Place the checks within the code (Con-troller and View).

This provides a flexible way to implement the authorization. However, it is also prone to implementation mistakes.

C. Classification of the weakness of Access Control Implementation

Table I summarizes the common implementation errors related to the access control of Web applications. Each row shows the location of the fault with weakness defined by Common Weakness Enumeration (CWE)[5].

[5]http://cwe.mitre.org/

Table I – Weakness and defect location of access control implementation.

Weakness (and error)	Defect location		
	PDP	PEP	Nav. (View)
Incorrect user management (CWE 286) Improper ownership management (CWE-282)	O		
Improper authentication (CWE-287) Improper authorization (CWE-285) Missing authorization(CWE-862) Incorrect authorization(CWE-863)		O	
Navigation error			O

For example, a coding mistake of PDP would lead to "CWE-282: Incorrect User Management" and "CWE-286: Improper Ownership Management". Fundamentally this is a design problem of access control model.

A problem with the implementation of the PEP can cause many weaknesses, such as "CWE-287: Improper Authentication", "CWE-285: Improper Authorization", "CWE-862: Missing Authorization", and "CWE-863: In-correct Authorization". If the programmer forgets to check the authority at the Controller, unauthorized access can be achieved. Even applications that do not have a transition to the Controller might miss the authorization check, leaving it vulnerable to attacks by forced browsing.

In addition, if the programmer forgets to check the authority at View to generate the link to the privileged Controller, an unexpected permissions error may occur. This is a navigation error of the application even if there is no problem in terms of security.

For secure implementation, the programmer must perfectly implement all of PDP, PEP, and Navigation in their entirety. Furthermore, the completeness of access control implementation must be maintained over the iterative development.

III. STATIC ANALYSIS OF ACCESS CONTROL IMPLEMENTATION

This section presents a method for testing the implementation of authentication and authorization by static verification. We focus on the completeness of the implementation of access control. Thus, the goal is to detect the following problems that may occur over the course of the implementation. These inconsistencies can be checked by the static verification of source code.

1) Missing PEP with correct PDP definition.
2) Incorrect PDP with correct PEP placement.
3) View code without authorization check.
4) Missing PEP at destination of View code with authorization check.
5) Vague policy definition

This last one has slightly different characteristics. The verification of a policy itself is difficult by a static analysis of the application code, but vague and ambiguous policy definitions can be detected.

Table II shows a list of warnings and severity with relevant code location. "o" denotes proper definition or

Table II – List of warnings and severities.

Warning	Sever-rity	Req.	PDP	PEP	Nav.
Complex policy definition (CPD)	low	–	x	–	–
Improper policy 1 (IP1)	high	–	x	o	–
Improper policy 2 (IP2)	high	–	x	–	o
Missing authentication 1 (MT1)	low	x	x	x	–
Missing authentication 2 (MT2)	high	o	-	x	–
Missing authentication 3 (MT3)	mid	x	o	x	–
Missing authorization 1 (MZ1)	high	–	o	x	–
Missing authorization 2 (MZ2)	high	–	o	x	o
Improper navigation (IN)	mid	–	o	o	x

Existence, “x” denotes missing definition or existence, and “–” denotes do-not-care. Most of the requirements can be elicited from the code. However, there are implicit requirements that can not be elicited at the code level, so we added “requirements” to clarify the implicit requirements. These must be provided to reduce false-positive warnings. The severities are defined as High: obvious vulnerability, Mid: inconsistent implementation and application bug, and Low: need requirements.

A. Static Analysis Tool for Rails

We developed a tool called “RailroadMap” that generates a navigation model from the application source code and checks the access control implementation. The tool is written in Ruby and uses Abstract Syntax Tree to analyze the source code. There are five processing steps (Fig. 3).

1) Parse the Controller, View, and Model source code on an application framework and generate a navigation model that abstracts the Web application behavior using a state machine. By converting the application code to a generic model that preserves the security features, the verification of the security feature implementation is easy to perform
2) Generate access control policy from the PDP code (app/models/ability.rb). Checking for vague policy definitions is performed at the same time.
3) Check consistency of access control implementation listed in Table II.
4) Generate a report in various forms.
5) Update and fix the code and security requirements to resolve the problems.

The report contains much useful information on how to fix and review the existing code. For example, we can review and confirm the policy definition by the ACL table. The warning message helps the programmer to determine the next action by providing the location of the problem, type of problem, and remedy.

