Automated Reverse Engineering of Role-based Access Control Policies of Web Applications[⋆]

Ha Thanh Le[a,1], Lwin Khin Shar[b,1,*], Domenico Bianculli[c], Lionel Claude Briand[c,d], Cu Duy Nguyen[e,1]

[a]*Eltien & Co., Australia*
[b]*School of Computing and Information Systems, Singapore Management University, Singapore*
[c]*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg*
[d]*University of Ottawa, Canada*
[e]*Cyberforce Department - POST Luxembourg, Luxembourg*

Abstract

Access control (AC) is an important security mechanism used in software systems to restrict access to sensitive resources. Therefore, it is essential to validate the correctness of AC implementations with respect to policy specifications or intended access rights. However, in practice, AC policy specifications are often missing or poorly documented; in some cases, AC policies are hard-coded in business logic implementations. This leads to difficulties in validating the correctness of policy implementations and detecting AC defects.

In this paper, we present a semi-automated framework for reverse-engineering of AC policies from Web applications. Our goal is to learn and recover role-based access control (RBAC) policies from implementations, which are then used to validate implemented policies and detect AC issues. Our framework, built on top of a suite of security tools, automatically explores a given Web application, mines domain input specifications from access logs, and systematically generates and executes more access requests using combinatorial test generation. To learn policies, we apply machine learning on the obtained data to characterize relevant attributes that influence AC. Finally, the inferred policies are presented to the security engineer, for validation with respect to intended access rights and for detecting AC issues. Inconsistent and insufficient policies are highlighted as potential AC issues, being either vulnerabilities or implementation errors.

We evaluated our approach on four Web applications (three open-source and a proprietary one built by our industry partner) in terms of the correctness of inferred policies. We also evaluated the usefulness of our approach by investigating whether it facilitates the detection of AC issues. The results show that 97.8% of the inferred policies are correct with respect to the actual AC implementation; the analysis of these policies led to the discovery of 64 AC issues that were reported to the developers.

Keywords: Access control testing, reverse engineering, access control policies, machine learning

1. Introduction

Web applications are widely used in various domains, from e-government, information management, healthcare, social networks and media, to finance and education. These systems manage and share a large amount of sensitive information and resources among various users in distributed, dynamic environments. An essential requirement for such systems is to ensure security and privacy for the users and their data. One way to guarantee security and privacy in these systems is to define and enforce *access control (AC)*

[⋆]This work has been partially supported by the National Research Fund, Luxembourg, grant agreements FNR/P10/03 and INTER/DFG/14/11092585.

[*]Corresponding author

Email addresses: `lkshar@smu.edu.sg` (Lwin Khin Shar), `domenico.bianculli@uni.lu` (Domenico Bianculli), `lionel.briand@uni.lu` (Lionel Claude Briand), `duycu.nguyen@post.lu` (Cu Duy Nguyen)

[1]Part of this work was done while the author was affiliated with University of Luxembourg.

policies [1], to restrict access (by users) to sensitive resources (e.g., a page with personal information) of the system.

Similarly to other functionalities of an application, the implementation of AC policies has to be tested against the specifications [2, 3, 4, 5]. More precisely, AC policy specifications can serve as test oracles and be used to generate test cases. However, in many scenarios, testers cannot rely on AC policy specifications because they are often missing (e.g., the AC policies have been directly hard-coded in the business logic of the application), poorly documented, or hard to understand (e.g., AC policies are encoded as bit strings in configuration files). A recent literature study by Daoudagh et al. [6] reported that many critical AC issues arise from ambiguous or faulty AC specifications.

In this paper, we address the problem of reverse-engineering AC policies from the AC implementation of a Web application. Our approach learns and infers role-based access control (RBAC [7]) policies from implementations, which are presented to security engineers. Our objective is to assist security engineers in validating AC implementations with respect to intended access rights and detecting AC issues. Our work is set in contexts where AC policy specifications are missing or existing specifications are ambiguous. We assume that the intended access control-related behaviors reside only in the mind of security engineers, which is often the case for Web applications that are poorly documented or maintained [8], as it was also the case for our industrial partner. We also consider contexts where the application source code is not available, as it is often the case for 3rd-party or proprietary applications (like the one provided by our industrial partner and further used for the evaluation of the approach presented in this paper).

In previous work [9], some of the authors presented a semi-automated framework for the extraction of AC policies from Web applications. The framework first automatically exercises the resources of a target Web application with a set of known users (and roles) and then applies a machine learning technique to infer AC policies based on the outcome of the resource access requests. It also annotates the extracted policies with confidence levels that can pinpoint potential AC issues (e.g., implementation errors). In this paper[2] we propose the *ReACP* framework, which extends previous work [9] along two dimensions:

Policy extraction. In previous work, the definition of the steps involved in extracting AC policies was preliminary. *ReACP* extends those steps and includes new techniques for extracting AC policies more accurately. More specifically,

- In Web applications, many resources are reachable only when meaningful data are provided in an access request. Without such meaningful data, the framework would not be able to discover resources and thus learn the corresponding AC policies. In *ReACP*, we apply Xinput, an XML schema for systematic specification of domain inputs for Web applications. Xinput allows domain experts to specify data types (e.g., string, numeric) and other input specifications using boundaries, enumerations, and length.
- *ReACP* proposes and implements rules to mine domain input specifications from logs of access requests.
- Previous work only generated access requests randomly. *ReACP* applies combinatorial testing to systematically generate diverse access requests. *ReACP* also details how system states are systematically set up to facilitate access requests generation and exploration of the system.
- *ReACP* defines content patterns and use regular expression matching to label the result of access requests more accurately. In previous work, permission labelling was only based on HTTP status codes.

Policy abstraction level. In previous work, the inferred AC policies were expressed at a low level of abstraction. In *ReACP* we introduce a *meta-attribute* concept and a method to process intermediate data, to capture important relationships among factors that influence AC policies. This raise the abstraction level of policies extracted, which facilitates manual analysis and detection of AC issues.

[2]A preliminary version of this work appeared as an internal technical report [10].

In addition, only two Web applications were used to evaluate the previous work. In this extension, four web applications were used to evaluate the approach. Our evaluation shows that (1) the new techniques proposed in this extension work achieve better accuracy at policy extraction and, (2) abstracting policies results in a substantial reduction of the number of policies required for manual analysis in comparison with previous work.

We have applied *ReACP* for reverse-engineering AC policies from four Web applications (three open-source and a proprietary one built by our industry partner). We evaluated our approach in terms of the correctness of the inferred policies with respect to the actual AC implementation. We also evaluated the usefulness of our approach by assessing whether it assists the detection of AC issues. The results show that 97.8% of the inferred policies are correct with respect to the actual AC implementation. The analysis of these policies led to the discovery of 64 AC issues that were reported to the developers (and further fixed in the case of the proprietary application).

To summarize, the main contributions of the paper are: 1) *ReACP*, a semi-automated framework for extracting AC policies of Web applications, based on systematic testing and machine learning, which supports enhanced resource discovery, extraction of high-level policies, and AC issues detection; 2) a large-scale empirical evaluation of the correctness of the policies inferred by *ReACP*, conducted on one industrial and three open-source Web applications.

The remainder of this paper is organized as follows. Section 2 provides some background concepts. Section 3 gives an overview of the proposed approach, with further details presented in Section 4. Section 5 reports on the evaluation of our approach. Section 6 discusses related work. Section 7 concludes the paper.

2. Preliminaries

In this section, we first present some background concepts used in the rest of the paper, such as the RBAC model (to which the policies extracted by *ReACP* conform) and decision trees (a machine learning technique leveraged by *ReACP* to infer AC policies). Then, we illustrate the running example that will be used in rest of the paper to explain the proposed approach.

2.1. Access Control

Access control [1] is a security mechanism that determines which users (either human or software-based agents) can access the resources of a system and which actions they can perform on a certain resource; the access is granted or denied based on the AC policies defined for the systems.

The most pervasive access control mechanism in enterprise IT infrastructures is *role-based access control (RBAC)* [11], in which access rights are determined based on the *role* assigned to each user and to the permissions (i.e., the actions that can be performed on resources) associated with each role.

Although several RBAC models have been presented in the last years (see [12] for a recent survey), in this paper we consider AC policies expressed according to the original RBAC96 model proposed in [7], and later on consolidated into a NIST standard [13]. The main concepts of the RBAC96 model are users, roles, sessions, and permissions. A role is assigned to users through a user-role assignment relation; a role-permission assignment relation captures the permissions assigned to each role.

In the context of Web applications, resources are server pages (e.g., PHP files), client pages (e.g., Javascript, HTML, and CSS files), images, and other files. Such resources are identifiable based on URIs (Uniform Resource Identifiers) as well as relevant request parameters and their values. A permission represents the possibility of accessing a resource through an HTTP method (e.g., GET, POST). A Web application can be any application that is deployed on a web server or a cloud system, accessible via HTTP or HTTPS.

2.2. Decision Trees

Classification is a form of supervised learning that is used to predict the category to which elements of an input data set belong, according to patterns found in the data and based on a training data set in which the categorization of elements is already known. The patterns considered in a classification problem are *structural*, i.e., they capture the structural features of elements and represent them in a form that can be

Table 1: Sample email data

Domain	ImageContent	Attachment	Size	Spam
abc1.com	normal	no	normal	no
abc2.com	normal	no	huge	no
abc3.com	high	no	normal	yes
bad.com	no	no	small	yes
abc4.com	normal	yes	normal	no
bad.com	no	no	normal	yes
abc5.com	high	no	small	yes
abc6.com	normal	yes	normal	yes
abc7.com	no	no	normal	no

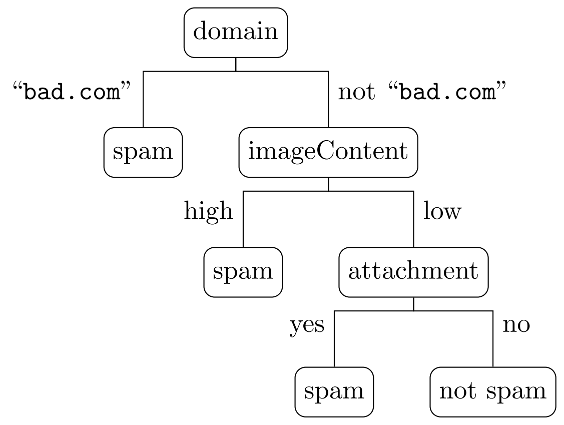


Figure 1: Decision tree learnt from the data in Table 1.

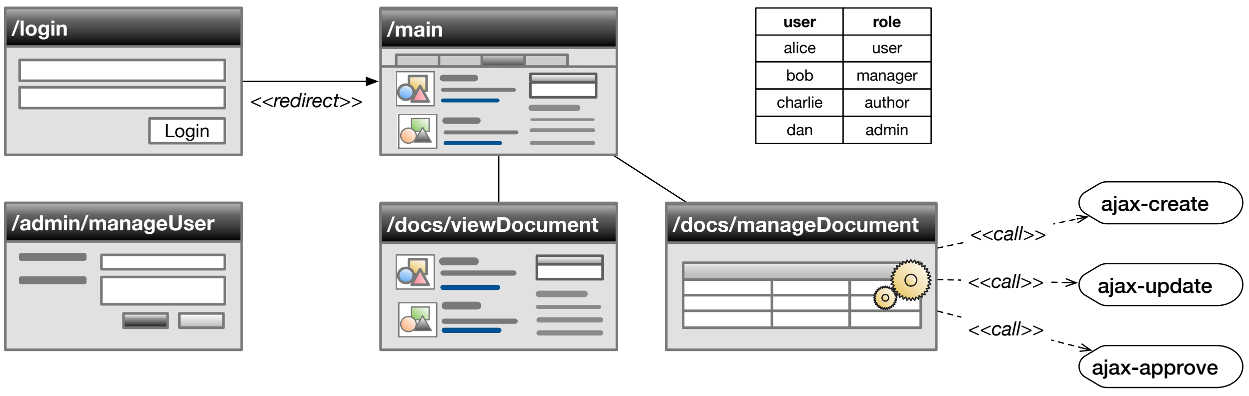
further examined and reasoned on. Examples of classification problems are deciding whether a message is "spam" or "not spam" or whether a bank transaction is "fraudulent" or "legitimate"; examples of structural patterns for these two problems could be, respectively, the internet domain of the mail sender and the recipient's country of the transaction. Algorithms performing classification are called *classifiers*.

Decision tree learning [14] is a classification technique that builds a predictive model in the form of a decision tree. A decision tree is a tree in which the internal nodes represent tests on features of the input data and the leaf nodes represent category labels assigned to the input data. The edges from one internal node to its children are labeled with each of the possible values for the feature denoted by the node itself. Each leaf can also be interpreted as a classification rule: the tests of the nodes on the path from the root to the leaf are the antecedent of the rule, and the category label assigned by the leaf is the consequent. In this paper, we will use the C4.5 classifier [15] to learn decision trees.

To illustrate, consider the raw email data shown in Table 1, in which the first four columns represent structural patterns (features) of emails and the last one corresponds to the category label associated with each email (e.g., obtained through manual marking by users). Figure 1 shows the corresponding decision tree learnt from these data. One of the classification rules generated suggests that if an email comes from the "bad.com" domain, then it is a spam email.

2.3. Running Example

Our running example is an application called *SDM* (*Simple web-based Document Management*), which consists of five server pages: `/login`, `/main`, `/docs/viewDocument`, `/docs/manageDocument`, and



user	role
alice	user
bob	manager
charlie	author
dan	admin

Figure 2: The *Simple web-based Document Management (SDM)* application.

`/admin/manageUser` (see Figure 2). A user accesses the system through the `/login` page; once authenticated, she is redirected to the `/main` page, which displays links to the `/docs/viewDocument` and `/docs/manageDocument` pages, depending on the user's role(s). Page `/docs/viewDocument` and `/docs/manageDocument` accept three parameters: `user` (the user name), `docId` (the unique identifier of the document to view or manage), and `docTitle` (the document title). Page `/docs/manageDocument` also accepts another parameter `action`, which is used to invoke three Ajax-based functions to create, update, and approve documents (`ajax-create`, `ajax-update`, and `ajax-approve`). These functions are enabled based on the role of the user accessing the page and, when triggered, they create dynamic forms through which documents can be uploaded and updated. Page `/admin/manageUser` allows admin users to create, update, delete, and assign roles to users.

We assume to know the list of users (and their login credentials) and the roles defined for the *SDM* application; they are shown in the top-right corner of Figure 2.

3. Overview of the *ReACP* Framework

The *ReACP* framework uses a black-box approach to infer the AC policies[3] from a Web application. It takes as input a set of user credentials[4] and the URLs of the main entry resources (e.g., login pages for regular users and admins) of the Web application; it returns a set of AC policies, where each policy has the form $\langle u, R, d \rangle$, with u representing the URI of a resource, R representing a role, and d the access control decision (allowed or denied).

The approach implemented in the *ReACP* framework consists of the following five steps, illustrated in Figure 3:

1. *Exploratory access logs generation.* We use a Web crawler to dynamically explore the various resources of the application and generate an initial set of access logs.

[3]Hereafter we will use the terms "AC policies" and "AC rules" interchangeably.
[4]As discussed in subsection 2.3, we assume that the user credentials include also the user's role.

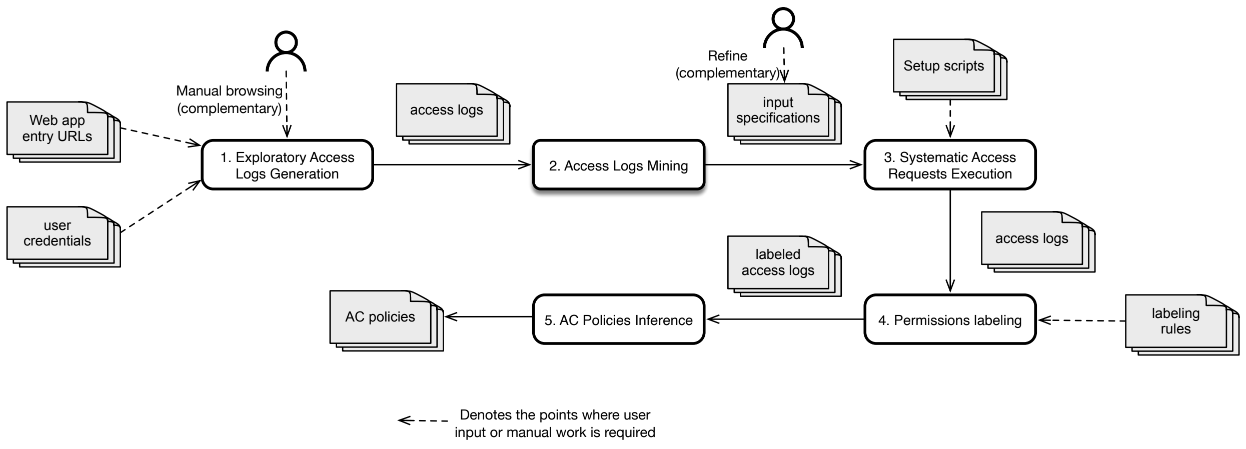


Figure 3: The workflow of the *ReACP* framework.