B. Extraction of Access Control Policy

When the application uses CanCan as a role-based authorization module, the PDP is an “app/models/ability.rb” code: this also includes policy definition in the form of a conditional expression, as shown in the example in Fig. 2. It defines the action for the object classified by the

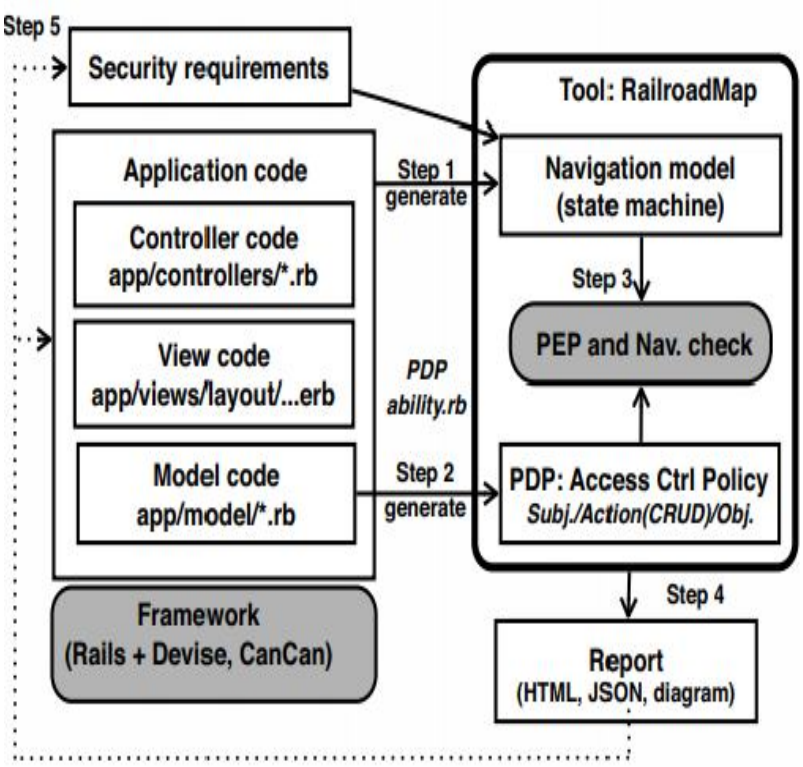


Figure 3 – Operation flow of proposed tool.

Subject (role) condition: for example, a role r in which set R denotes roles defined by code ($r \in R$). Similarly, an asset/object ($a \in A$) and operation/action ($o \in O$) are extracted from code. The asset is Model which is a persistent storage of the application (database). The basic operation types are "create", "read", "update", and "delete" (CRUD). Then, the tool provides an $ACL[r][a][o]$ table as a result of parsing the code for the manual inspection of the policy definition. This also supports the assignment of multiple roles to a single user.

However, we noticed that there is a complex conditional expression to define the policy. In such cases, using the simple ACL table is difficult because there is a dependency between roles. Using a complex conditional expression to define the policy is not a bug, but an unnecessary complexity is undesirable. We feel it is better to keep things as simple as possible and define the action for an object by individual role. Therefore, the tool supports a simple check of the existence of ELSIF and nested IF condition and issues a low severity warning, "complex policy definition (CPD)".

C. Requirements for dealing with ambiguity of code

The policy definition is not exhaustive. We treat "un-defined" access control in policy as deny of access. However, "unplaced" PEP is allow of access. As a result, "undefined" and 'unplaced" both mean allow of access. Such an implicit assumption causes ambiguity and is difficult to test by static analysis of code. Currently, we raise the low severity warning for such implicit states. This can be reduced by specifying a requirement or by adding an explicit code. We use the requirements in order to complement the ambiguity on the code. A set Q is requirements. Access state $s \in Q$ denotes allow of access.

D. Generating navigation model

A navigation model is an abstracted behavior of Web application in the form of a state machine. We assign each method of Controller class (app/controllers/*.rb) and corresponding View code (app/views/*/*.erb) to a state. Therefore, the behavior of the application is a set of the transition between Controller and View, as shown in Fig.

Table III – Correspondence between application code and navigation model.

Code type	File	Scope	Navigation model
Model (CanCan)	app/models/ability.rb	class (par file)	$r \in R$, $a \in A$, $o \in O$, (Role,Asset,Operation) *ACL*[*r*][*a*][*o*]*table*
Model (Ruby/ORM)	app/models/[*a*].rb	class (par file)/code	$v \in V$ (Variables)
Controller (Ruby)	app/controllers/[*a*] controller.rb	method (= *o*)	$s \in S_C$ (Controller state)
		command (redirect) command (render) command (sec. funds) variable	$t \in T_{CC}$ (Transition from C to C) $t \in T_{CV}$ (Transition from C to V) *g* (Guard of trans. and attribute of the state) $v \in V$ (Variables)
View (template, ERB)	app/views/[*a*]/[*o*].erb	file	$s \in S_V$ (View state)
		condition and command (link to, submit)	$t \in T_{VC}$ (Transition from V to C) with *g* (Guard of trans.)

Each state corresponds to a URL of the Web application. In addition, an access control command relevant to each state is recorded as an attribute of that state.

Rails commands such as “redirect”, “render”, “link to”, and “submit” mean that a transition from the state contains the command to the state that the command indicated. Table III summarizes the mapping between code and navigation model. If there are multiple transitions from the state, the destination state is selected by each guard condition calculated from the application’s internal state (T_{CC} and T_{CV} in Table III). Alternatively, there are multiple destinations based on what the users select at the browser side (T_{VC} in Table III). A transition to the privileged state may have a guard condition that checks the authority of the application user.