2. *Access logs mining.* The access logs generated in the previous step are mined for deriving input specifications that can be used to generate new access requests to further exercise the application.

3. *Systematic access requests execution.* The input specifications mined in the previous step are used to systematically generate new access requests for the application; these requests are then executed and logged.

4. *Permission labeling.* The HTTP responses in the access logs obtained in the previous step are labeled according to some predefined rules, to indicate whether an access request for a specific resource was allowed or denied.

5. *AC policies inference.* The labeled access logs obtained in the previous step are processed through a machine learning technique to infer AC policies at a high level of abstraction. The inferred policies are also annotated with confidence levels, which can pinpoint potential AC issues.

The next section presents each of these steps in detail.

4. The ReACP Framework in detail

4.1. Step 1: Exploratory Access Logs Generation

This step takes as input the set of user credentials and the URLs of the main entry resources of the application (e.g., login pages for regular users and admins), and produces a set of access logs recording the HTTP request and response messages, tagged with user credentials (name and role).

The goal of this step is to explore the application through different navigation paths, in order to identify the different resources available in the application and record which user (when logged in with a certain role) can access which resources.

This step is accomplished through the use of a Web crawler (also known as Web spider), which is a tool for the retrieval of Web resources. Starting from the URL of an entry resource, a Web crawler extracts hyperlinks from the contents of the pages and then iteratively navigates the Web resources addressed by those links [16]; advanced Web crawlers can deal with Web forms and support modern Web technologies such as AJAX and JSON. In our framework, the crawler accesses the application using the entry resources. The crawling is iterated over multiple sessions, with one session for each user credentials provided in input. Since crawlers may have difficulties in identifying isolated Web resources (i.e., resources that are not linked from others), we also include in the set of entry resources all static resources (e.g., server pages, images, folders) contained in the Web root of the application. In our implementation of the *ReACP* framework, we use the spider module of *BurpSuite* [17], a commercial security tool suite. We deployed the BurpSuite proxy between client and server to capture the HTTP requests and responses [5] After each exploratory session, the captured messages are tagged with the user's name and role.

Notice that the crawler might not be able to reach pages that can be accessed only by providing input data that comply with the business logic. To overcome this limitation, in this step it is also possible to complement the automated crawling process with manual browsing of the application. During manual browsing, all requests and their corresponding responses are still recorded through the BurpSuite proxy.

4.2. Step 2: Access Logs Mining

This step takes as input the access logs generated in the previous step and mines them, to derive input specifications that can be used to generate additional, meaningful access requests to further exercise the application (as will be discussed in Section 4.3).

The mining process analyzes the HTTP requests and responses in the logs to extract the following information:

- *URLs* used in HTTP requests and in the `url` attributes of hyperlinks returned in HTTP responses. Besides the path and the query string, each URL also includes the protocol, the host, and the port.
- *parameter names* and *values* used in:
  - *URL query strings.* The query string of a URL contains pairs of the form ⟨parameter, value ⟩; for example, in the URL `http://examp.le/p?q=10&r=3`, the query string is the part following the '`?`' symbol and contains two pairs ($\langle q, 10\rangle$ and $\langle r, 3\rangle$). We consider the query strings of all the URL fetched as described in the previous item.
  - *Body of POST requests.* POST requests may contain parameter-value pairs resulting from the submission of a Web form.
  - *Web forms.* Web forms in HTTP responses can contain server-generated or predefined values for input fields (e.g., in `select` elements).
  - *Headers.* Headers in HTTP responses can contain server-generated cookies.

Given an input parameter p, we use the following rules to determine the input specifications:

1) *Source*: if p corresponds to a cookie in the header or to a hidden input field in a Web form, we consider the values of p as server-generated values (i.e., values not provided by the users) and assign to p the special type *SERVER*.

2) *Type*: if all the values of p recorded in the log are of the same type T (e.g., integer, double, string), then we assign to P type T; otherwise we assign to p type *string*.

3) *Enumeration*: if the number of distinct values for p recorded in the log is less than a predefined threshold[6], then the type of p is an enumeration whose elements are the logged values for p.

4) *Length*: if the type of p is *string* and rule 3 is not applicable, then the type of p is annotated with the minimum and maximum string lengths, as determined by the logged values.

5) *Boundary*: if the type of p is *numeric* and rule 3 is not applicable, the logged values of p are clustered using the standard Expectation-Maximization (EM) algorithm [18], which can estimate the number of clusters in addition to providing the clusters themselves. The type of p is then annotated with the identified clusters; each of them is represented by the values of the corresponding logged values.

[5]Since these are standard HTTP messages, they can be captured with any Web proxy tool with similar functionalities.

[6]This threshold is user-configurable and it defaults to 50 in our implementation. In our preliminary experiments, we observed that systematic access requests execution is not scalable when this threshold exceeds 50.

The extracted information is then recorded in an XML file following the Xinput [19] format. One can revise this file to exclude invalid or irrelevant inputs and produce a more accurate input specification based on her/his domain knowledge. Xinput is an XML schema written in the XML Schema Definition (XSD) language[7] that provides a systematic and convenient way of specifying inputs for multiple Web pages. For example, in Xinput an input field of a Web page is defined by its name, source (i.e., intended for the end-user or the Web server), data type (e.g., string or integer), and input classes. The latter can be defined using the well-known restriction specifications of XSD, including *enumeration*, *boundary* (e.g., for both string and numeric classes), and *length* (e.g., for strings).

4.3. Step 3: Systematic Access Requests Execution

This step takes as input the input specifications produced in the previous step and executes the application with new access requests systematically generated from the specifications. The goal of this step is to further exercise the application and determine how it responds to requests containing new data. As a result, it produces more complete access logs than those produced in the initial exploratory step (see Section 4.1). This step consists of two sub-steps: 1) setting up system states; 2) access requests generation and execution.

4.3.1. Setting up System States

System states are defined as server or client states in Web applications that may influence access request permission. Such states need to be properly set up before executing access requests in order to extract AC policies correctly. For example, AC policies for accessing documents created by admin users cannot be extracted if there is no test document created in the file system of the Web application and assigned to an admin user.

Server states are defined in terms of the state of server-side resources, such as database or file system content. These states should be prepared before executing a request and rolled back either after completing a request execution session or when the execution of a request leads to a change in a system resource (e.g., a change in the file system). Server states are set up in the same way as done when applying Web application testing methods such as the one described in reference [20]. We populate the databases and configure the Web application under test with the same settings used by the application when it runs in real settings. To automate this, our framework supports the execution of external scripts for initializing and resetting server-side resources after a request execution session.

Client states are usually stored in cookies and hidden form fields and are used to keep track of the user interaction with the server. For example, in e-commerce systems, the client state keeps track of the user's activities and of his shopping cart. To deal with client states, *ReACP* executes access requests in sessions, with each session dedicated to a user who may or may not be authenticated. In each session, before submitting an access request to a resource available at an URL *url*, *ReACP* checks the input specifications to see if there is an input parameter of type *SERVER* (see Section 4.2). If this is the case, *ReACP* executes an HTTP GET/POST request to *url* with no input parameters to obtain cookies and hidden input values from the HTTP response. These cookies and input values of hidden form fields are then used as value of the input parameters having type *SERVER* when submitting access requests to *url* throughout the session.

Our rationale here is to generate access requests within a valid user session and to observe resulting AC policies implemented by the application. Notice that, in our setup, a valid user session may also refer to that of an anonymous user (i.e., an un-authenticated user), as some applications allow users to access certain resources anonymously and generate the corresponding session identifiers. Our access requests may include manipulated values for inputs of other types, to access arbitrary resources, as explained in Section 4.3.2.

4.3.2. Access Requests Generation and Execution

This step generates values for the input parameters specified in the input specifications obtained in step 2, and uses them to execute new access requests, to explore new access control behaviors.

[7]http://www.w3.org/XML/Schema

Table 2: Input parameters and classes of the page `/docs/manageDocument` in the running example

Input parameter	Input class
`user`	*alice*, *bob*, *charlie*, *dan*
`docId`	*c1*, *c2*, *c3*, *c4*, *c5*, *c6*, *c7*, *c8*, *c9*, *c10*
`docTitle`	*short*, *long*
`action`	*create*, *update*, *approve*

In the following, we illustrate how *ReACP* generates values for input parameters having a type different from *SERVER*. If the type of an input parameter is an enumeration (see rule 3 in subsection 4.2), each of its logged values becomes an input class. When an input parameter is annotated with the minimum and maximum string lengths (see rule 4 in subsection 4.2), the parameter has two input classes: one determined by the minimum length and the other determined by the maximum length. If the type of an input parameter is annotated with clusters (see rule 5 in subsection 4.2), each cluster becomes an input class.

For instance, Table 2 shows four input parameters and their input class specifications characterizing the `/docs/manageDocument` page of our running example application. The parameter `user` is specified with four input classes, *alice*, *bob*, *charlie*, *dan*, which correspond to the four enumerated logged values of `user`. The parameter `docId` is specified with ten input classes, *c1*, *c2*, . . . , *c10*; each class represents a cluster of unique document identifiers that have been logged. The parameter `docTitle` is specified with two input classes, *short* and *long*, which correspond to the minimum string length and the maximum string length, as determined by the logged values. The parameter `action` is specified with three input classes, *create*, *update*, *approve*, which correspond to the three distinct logged values of `action`.

Ideally, all possible combinations of the input classes of the input parameters should be used to generate new access requests and exercise the page so as to observe all possible access control behaviors. However, trying to exercise all combinations would lead to an exponentially large number of requests to handle. To avoid this issue, we apply *combinatorial testing* [21, 22, 23, 24] to generate combinations of input values and achieve an adequate degree of combinatorial coverage while minimizing the number of requests to be executed. More specifically, we apply a pairwise test generation following a greedy strategy, implemented using a tool called AllPairs[8]. This choice stems from existing evidence suggesting that there is little difference between test combinations generated by greedy combination strategies and by other strategies using metaheuristic search algorithms [25]. For the above example, AllPairs generates only 40 pairwise combinations.

To generate the actual requests we instantiate the combinations by deriving concrete values from the input class specifications. The instantiation rules defined for the different types of specifications are explained below using as an example the combination ⟨*alice*, *c1*, *short*, *create*⟩, generated by AllPairs for our running example.

- *Enumeration*: pick the only available value since the input class contains only one element (a logged value). In our example, we pick value `"alice"` from the class *alice* for `user` and value `"create"` from the class *create* for `action`.

- *Length*: generate two strings that satisfy the string length specifications: one is randomly generated and the other is randomly selected from the logged values; then, pick one of the two strings randomly. For instance, the class *short* of `docTitle` in our running example represents strings with length less than or equal to 10. In this case, our approach generates a random string `"abc"` and selects the string `"profile"` from the logged values of `docTitle`; notice that both are within the string length specified by the class *short*. For the sake of illustration, let us assume that our framework randomly picks `"abc"` as the value for `docTitle`.

- *Boundary*: generate two numeric values that satisfy the boundary specifications: one is a randomly generated and the other is randomly selected from the logged values; then, pick one of the two values

[8]The tool, developed by A. G. McDowell, was previously available at `http://www.mcdowella.demon.co.uk/allPairs.html`.

Table 3: Regular expressions used to classify response content

Content pattern	**Content type**
`.*(not authorized|not allowed|insufficient privilege| not have permission|access denied).*`	*denial*
`.*(error|exception|problem|failed).*`	*error*
`.*∼(not authorized|not allowed|insufficient privilege|error| not have permission|access denied|exception|problem|failed).*`	*normal*

randomly. Notice that the random numeric value generated is either of type integer or of type double depending on the data type of the input parameter. For instance, the *c1* class of `docId` in our running example requires integer values between 1 and 10. In this case, our framework generates a random number with value 10 and selects the number 5 from the logged values of `docId`; both are within the range specified by the class *c1*. For the sake of illustration, let us assume that our framework randomly picks 5 as the value for `docId`.

By applying the above rules to our running example, we can send an HTTP request to exercise the page `/docs/manageDocument` with four input parameters: `user` = `"alice"`, `docId` = 5, `docTitle` = `"abc"`, and `action` = `"create"`.

To execute the access requests, our framework performs, for each user, the necessary authentication using the provided user credentials (see Section 4.1). It then sets up the proper system states (see section 4.3.1) and executes the generated access requests. The access logs containing the request/response messages are then generated with the BurpSuite proxy like in step 1 (see Section 4.1).

4.4. Step 4: Permissions Labeling

This step takes as input the access logs generated in the previous step and a set of labeling rules. It analyzes each response in the access logs with respect to the labeling rules and determines whether the application grants or denies access to the requested resource. It yields access logs labeled with permissions; the labels are *allowed*, *denied*, and *unknown* (when the permission cannot be identified).

To determine access permissions, our approach mainly uses the standard HTTP status codes [26]. However, some Web applications gracefully deny unauthorized accesses by returning an "OK" status (HTTP response code = 200) and adding some text (e.g., "You are not authorized to access this page") to the response body as HTML content. Since this HTML content usually follows certain patterns, we use regular expressions (see Table 3) to detect whether the response body indicates a *normal* response, an access *denial*, or an *error*.

We include in *ReACP* a predefined set of *generic* labeling rules; they are shown in Table 4. Since different applications may implement different behaviors for allowing or denying access requests, users may need to refine these rules in an application-specific way so as to correctly label permissions. For example, as shown in Table 4, the permission label for a request resulting in the HTTP response code 204 is generally defined as "unknown" regardless of the response content; however, for a specific application, the user may refine this result as "denied" or "allowed". In our implementation, users may provide application-specific labeling rules in a configuration file; they take priority over the generic labeling rules.

To continue our running example, Table 5 shows the labeled access requests corresponding to the instantiation of the pairwise combinations of input classes in Table 2; the rightmost column contains the access permissions.

4.5. Step 5: AC Policies Inference

This step takes as input the access logs labeled with permissions generated in the previous step and applies machine learning to infer AC policies that are implemented in the application. This step consists of two sub-steps: 1) pre-processing data and 2) learning AC policies.

Table 4: Labeling rules for determining access permissions

Response code	Response content	Permission
200	*denial*	denied
200	*error*	unknown
200	*normal*	allowed
201	any	allowed
202	any	unknown
203	*denial*	denied
203	*error*	unknown
203	*normal*	allowed
204	any	unknown
205	any	allowed
206	any	unknown
3xx	any	unknown
400	any	unknown
401	any	denied
403	any	denied
404	any	unknown
405	any	denied
406	any	unknown
407	any	denied
408	any	unknown
409	any	denied
41x	any	unknown
500	*denial*	denied
5xx	*normal*	unknown

4.5.1. Pre-processing Data

Since the goal of this work is to learn AC policies for validating implemented policies and detecting AC issues, it is desirable that the learned AC policies explicitly capture the key factors that impact access control, abstracting away unimportant details. For instance, in our running example, a user assigned to the role `user` is only allowed to access his own documents. If we have long lists of document identifiers and users, we want to avoid learning specific AC policies for every document, as this would unnecessarily complicate manual analysis. Instead, we would like to raise the abstraction level of the inferred policies: in the example, it means that we would like to learn, at an abstract level, how document ownership influences AC policies. To achieve this goal, we pre-process data before learning the actual policies, by retaining attributes or their relationships that really matter in AC policies. The pre-processing step relies on the concept of *meta-attribute*, which represents a relationship between two or more attributes. First, for each row of the labeled access log, we generate a meta-attribute called *role*, which associates the user with her role as determined from the list of user credentials provided as input to *ReACP*. In addition, we support the definition of application-specific meta-attributes, such as *ownership*, *authorship*, and *assignment*. The values of such meta-attributes can often be mapped to a Boolean value, indicating whether the underlying relationship exists. Application-specific meta-attributes can be defined in *ReACP* as rules following the template below:

```
IF att1 IN set1 [AND | OR] att2 IN set2
   THEN meta-att = value-t ELSE meta-att = value-f
```

The above rule sets — for each row in the labeled access log — the value of the meta-attribute `meta-att` either to value `value-t` or to value `value-f` depending on the values of attributes `att1` and `att2`, which