Variables that show the internal status and can be used in the guard conditions to control the application behavior need to be modeled. At the very least, the tool should support all related variables of access control features such as session information since these variables need to check for improper navigation errors (*V* in Table III).

From the access control point of view, the means of *a* (asset) and *o* (operation) correspond only to the file path or method name since “Convention over Configuration” is the principle of Rails. In the navigation model, the controller state is equivalent to the *a* + *o* of ACL. The command used for access controls is stored in the state as an attribute flag. This flag has three values: true, false, and nil. If there is no enforcement at this stage, the flag is nil and an implicit assumption may exist for this. Otherwise, it has explicit values defined by the relevant command. Note that the security requirements reduce this implicit assumption (Step 5 in Fig. 3).

At present, the tool supports Ruby and embedded Ruby (ERB, *.erb files as View template) code. Other languages used by the Rails application such as HTML Abstraction Markup Language (Haml) and JavaScript are not yet sup-ported. Haml is used as a templating language instead of the ERB, while Asynchronous JavaScript and XML (Ajax) is frequently used on the client side to create asynchronous Web applications. It is not really necessary to have access control since the PDP and PEP are implemented on the server side. However, the extraction of client-side behavior controlled by the JavaScript code has some limitations.

E. Security check using navigation model

We verify the security features recorded in the navigation model with the access control policy. At this stage, we assume that the policy definition is correct.

First, we check the consistency between the policy definition and the placement of PEP at the Controller. If there is any missing PEP against policy, the tool issues a “missing authorization 1 (MZ1)” warning with high severity. An authentication test is a

confirmation of the presence or absence of the placement of the authentication check command in the Controller code (attribute at state). Algorithm 1 is an inconsistency detection algorithm for PEP placement.

Algorithm 1 PEP check.

Require : ACL, S Nav. model
Require : Q Requirements
Require : $W \leftarrow 0$ Warning

```
 1: procedure CHECKPEP
 2:     for each s ∈ S_C do
 3:         res1 ← hasAuthenticationCheck(s)
 4:         res2 ← hasAuthorizationCheck(s)
 5:         res3 ← hasPolicyDefinition(s)
 6:         if ¬res1&¬res2&¬∃s ∈ Q then
 7:             W ← W ∪ NewWarning(MT1)
 8:         else if ¬res1&∃s ∈ Q then
 9:             W ← W ∪ NewWarning(MT2)
10:         else if ¬res1&res3&¬∃s ∈ Q then
11:             W ← W ∪ NewWarning(MT3)
12:         end if
13:         if res3&¬res2 then
14:             W ← W ∪ NewWarning(MZ1)
15:         end if
16:     end for
17: end procedure
```

Second, we check the consistency between the privileged Controller and the transition from the View to the Controller. If there is an un-guarded transition to the privileged Controller, the tool issues an “improper navigation (IN)” warning with medium severity. We use this navigation model because it is easy to trace the relationship between the transition of the application and authority.
At the same time, a missing policy definition at the Controller state and the guard condition of the transition despite the authentication check is reported as an “im- proper policy (IP1, IP2)” warning with high severity. There are two description patterns for the authorization checks at the View.

```
1 <%= if can? :create, Project %>
2 <%= link_to ’New Project’, new_project_path %>
3 <%= end %>
```

Listing 1 – View with authority check 1 (AC1).

The check of Listing 1 is based on *o* (operation: create) and *a* (asset: Project). The target of this guard must be same with the destination Controller. This evaluation is provided by the PDP. Therefore, the destination Controller must have a consistent PEP and *ACL* (AC1).

```
1 <%= if user.has_role? :admin %>
2 <%= link_to ’New Project’, new_project_path%>
3 <%= end %>
```

Listing 2 – View with authority check 2 (AC2).

The Listing 2 example just verifies the role (subject) of the current user. This can cause a navigation error if the admin role does not have permission to create a new project. In addition, this guard does not check with the PDP, so inconstancy between the guard and the ACL might occur (AC2).

Algorithm 2 is a detection algorithm of improper navigation and missing authentication based on the recursive call to traverse the navigation model. This test is executed by each role *r* and several start states to cover as many transitions as possible. In addition, it is necessary to limit the depth of the call to avoid an infinite loop. We recorded the transition history and calculated the coverage as a metric for adjusting the start state and search depth. The *EvaluateGuard* function evaluates the guard expression. It returns five values depending on the coding type of guard: *deny* and *allow* are the judgment by the PDP (AC1), *true* and *f alse* are the judgment by condition (AC2), and *noguard* is for other cases.

F. Report

After the security check, the tool generates a report in various forms, including HTML, JSON, and dot (state diagram). The report in the HTML form clearly shows the access control table, list of warnings, and the navigation model. By referencing this report, the programmer understands the current status of the implementation and determines the action to be taken next.