Table 5: Labeled access logs for the page `/docs/manageDocument`

#	`user`	`docId`	`docTitle`	`action`	Permission
1	`"alice"`	5	`"abc"`	`"create"`	*denied*
2	`"alice"`	11	`"sales results"`	`"create"`	*denied*
			[. . .]		
10	`"alice"`	105	`"def"`	`"create"`	*denied*
11	`"bob"`	7	`"customer data"`	`"update"`	*allowed*
12	`"bob"`	14	`"profile"`	`"update"`	*allowed*
			[. . .]		
20	`"bob"`	111	`"ghi"`	`"create"`	*denied*
21	`"charlie"`	3	`"new customer data"`	`"create"`	*allowed*
			[. . .]		
40	`"dan"`	107	`"abcdefghijkl"`	`"approve"`	*unknown*

Table 6: Pre-processed access logs for the page `/docs/manageDocument`

#	`user`	`docId`	`docTitle`	`action`	`role`	`isOwned`	Permission
1	`"alice"`	5	`"abc"`	`"create"`	`user`	0	*denied*
2	`"alice"`	11	`"sales results"`	`"create"`	`user`	0	*denied*
				[. . .]			
10	`"alice"`	105	`"def"`	`"create"`	`user`	0	*denied*
11	`"bob"`	7	`"customer data"`	`"update"`	`manager`	1	*allowed*
12	`"bob"`	14	`"profile"`	`"update"`	`manager`	1	*allowed*
				[. . .]			
20	`"bob"`	111	`"ghi"`	`"create"`	`manager`	0	*denied*
21	`"charlie"`	3	`"new customer data"`	`"create"`	`author`	1	*allowed*
				[. . .]			
40	`"dan"`	107	`"abcdefghijkl"`	`"approve"`	`admin`	0	*unknown*

are checked with respect to the values contained in sets `set1` and `set2`, respectively; notice the possibility of using a conjunction or a disjunction for the conditions on attributes `att1` and `att2`.

For example, let us consider the access log data shown in Table 5. We could choose to capture document ownership with a meta-attribute called `isOwned` by defining the following rules:

- `IF user IN {alice} AND docId IN {1, 2} THEN isOwned = 1 ELSE isOwned = 0`
- `IF user IN {bob} AND docId IN {7, 14} THEN isOwned = 1 ELSE isOwned = 0`
- `IF user IN {charlie} AND docId IN {3, 5} THEN isOwned = 1 ELSE isOwned = 0`
- `IF user IN {dan} AND docId IN {11} THEN isOwned = 1 ELSE isOwned = 0`

The application of these rules to the access log in Table 5 results in the pre-processed access log shown in Table 6; columns `role` and `isOwned` correspond to the meta-attributes generated in this pre-processing step.

4.5.2. Learning AC Policies

To learn AC policies from access data, we consider the permissions of access requests as class labels (i.e., *allowed*, *denied* or *unknown*) and propose to apply a classification technique [14] to automatically identify the attributes (e.g., role) that characterize AC policies. Among various classification techniques, we use *decision trees* (see Section 2.2) because the output of the classification has the form of a tree, from which

```
role = admin: allowed (9,1,90%)
|
role != admin
| isOwned = 1
| | action = update: allowed (6,0,100%)
| | action != update
| | | role = manager: allowed (2,0,100%)
| | | role != manager: denied (4,0,100%)
| isOwned != 1
| | action = create
| | | role = user: denied (3,0,100%)
| | | role != user: allowed (5,0,100%)
| | action != create: denied (10,0,100%)
```

Figure 4: An example of decision tree learned from the access logs for the `/docs/manageDocument` page.

one can extract IF-THEN rules, which can then further be analyzed by security engineers; in our case, a classification rule represents an AC policy. More specifically, to learn decision trees we use the J48 classifier (included in Weka [27]) that implements (an extension of) the C4.5 algorithm [15].

In our context, we use decision trees to characterize and explain the labeled access data (training data) in the most accurate way possible; we remark that, differently from mainstream applications, we do not use decision trees for prediction. Since our goal is to capture as many AC policies as possible, we are not concerned with over-fitting the training data. Therefore, we turn off the pruning option of J48 [14] to generate the most precise tree possible.

In the decision tree, each leaf node is annotated with a *prediction confidence*, which indicates the homogeneity of the data sample that falls into a leaf node; in our case it is expressed in terms of the ratio of access requests that are allowed, denied, or unknown. In this respect, we say that a policy is *consistent* if it predicts with 100% confidence the permission (allowed, denied, or unknown) given in the corresponding leaf node. *Inconsistent* policies are those for which in their corresponding leaf nodes some access requests are allowed while others are denied or unknown, or vice versa. Our tool highlights such inconsistent rules in its output since they may indicate potential issues in the AC implementation. For example, Figure 4 depicts the decision tree learned from the access logs for the page `/docs/manageDocument` in Table 6. The classifier identified three attributes that are relevant for access control: `role`, `isOwned`, and `action`. There are seven leaf nodes and they are annotated with the number of correctly classified instances, the number of incorrectly classified instances, and the prediction confidence. The first leaf node correctly classified nine instances and incorrectly classified one instance, hence its prediction confidence is 90%. The incorrectly classified instance is due to the instance labeled with the *unknown* permission in the access logs (last row in Table 6). The other leaf nodes correctly classified the remaining instances, resulting in a 100% prediction confidence. In total, seven AC policies regarding the access to the documents under the page `/docs/manageDocument` can be easily derived from the tree in Figure 4; such policies are easy to analyze and are shown in Figure 5. For instance, policy $P2$ reads as follows: "users that have not been assigned the `admin` may only perform the action `update` on the documents that they own"; other policies read in a similar way. Notice that the policy `P1` is annotated with an `"inconsistent"` label as it has less than 100% prediction confidence.

4.6. Guidelines for Analyzing Inferred AC Policies

The inferred AC policies should be checked by analysts to identify discrepancies with expected AC requirements and possibly detect AC vulnerabilities such as *privilege escalation*, *information disclosure*, and *missing access control* (see [28]). Toward this end, since the number of inferred policies could be large, below we provide some guidelines and heuristics for analysts, to prioritize the analysis by paying special attention to the following policy types:

```
P1: IF role = admin THEN allowed @inconsistent

P2: IF role != admin AND isOwned = 1 AND action = update THEN allowed

P3: IF role != admin AND isOwned = 1 AND action != update AND role = manager THEN allowed

P4: IF role != admin AND isOwned = 1 AND action != update AND role != manager THEN denied

P5: IF role != admin AND isOwned != 1 AND action = create AND role = user THEN denied

P6: IF role != admin AND isOwned != 1 AND action = create AND role != user THEN allowed

P7: IF role != admin AND isOwned != 1 AND action != create THEN denied
```

Figure 5: AC Policies derived from the decision tree in Figure 4.

- *Inconsistent AC policies* (i.e., AC policies with less than 100% prediction confidence). Analysts should check whether the inconsistency is due to AC implementation errors or other reasons that affect access control.
- *AC policies labeled with "unknown" permissions.* These policies may reflect AC implementation errors.
- *AC policies allowing all users or low-privileged users (e.g., guest users) access to sensitive resources.* We have observed many cases where sensitive resources have been left unprotected by the implemented AC mechanism. It is up to the domain experts to define resource sensitiveness.
- *AC policies related to database, configuration, installation, backup files, and other static documents.* Such documents might have been left accidentally by developers before deployment and access to them could be unprotected. Hence, they could be the target of unauthorized direct access.

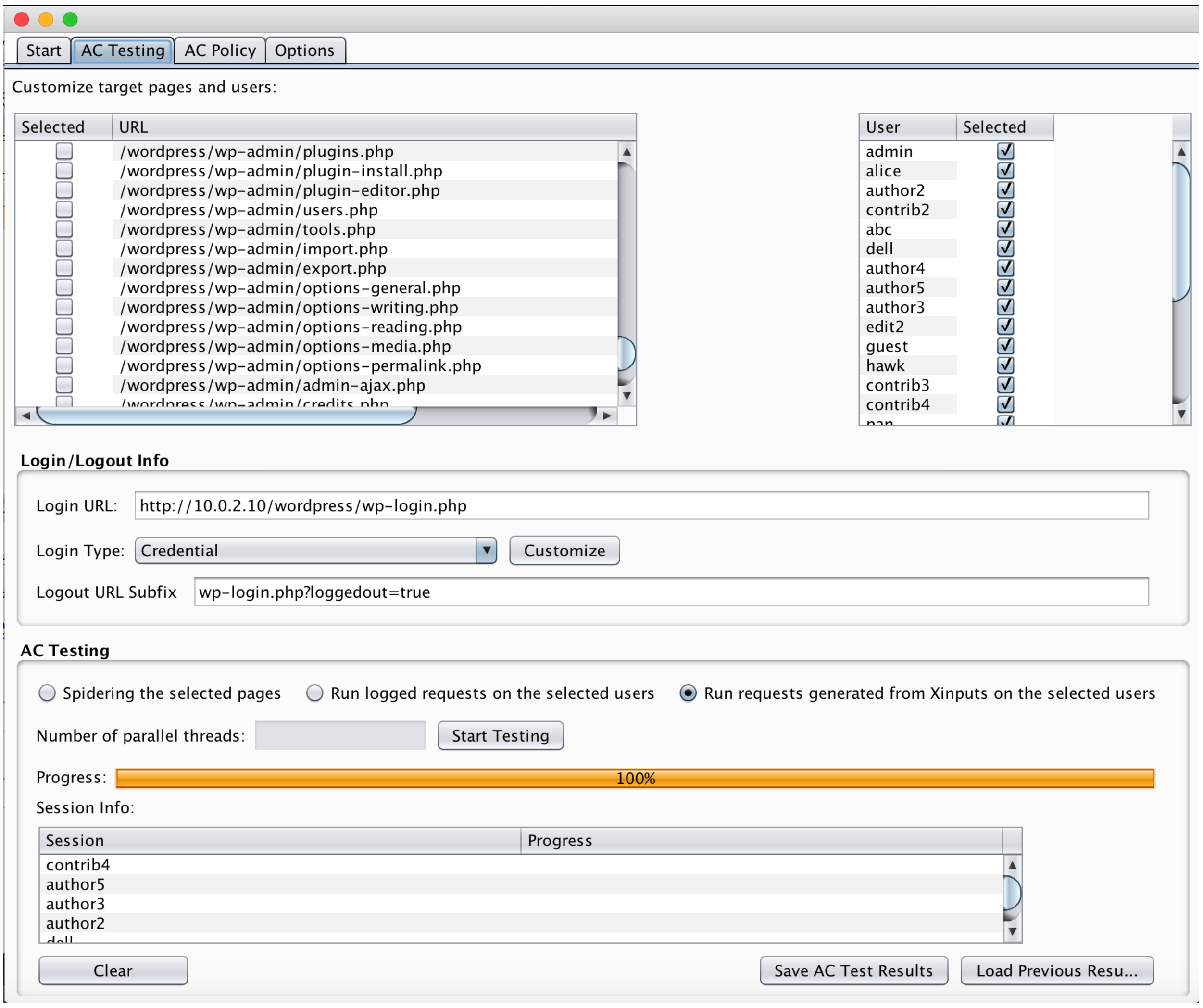
In principle, these guidelines can be followed in any order, since different applications may exhibit different characteristics of AC issues. However, in our experiments (Section 5), we followed the order as listed above and found it to be effective for quickly identifying AC issues. Moreover, we remark that the first two cases ("inconsistent" permissions and "unknown" cases) are highlighted by our tool, enabling their fast identification.

For instance, in our running example, policy `P1` is labeled as *inconsistent*. Following the guidelines above, an analyst would find out that this inconsistency is due to an access instance labeled with the *unknown* permission in the access logs (see Table 5); this could be possibly caused by some inconsistencies in the AC implementation.

4.7. Implementation

We have implemented the *ReACP* approach on top of the GUI tool ACMate [29], previously developed by some of the authors [9]. Figure 6 shows a screenshot of the tool; it is integrated with BurpSuite [17] to leverage its features, including its proxy, spider, and site map.

The tool provides users with a graphical user interface to specify the input configuration of the application (i.e., a list of *username* and *password* for the users of the application under testing) and a set of permission filters for classifying access responses. It supports the extraction of domain input specifications from proxy logs; the extracted specifications are saved as Xinput files and can further checked and refined by users. Furthermore, the tool provides users with options for customizing the test execution (e.g., running test cases in sessions) and setting the states of target servers. The tool includes the decision tree algorithm J48 [27]; its output consists of the decision trees yielded by the classifier and of the textual representation (in the form of IF-THEN rules) of the inferred AC policies.


Start | AC Testing | AC Policy | Options

Customize target pages and users:

Selected	URL
☐	/wordpress/wp-admin/plugins.php
☐	/wordpress/wp-admin/plugin-install.php
☐	/wordpress/wp-admin/plugin-editor.php
☐	/wordpress/wp-admin/users.php
☐	/wordpress/wp-admin/tools.php
☐	/wordpress/wp-admin/import.php
☐	/wordpress/wp-admin/export.php
☐	/wordpress/wp-admin/options-general.php
☐	/wordpress/wp-admin/options-writing.php
☐	/wordpress/wp-admin/options-reading.php
☐	/wordpress/wp-admin/options-media.php
☐	/wordpress/wp-admin/options-permalink.php
☐	/wordpress/wp-admin/admin-ajax.php

User	Selected
admin	✓
alice	✓
author2	✓
contrib2	✓
abc	✓
dell	✓
author4	✓
author5	✓
author3	✓
edit2	✓
guest	✓
hawk	✓
contrib3	✓
contrib4	✓

Login/Logout Info

Login URL: http://10.0.2.10/wordpress/wp-login.php

Login Type: Credential | Customize

Logout URL Subfix: wp-login.php?loggedout=true

AC Testing

Spidering the selected pages | Run logged requests on the selected users | [✓ Run requests generated from Xinputs on the selected users]

Number of parallel threads: | Start Testing

Progress: 100%

Session Info:

Session	Progress
contrib4	
author5	
author3	
author2	

Clear | Save AC Test Results | Load Previous Resu...

Figure 6: Screenshot of the tool implementing the *ReACP* approach

5. Experimental Evaluation

In this section, we report on the evaluation of our proposed approach, focusing on the correctness of the inferred AC policies and on the usefulness of the latter for detecting AC issues; we investigate the three research questions stated below.

ReACP aims to infer the policies that have been actually implemented in Web applications. Therefore, we need to validate the inferred policies to assess whether they correctly reflect the implementation; the first research question we investigate is the following:

RQ1: *Does ReACP correctly infer the AC policies implemented in Web applications?*

In *ReACP*, we propose to use application-specific (app-specific) labeling rules whenever feasible and prioritize them over generic labeling rules so that *ReACP* can infer AC policies more correctly. Hence, in the next research question we investigate the role of applying app-specific labeling rules on top of generic labeling rules:

Table 7: Subject applications used in the evaluation

Application	**Type**	**Language**	**#Server Pages**	**LOCs**	**#Roles**
iTrust	open source	JSP	220	23859	8
TaskFreak	open source	PHP	180	19448	4
WordPress	open source	PHP	482	133023	6
SECP	proprietary	AJAX/Java	n/a	100K+	25

RQ2: *Does applying app-specific labeling rules help* ReACP *learn AC policies more correctly than when applying only generic labeling rules?*

We check the inferred policies for inconsistency and validate them against stakeholders' expected/intended AC requirements to identify discrepancies. These are indicators of AC issues, including vulnerabilities that lead to privilege escalations or faulty enforcement implementations that do not satisfy the requirements (e.g., denying accesses to entitled users). Therefore, the last research question assesses whether the policies inferred with *ReACP* help pinpoint AC issues:

RQ3: *Do the inferred AC policies help detect AC issues?*

In the following sections, we first describe the selected subject applications and the experimental procedure; then, we present and discuss the experimental results for the three research questions.

5.1. Subject Applications and Settings

We evaluated ReACP on three open-source Web applications (iTrust[9], TaskFreak[10], and WordPress[11]), and one proprietary application (hereafter denoted as SECP) developed by our industrial partner. iTrust is an electronic health records system developed created as a course project at North Carolina State University and used to teach software engineering [30]; TaskFreak is a popular open-source system for project management; WordPress is a widely used content management system (CMS) for blogging and news publishing; SECP[12] is a distributed information sharing platform for emergency services and crisis management. While all the applications support the concepts of role and role-permission assignment, the implementation of AC enforcement is different among applications. For instance, WordPress and SECP allow permission administration at run time while TaskFreak and iTrust use hard-code role permissions; WordPress and TaskFreak take into account ownership relationship between users and resources, whereas iTrust does not.