G. Agile security feature implementation with static analysis

One of the benefits of the static verification tool is that anyone can run it from a command line at any time. The first run must be at a stage at which the implementation has proceeded to a sufficient level. If there is a problem to solve or any ambiguity, fix the code and define the security requirements to clarify them. Once a stable condition is maintained, run the tool after making changes to the code or adding new code. If a problem occurs with the requirement, review and decide whether to fix the code or the requirement. Using the tool enables us to ensure the quality of the security features throughout the iterative development.

Algorithm 2 Navigation check.

```
Require: R, ACL, S, T                          Nav. model
Require: ss ∈ S_V                              Start state
Require: r ∈ R                                 Role
Require: W                                     Warning
 1: procedure CHECKNAV(ss, r)
 2:     for each t in GetTransFrom(T_VC, ss) do
 3:         sd ← GetDstState(t)
 4:         if isVisited(sd, r) then           detect infinite loop
 5:             return                         escape
 6:         else
 7:             setVisited(sd, r)              set flag
 8:         end if
 9:         g ← GetGuard(t)
10:         o ← GetOperation(sd)
11:         a ← GetAsset(sd)
                ← EvaluateGuard(g, r, o,
12:         res1 a)                            Trans
13:         res2 ← EvaluatePEP (sd, r)         State
14:         if res1 = deny then                guard: AC1
15:             if res2 = deny then            end of transition
16:                 return
17:             else                           inconsistent V and C
18:                 W ← W ∪ NewWarning(IN )
19:             end if
20:         else if res1 = allow then          guard: AC1
21:             if res2 = allow then
```

```
22:              W ← W ∪ N ewW arning(M Z2)
23:          end if
         else
24:      if     res1 = true then            guard: AC2
25:          if  res2 = allow then       improper guard
26:              W ← W ∪ N ewW arning(IN )
27:          end if
         else
28:      if     res1 = f alse then          guard: AC2
29:          if  res2 = deny then        improper guard
30:              W ← W ∪ N ewW arning(IN )
31:          end if
32:      else                                  no guard
33:          if  res2 = allow then
34:              W ← W ∪ N ewW arning(IN )     FP?
35:          else if  res2 = deny then
36:              W ← W ∪ N ewW arning(M Z2)
37:          end if
38:      end if
39:      return CHECKNAV(sd, r)          recursive call
40:  end for
41: end procedure
```

Rails scaffolding provides a quick way to add new resources: it automatically generates runnable MVC code, and then the programmer must add security features since the generated code does not support any itself. Our tool provides the metrics of the current implementation status of the security feature by warning messages, and the programmer can thus proceed with stepwise implementation of the security features by using the metrics. A typical example is the following:

1) Update/modify policy definition.
2) Update/modify PEP in Controller.
3) Update/modify link and guard condition in View

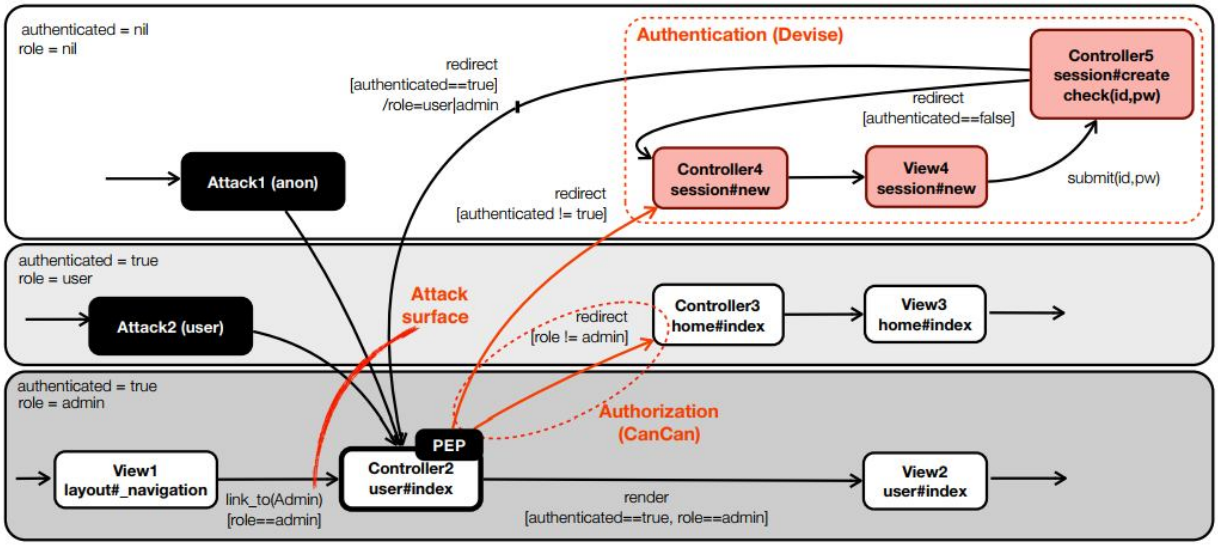


Figure 4 – Navigation model around Devise authentication and CanCan authorization.