Table 7 shows the statistics of the subject applications in terms of source code availability, programming language, size — expressed both in terms of number of server pages and in terms of number of lines of code (LOCs)[13], and number of predefined, hard-coded roles. The heterogeneity of the applications in terms of size, language, and AC enforcement mechanisms make them interesting subject applications for our evaluation.

In terms of experimental settings, WordPress and TaskFreak were deployed on an Apache HTTP Server whereas iTrust was deployed on Apache Tomcat running on a Linux server; SECP was deployed at the partner's premises and accessed remotely through a dedicated VPN connection. In addition to the predefined, hard-coded roles, in some applications (e.g., WordPress) we added a special role to represent anonymous accesses and check whether anonymous users could carry out any unauthorized operations. Furthermore, to ensure the systematic discovery of AC behaviors, we set up system states (both server and client states)

[9]`https://sourceforge.net/projects/itrust/`

[10]`http://www.taskfreak.com` (version 0.6.4)

[11]`https://wordpress.org`

[12]The actual name of the application has been omitted for confidentiality reasons.

[13]The number of LOCs for the open-source subject applications was retrieved using *cloc* (`https://github.com/AlDanial/cloc`), a popular tool for measuring code metrics; this measurement also includes the library files. We could not measure the size of the SECP application but were told that it has over 100 KLOCs, excluding library files.

as discussed in subsection 4.3.1. All the Web applications were setup in a real Web environment. During the initial stage of the experiments, the first author spent a few hours to use the open-source applications logged in as various users (Table 8). We then collected the data from databases, which were then used to create system states. For the SECP application, we obtained the setup data from our industrial partner as they were readily available.

As discussed in Section 4, users' roles and resource ownership are important factors in access control enforcement. Therefore, in our experiments, we prepared data in a way that ensures the coverage of user-role assignments and resource ownership/assignment settings. Moreover, for each application under test, we created a set of users and assigned them to the available roles. We also created users with "anonymous" role, to further exercise our approach with invalid user sessions[14]. We then populated data for each user (e.g., by creating new posts and setting up the ownership to the posts for WordPress, and by creating new missions for SECP). Furthermore, to avoid interferences among test sessions (e.g., to avoid that after testing the access with one user, resource states could be changed and affect the access permissions of other users), we saved and restored the states of the system before and after each test session[15].

5.2. Procedure

We performed our experiments using the ACMate tool, following the five steps of *ReACP* (see Section 4). For each subject under test, for step 1 (Exploratory Access Logs Generation), ACMate created a valid session in the browser by authenticating on the application website with the given credentials of a test user[16]. ACMate then crawled the Web application under the same user session. We collected data about the accesses to the application resources by enabling the proxy option in the browser and setting it to direct the requests/responses to BurpSuite's proxy. We complemented the crawling session with manual browsing of the application, to simulate different access situations for the links missed by the crawler. This was done for each test user in the given set of user credentials we created (see Section 5.1) as well as for the "anonymous" users; we spent about 15 minutes of manual browsing for each test user.

Afterwards, we used ACMate's automated mining feature (step 2 - Access Logs Mining) to analyze the resulting output and generate an Xinput file for input specification. We manually checked the input specifications for the validity of the data classes and corrected anomalous entities. For example, when mining the logs of `addTelemedicineData.jsp`, a web page of iTrust, ACMate yielded an input specification of type enumeration for the `diastolicBloodPressure` attribute, with two concrete values, 25 and 60. The value 25 is invalid and was randomly entered by the crawler; therefore, we updated the attribute value class to match the valid range $[40, 150]$. We used the revised input specification to generate and execute the access requests for step 3 (Systematic Access Requests Execution).

Notice that we refine input specifications so that ACMate has a good baseline for systematic test generation. As our test input generation process involves *randomization* (see Section 4.3.2), the access requests generated by ACMate may still contain invalid inputs. In this case, the Web application may throw error messages or encounter exceptions. This is intended to capture whether the Web application correctly implements the AC policies regarding such invalid access requests. This is done in step 4 and step 5 below.

As part of step 4 (Permissions Labeling), for correctly classifying access permissions, we defined *app-specific labeling rules* for each application, on top of the generic labeling rules discussed in Section 4.4. More specifically, we defined 17 rules for iTrust, 56 rules for TaskFreak, 20 rules for WordPress, and 24 rules for SECP. Our app-specific labeling rules match responses having HTTP response codes 301, 40x or 50x with specific content patterns (e.g., "Server Error", "Request timeout"), responses having HTTP code 200 with messages like "You are not authorized", and responses with error messages like "ITrustException"

[14]Notice that, if the application correctly implements the AC policy that an anonymous user cannot access a document of a registered user, the access requests within the session of an anonymous user that attempts to access arbitrary resources may end up in invalid sessions.

[15]In the case of the open-source applications, we used the system image restoration feature provided by the virtualization environment on which the applications were deployed; in the case of the SECP application, we reset and restored the application database using the remote login.

[16]ACMate left the username and password input fields blank during the authentication of "anonymous" users

```
1  <!-- Confirmation question before executing a requested command. -->
2  <!-- If a user is asked this question, he is able to perform the command -->
3  <!-- Define as allowed and override generic rule: 403 - any content - denied -->
4  <Filter permission="allowed">
5      <StatusCodePattern matched="true">403</StatusCodePattern>
6      <ContentPattern matched="true">error-page</ContentPattern>
7      <ContentPattern matched="true">Are you sure you want to do this?</ContentPattern>
8  </Filter>
9
10 <!-- read private pages is denied -->
11 <!-- 404: Not Found while the page is private; reading a private page is denied -->
12 <!-- Define as denied and override generic rule: 404 - any content - unknown -->
13 <Filter permission="denied">
14     <StatusCodePattern matched="true">404</StatusCodePattern>
15     <ContentPattern matched="true">It looks like nothing was found at this location. </
           ContentPattern>
16 </Filter>
```

Figure 7: An excerpt of app-specific permission labeling rules for WordPress.

Table 8: Total number of access requests generated by *ReACP* (column *Requests*) and time (in hours) taken to analyze each test subject (column *Total*), further presented in terms of time taken to execute access requests (column *Execution*), create user credentials and setup scripts (column *Setup*), manually browse the application (column *Browsing*), refine the input specification inferred by *ReACP* (column *Refining*), and derive app-specific labeling rules (column *Labeling*); column *Tot. manual* indicates the total time taken by the manual activities (i.e., the sum of columns *Setup, Browsing, Refining*, and *Labeling*)

Application	Requests	Total	Execution	Setup	Browsing	Refining	Labeling	Tot. Manual
iTrust	92733	71.8	51.3	8	2	6.5	4	20.5
TaskFreak	50250	42.03	12.03	8	6	2	14	30
WordPress	213252	45.4	11.4	8	3	18	5	34
SECP	19969	528.2	499.2	3	11	9	6	29

and "NullPointerException". An excerpt of app-specific labeling rules defined for WordPress is shown in Figure 7. The first app-specific rule (lines 1–8) states that if the server returns a response page with code 403, containing the messages "error-page" and "Are you sure you want to do this", the access request is to be labeled as *allowed*. This rule, which overrides the generic labeling rules for response code 403, takes into account the app-specific behavior for which the server may ask for a confirmation before letting the user proceed. Similarly, the second rule (lines 10–16) states that when the server returns a response page with code 404, containing the message "It looks like nothing was found at this location", the access request is to be labeled as *denied*. This rule deals with the case in which a user requests to read a private post owned by another user.

Finally, as part of step 5, we inferred the AC policies. In this step, for resources where relationships among attributes can help characterize AC policies, we pre-processed data as described in subsection 4.5.1, by defining meta-attributes representing the relationships.

Table 8 shows the total number of access requests generated by *ReACP* (column *Requests*) and the time (in hours) taken to analyze each test subject (column *Total*), further presented in terms of time taken to execute access requests (column *Execution*), create user credentials and setup scripts (column *Setup*), manually browse the application (column *Browsing*), refine the input specification inferred by *ReACP* (column *Refining*), and derive app-specific labeling rules (column *Labeling*); column *Tot. manual* indicates the total time taken by the manual activities (i.e., the sum of columns *Setup, Browsing, Refining*, and *Labeling*). In terms of execution time, SECP took a long time (499.2 hours) because it was deployed on our industrial partner's remote server and each access request took a while to execute. The rest of the applications were deployed locally in our environment, resulting in faster access requests execution.