In each step, the programmer can concentrate on the same type of coding work. This will greatly improve the coding speed and quality. We feel this kind of tool is especially useful for the novice programmer.

IV. EMPIRICAL EVALUATION

To evaluate the operation and advantages of our security assessment method and tool, we performed two approaches while using the tool to develop a sample ap-plication and then tested existing applications. Before we report these approaches, we explain the actual navigation model of Devise and CanCan.

A. Navigation model of Devise/CanCan

"Rails App for Devise with CanCan and Twitter Boot-strap" [6] is a well known tutorial application that supports an admin role only. Figure 4 shows the generated navigation model from an example that explains the typical behavior of authentication by Devise and authorization by CanCan. The state name is set by *Asset#Operation* of the Controller and the View. We use two variables, *authenticated* and *role*, to model the access control behavior.

In this example, Controller2 *user#index* must be accessed by the privileged user, *role* = *admin*. There are two attack scenarios against this state.

1) Attack 1: Access by unauthorized user Forced access to this state by a non-authenticated user is redirected to the sign-in page (Con-troller4). If the application user provides the correct ID and password, the user is redirected to Controller2 with attribute *authenticated = true*.

2) Attack 2: Access by non-privileged user Forced access to this state by a normal user,

https://github.com/RailsApps/rails3-bootstrap-devise-cancan

Table IV – Access control list.

Role	Assets		
	All	Article	Comment
Admin	CRUD	-	-
Moderator	-	RD	RD
Reporter	-	CRU	R
Member	-	R	CRU

(C: Create, R: Read, U: Update, D: Delete)

Table V – Metrics.

Development steps	State	Trans	Warngs		
			MT3	MZ1	IN
Step 0. Original app.	46	60	0	0	0
Step 1. Add new assets	72	114	14	0	0
Step 2. Update PDP	72	114	14	6	0
Step 3. Update PEP	72	114	0	0	145
Step 4. Update View	72	114	0	0	12
Step 5. Suppress FP	72	114	0	0	(12)

role = user, is redirected to the home page (Controller3) with an error message.

Therefore, programmers must place the PEP code care-fully in the privileged states. Even a small implementation bug can lead to a serious vulnerability of the Web application.

View1, *layout# navigation*, is dynamically generated by the template. The selectable destinations listed on the page content are changed by the application's internal state (=variables). In this example, only the user that has the "admin" role has the link to Controller2 and the behavior is modeled by the guard condition of the transition between View1 and Controller2.

B. Empirical evaluation with example application development

We can use the warnings reported by the tool as metrics that identify the implementation status of security features

Table VI – Test results

Ruby on Rails Applications	Brakeman warnings			RailroadMap									
				Navigation Model and Policy					warnings				
	MA	XSS	SQL inject-ion	No. of states	No. of Transi-tion .	No. of Role	No. of Obj.	PEP type	Complex policy def. (CPD)	Missing authen-ticatio n (MT1+3)	Missing authori-zation 2 (MZ1)	Missing author i-zation 3 (MZ2)	Improper nav. (IN)
9balloons	0	0	0	60	92	2	2	a	1	9	1	11	8
Artdealer	0	0	0	58	74	2	2	a	2	1	0	3	0
avare	7	0	0	89	125	6	5	b, c	8	14	4	1	18
Communautaire	4	0	0	94	111	2	2	a	3	30	4	5	2
consultorio	8	0	0	94	84	2	1		1	33	2	0	0
fast-ticket	2	0	0	48	73	2	2		1	7	2	0	0
illyan	2	0	0	97	120	2	2	a, b	3	30	3	18	18
s2l	16	0	2	146	216	5	8		15	60	12	13	19
shiroipantsu	0	1	1	92	119	4	3	a	6	25	1	11	22
talks	3	0	0	135	248	3	3	b, c	4	63	9	2	74
wm-app	22	0	0	215	285	3	1	a, b	2	0	2	76	135
zmchapters	0	0	0	174	223	3	11	a, b	17	66	2	57	85
rails3-bootstrap-devise-cancan	0	0	0	46	60	2	1	b	0	0	0	0	0

In order to investigate the relationship between the security features implementation and the metrics, we developed an example Rails application with this tool. We extend this example application (described in the previous section) to a simple content management system. The original application simply had an "admin" role and a generic asset, "all". We added two assets ("Article" and "Comment") and three roles ("moderator", "reporter", and "member") and defined the access control list as shown in Table IV.

In this evaluation, we check the metrics by the six steps shown in Table V.

Step 0: Original application

First, we make sure that the original tutorial does not have any security warnings (step 0).

Step 1: Adding new assets

We then added two assets by using the "scaffold" command. This command creates a template code of the model, view, and controller for a new asset in a single operation. The template code provides a scaffold and it is ready to run and test. However, the

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 6.887
Volume 6 Issue IX, Sep 2018- Available at www.ijraset.com

scaffolding does not generate a code with a security feature, so RailroadMap reports authentication errors (MT3) for new as-sets (step 1).

Step 2: Adding new policy

In step 2, we enhance the RBAC policy, as shown in Table IV. After this, RailroadMap starts to issue "missing authorization (MZ1)" warnings.

Step 3: Adding PEP

In step 3, we add PEP (code for authentication and authorization) to the Controllers until we eliminate all of the MT3 and MZ2 warnings. This work is easy since the warning messages navigates the location of the missing PEP. How-ever, after the PEP has been located, the tool starts issuing many "improper navigation (IN)" warnings.

Step 4: Correcting navigation behaviors

In step 4, we fix the View codes to eliminate all IN warnings. There are many transitions that cause authorization errors (navigation errors) since the scaffolding generates all possible transitions as the template. After the fix, 12 warnings still remain.

In general, the static analysis tool has some limitations in both capability and accuracy of the code analysis.

These limitations cause false-positive warnings that are identified by manual inspection and review. At present, our tool does not support some of the conventional role situations. If all roles have read access privilege to the same resource, no guard is needed for these transitions in a practical sense. They thus become false-positive warnings. To identify these false-positive warnings, the tool supports a suppressing-warning list (simple hash table based on the warning message). After adding the 12 warnings to the suppressing-warning list, all warnings are gone and access control is successfully implemented (step 5).

C. Testing real applications

In order to evaluate the tool and confirm the presence or absence of problems in actual implementation, we searched for Rails applications published on GitHub[7] that use both Devise and CanCan and verified the consistency of the implementation of the authentication and authorization of 13 applications, as shown in Table VI.

First, to compare the quality of target applications, we applied Brakeman [8], a popular static checker, against them. For Brakeman, we simply picked these three common weaknesses detected at the code as a reference of code quality. Note that Brakeman reports many warnings caused by using an older and more vulnerable version of Rails. We did not include these warnings in our numbers. The result is shown in the left-hand columns of Table VI. The Rails framework supports a security function to prevent common Web application vulnerabilities. Thus, there are few cross-cite scripting (XSS) and SQL injection warnings. In contrast, there are many "Mass Assignment (MA)" warnings that are a unique vulnerability of Rails. Of course, these warnings contain many false-positive warnings, but they do need to be identified manually.

The statistics of the navigation model and policy of each application are shown in the middle columns. The model size of the applications taken as the sample are varied. The PEP type indicates the implementation styles of CanCan. There are three types, "a", "b", and "c", as shown in section II-B. The blank means there is no PEP in the Controller or View codes.

The right-hand columns indicate the number of problems detected by RailroadMap. We omitted the "missing authentication (MT2)" test since there are no given security requirements for these applications. Also, the "improper policy" test was omitted due to the unclear policy definition at present. The reason the "missing authentication" has so many warnings is that it contains false-positive warnings resulting from the absence of security requirements. It is quite difficult to test the security without explicit, concrete security requirements, so we created a security requirements to clarify the security design and implementation.

The implementation level of each applications is un-known, as they were merely picked from a public code repository. However, we found that there was an in-completeness of access control implementation in many applications. This is thought to be due to the following facts.

1) Need to add a code to dispersed locations to enable policy enforcement.
2) Variety of possible implementation methods.
3) Ambiguity of the policy and enforcement definition by code.
4) Use of improper reference and example codes.

However, these problems can be identified and fixed by the proposed technique. As shown in the previous section, the test metrics provided by the tool help with the robust implementation of security features.

V. RELATED WORKS

Alalfi et al. surveyed modeling methods for Web applications [1]. They categorized the methods into four levels: interactive behavior modeling, content modeling, navigation modeling, and hybrid modeling. Our navigation model supports both interactive behavior and navigation of dynamic Web applications.