Table 9: Number of correct policies inferred using generic and app-specific labeling rules

Application	*Resources*	*Generic*			*App-specific*		
		Inferred Policies	*Correct Policies*	*%Correct*	*Inferred Policies*	*Correct Policies*	*%Correct*
iTrust	162	722	625	86.6	693	688	99.3
TaskFreak	21	584	434	74.3	1547	1544	99.7
WordPress	28	237	52	21.9	278	201	72.3
SECP	61	4003	1585	39.6	4113	4054	98.6
Total	272	5546	2696	48.6	6631	6487	97.8

5.3. Correctness of Inferred AC Policies (RQ1 and RQ2)

To answer RQ1 and RQ2, we verified the correctness of the inferred policies by checking them against the actual implementation; in the case of RQ1, we used the policies inferred using generic labeling rules whereas for RQ2 we used the policies inferred using app-specific labeling rules.

Table 9 shows the evaluation results obtained for the four applications when using generic (column *Generic*) and app-specific (column *App-Specific*) labeling rules. Column *Resources* denotes the number of resources tested by ACMate; column *Inferred Policies* indicates the number of inferred policies; columns *Correct Policies* and *%Correct* report the number and the percentage of policies that conform to the actual AC implementation, respectively.

The answer to RQ1 is that our approach, when using generic labeling rules, on average correctly infers 48.6% of the policies. We analyzed the incorrect policies and noticed that they were mainly caused by incomplete permission labeling rules; this is somehow expected because generic labeling rules are not specific to a given application.

The answer to RQ2 is that our approach, when using app-specific labeling rules, on average correctly infers 97.8% of the policies. The results show that app-specific labeling rules are highly important for inferring AC policies correctly, since they eliminate most of the problems found when using only generic labeling rules. However, the trade-off for using app-specific rules is the cost for defining them, since the process is manual. In our case, one of the authors inspected the results obtained by using generic labeling rules and defined app-specific rules; on average, this process took 28 hours per application. In practice, domain experts are expected to define them; based on our experience with our industrial partner, we expect the process to be much faster when leveraging domain knowledge.

Another issue when using app-specific rules is that they may not be refined enough. Currently, our labeling rules rely only on response codes and on the content of the response body (e.g., the presence of a specific string indicating an access permission). They fail to recognize the denial cases in which one must take into account the relationship between the requesting user and the target resource. For example, in TaskFreak, when an adversarial user requests profile information of other users, the application returns (with HTTP code 200) a page with the profile of the requesting user instead of replying with a denial message; in such a case, the analysis of the response content is not enough to classify the request correctly. We observed this issue with all four applications and identified it as one of the causes of learning incorrect policies.

We also noticed that incorrectly inferred policies were due to the incomplete or incorrect states initialization during test execution sessions. As discussed in subsection 4.3.2, server and client states may affect access permissions and incomplete or incorrect states may lead to denial of accesses. For example, WordPress uses a dynamic parameter called *nonce* to control the order of AC requests; all requests without the parameter *nonce* are denied. We observed that, for some of the requests we executed, their states were not completely set, leading to learning incorrect policies; this was the main reason for correctly learning only 72.3% of the policies for WordPress, even when using app-specific rules.

In previous work [9][17] some of the authors analyzed the same iTrust application, extracting 1518 (low-level) AC policies, out of which 1441 (94.92%) were correctly inferred. In comparison, in this work we extracted 693 (high-level) AC policies, out of which 688 (99.3%) were correctly inferred. These results show that the new approach for policy extraction proposed in this work and implemented in the *ReACP* framework achieves better accuracy. Moreover, extracting policies at an abstract level results in a much less number of policies, which facilitates manual analysis.

5.4. Detecting AC Issues Using the Inferred Policies (RQ3)

To answer RQ3, we compared the inferred policies with the application *Gold Standard (GS) policies*, to detect possible discrepancies and analyze their causes in terms of AC issues (i.e., AC vulnerabilities). GS policies are policies that reflect the intended access rights of security engineers. We derived the GS policies from available requirement documents (for SECP and WordPress) and from the AC configuration files (for iTrust and TaskFreak); in the case of SECP, we had several discussions with its security engineers to complement the available requirement documents and better understand their intended access rights.

[17]The same paper also presented an analysis of SECP; however, the results are not comparable with those presented in this work as the version of SECP considered in this paper, with respect to the one used in reference [9], has undergone major changes in terms of access control policies.

Table 10: AC issues detected for the potentially problematic policies

	Generic					*App-specific*				
	Policies	*PrivEsc*	*InfDis*	*MisAC*	*Min*	*Policies*	*PrivEsc*	*InfDis*	*MisAC*	*Min*
iTrust										
Inc	6					2				
Unk	96					14		8		
Prv	20			17		34			29	
Dev	0					3				
Oth	0					0				
Total	122			17		53		8	29	
TaskFreak										
Inc	0					8	3			
Unk	148					1		1		
Prv	19	1		5		94	5		5	
Dev	0					1				
Oth	4				1	4				1
Total	171	1		5	1	108	8	1	5	1
WordPress										
Inc	1				1	1				1
Unk	122					77				
Prv	0					0				
Dev	0					0				
Oth	0					5				2
Total	123				1	83				3
SECP										
Inc	0					0				
Unk	2335		2			55		2		
Prv	2	1				567	3			
Dev	0					0				
Oth	54				2	102				4
Total	2391	1	2		2	724	3	2		4
Overall Total	2807	2	2	22	4	968	11	11	34	8

Following the guidelines proposed in Section 4.6 for the analysis of the inferred AC policies, we partitioned the potentially problematic policies into five categories: inconsistent policies (*Inc*), policies labeled with "unknown" permissions (*Unk*), policies allowing low-privileged users access to sensitive resources (*Prv*), policies related to static documents (*Dev*), and policies that do not belong to any of the previous categories and that led to the detection of AC issues (*Oth*). We considered four types of AC issues: privilege escalation (*PrivEsc*), information disclosure (*InfDis*), missing access control (*MisAC*), and minor AC issues (i.e., defects) such as unclear or ambiguous responses to access requests (*Min*).

Table 10 shows the number of potentially problematic policies (column *Policies*) and the type and number of AC issues found for the four applications when using generic (column *Generic*) and app-specific (column *App-Specific*) labeling rules. The table also shows that using app-specific labeling rules results in the detection of more AC issues compared to using generic labeling rules. More specifically, we found the AC issues detected using generic rules to be a subset of those detected using app-specific rules; therefore, in the following we only discuss the AC issues detected using app-specific rules.

In total, we found 64 issues, distributed as follows: 37 vulnerabilities (8 *InfDis*, 29 *MisAC*) in iTrust, 14 vulnerabilities (8 *PrivEsc*, 1 *InfDis*, 5 *MisAC*) and 1 defect in TaskFreak, 0 vulnerabilities and 3 defects in WordPress, 5 vulnerabilities (3 *PrivEsc*, 2 *InfDis*) and 4 defects in SECP. To the best of our knowledge, the vulnerabilities in TaskFreak had not been published before; we informed the developers and submitted the vulnerabilities to the NVD database (US National Vulnerability Database)[18]. As for the vulnerabilities in SECP, we informed our industrial partner, who quickly fixed them. These results can be used to answer RQ3: our approach can help detect AC issues, by analyzing the inferred policies and investigating those with discrepancies from intended access rights.

In the remaining of this section, we describe in detail the AC issues detected in each open-source application; we are not allowed to include further details about SECP because of a confidentiality agreement.

iTrust. In our analysis, we found 29 resources located in important directories (`util`, `hcp-patient`, `errors` and `DataTables`), which are not protected by the implemented AC policies (*MisAC* vulnerability); these resources can leak important information, such as patient data, database tables or transactional logs. Some examples of this vulnerability are:

- the `/util/resetPassword.jsp` page allows any user (even an anonymous user without authentication) to change the password of any other user;
- the `/util/getUser.jsp` page is accessible without proper authorization enforcement and displays users' private data;
- the `/errors/reboot.jsp` page allows anyone to reboot the web server, which might render the system inaccessible to all users.

We also found eight resources affected by information disclosure issues: when sending certain AC requests with invalid parameter values to the resources located in `/auth/getPersonnelID.jsp`, `/auth/hcp`, and `/auth/patient`, the application returns error messages disclosing snippets of source code and database information.

TaskFreak