The model-driven approach is considered appropriate for secure software engineering but is not very accepted in code-centric development. It requires a good model-to-code generator because otherwise the developer must keep both the model and the code up to date by hand An alternative practical approach is creating a model from application source code or trace to handle actual applications by the model. Andrews et al. proposed a testing method for Web applications by using FSM to build models from source code and constraints defined by the tester [2][3]. This approach suffers from a state space explosion problem since it focuses on detailed behavior of Web applications by defining fine-grained FSM. In contrast, we define a model using coarse-grained FSM to represent security features efficiently. Sprenkle et al. generated a navigation model from an application trace and created abstracted test cases for Web applications written in Java [17]. They used execution traces to create the model. This approach cannot provide comprehensive abstraction without preparing a comprehensive test. Our approach creates the navigation model from the application code to avoid this drawback. Security testing of Web applications written in dynamic languages such as PHP, Perl, Python, and Ruby is a relatively new research area. Sun et al. generated “sitemaps” per roles from the PHP Web application source code and detected access-control vulnerabilities [18]. Our approach focuses on the faultlessness of the implementation as the test oracle. Gauthier et al. presented a tool for the detection and repair of access control vulnerabilities in PHP Web applications [13]. The tool also identifies inconsistencies in access control implementation of the application code through the modeling. Our approach is much focused on development process and helps the robust access control implementation by categorizing the warnings to fault lo-cations. There is currently not much research or tools on static security analysis for Ruby and Rails. Chaudhuri and Foster used symbolic execution to check for vulnerabilities (XSS, CSRF, session manipulation, and unauthorized access) in Rails [10]. They converted Ruby code into AST and wrote the parser in OCaml. It was not easy to maintain support for the latest Ruby and Rails. RailroadMap is written in Ruby so that it can be thoroughly integrated with Rails. Doupe et al. proposed a static checker for execution after redirect (EAR) vulnerabilities [11]. However, the scope of the code to detect EARs was very small. Collins developed a static security analysis tool called Brakeman [9]. The tool is light-weight and easy to use with the application development practices. Our method and tool are intended to resolve inadequate access control design and implementation by checking the inconsistencies of the security implementation and eliciting the security requirements.

Agile development is closely related to code-centric Web application development. Beznosov et al. pointed out issues between agile development processes and conventional security assurance techniques [8]. Bartsch interviewed agile developers about security in actual agile development projects [4]. His findings suggested that developers should focus on adequate customer involvement, developer security awareness and expertise, and continuously improving their development process to ensure security. Our approach addresses these issues by providing the tool to support a static test of access control implementation.

VI. CONCLUSION

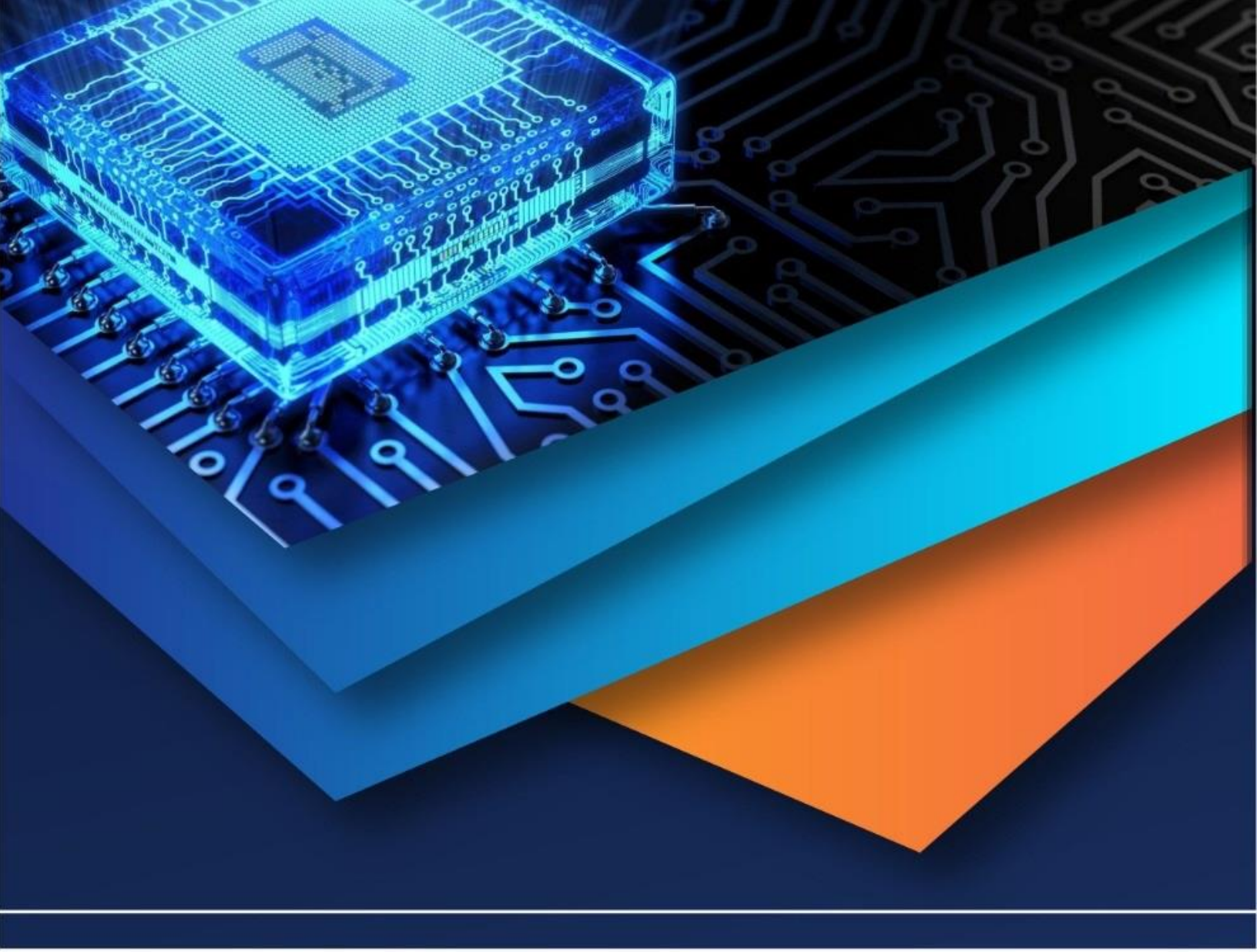
We developed a tool that performs static verification of access control errors of Rails and conducted testing with 13 published Rails applications. Results showed that many of the applications have obvious implementation miss of access control features. We also demonstrated the usage of the proposed tool through a sample applica-tion development. The tool supports the development and implementation process of security features by reporting warnings in the code.

False-positive warnings, which pose a problem for static verification tools, can be reduced by defining the require-ments to clarify the intended security design. In addition, warnings due to insufficient capability of the tool are sup-pressed by a suppressing-warning list. Once an application has become consistent, the tool can immediately detect accidental inconsistencies that occur during an iterative development.

For future work, we intended to further improve the verification capability and accuracy of the tool. So far, it can support Devise and CanCan, but it should be able to support other security modules as well. In the future, we want to enhance the tool to generate test cases that sub-stantiate the problems found in static verification. The tool can be integrated with other static verification tools such as Brakeman and build bridges between static security tests and test-driven development. We have implemented the tool in Ruby and made it suitable for Rails because an integration with the development environment is crucial in order for programmers to use the tool seamlessly. However, modeling security behavior from code is a useful method in code-centric agile development in general. The metrics provided by the tool are useful for implementing security features and reducing the burden on the program-mer.

REFERENCES

[1] M. Alalfi and J. Cordy, "A survey of analysis models and methods in website verification and testing," in Proceedings of the 7th international conference on Web engineering (ICWE07), July 2007, pp. 306-311.
[2] A. Andrews, J. Offutt, and R. Alexander, "Testing web applications by modeling with FSMs," in Software and Systems Modeling, 2004, pp. 1-28.
[3] A. Andrews, J. Offutt, C. Dyreson, C. J. Mallery, K. Jerath, and R. Alexander, "Scalability issues with using FSMWeb to test web applications," Information and Software Technology, vol. 52, no. 1, pp. 52-66, 2010.
[4] S. Bartsch, "Practitioners Perspectives on Security in Agile Development," in Proceedings of the Sixth International Conference on Availability, Reliability and Security, Aug. 2011, pp. 479-484
[5] D. Basin , J. Doser, and T. Lodderstedt, "Model driven security: From UML models to access control infrastructures, " in ACM Transactions on Software Engineering and Methodology (TOSEM), vol.15 no.1, pp. 39–91, January 2006
[6] D.A. Basin, M. Clavel, and M Egea, "A decade of model-driven security," in Proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT), June 2011, pp. 1–10
[7] K. Beck, "Test-Driven Development. By Example," Addison Wesley, 2003.
[8] K. Beznosov and P. Kruchten, "Towards agile security as-surance," in Proceedings of the the 2004 workshop on New security paradigms (NSPW 04), Sept. 2004, pp. 47–54.
[9] J. Collins, "Keeping Rails Applications on Track with Brake-man," RailsConf2012, April 2012.
[10] 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 10), Oct. 2010, pp. 585–594,
[11] A. Doupe, B. Boe, C. Kruegel, and G. Vigna, "Fear the EAR : Discovering and Mitigating Execution After Redirect Vulnerabilities," in Proceedings of the 18th ACM conference on Computer and communications security (CCS 11), Oct. 2011, pp. 251–262.
[12] A. Forward, "Problems and Opportunities for Model-Centric Versus Code-Centric Software Development: A Sur-vey of Software Professionals" in Proceedings of the 2008 international workshop on Models in software engineering (MiSE08), May 2008, pp. 27–32.
[13] F. Gauthier, and E. Merlo, "Fast Detection of Access Control Vulnerabilities in PHP Applications," in Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), Oct. 2012, pp. 247–256.
[14] J. Jensen, and M.G. Jaarun, "Not Ready for Prime Time: A Survey on Security in Model Driven Development," In-ternational Journal of Secure Software Engineering, vol. 2, no. 4, pp. 4961, 2011.
[15] D. North, "Introducing BDD", 2004 [Online]. Available: http://dannorth.net/introducing-bdd
[16] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based Access Control Models," IEEE Computer 29, 3847, 1996
[17] S. Sprenkle, L. Pollock, and L. Simko, "A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications," in Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verifi-cation and Validation, Mar. 2011, pp. 230–239.
[18] F. Sun, L. Xi, and Z. Su, "Static detection of access control vulnerabilities in web applications," in Proceedings of the 20th USENIX conference on Security (SEC'11), Aug. 2011, pp. 11–11.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY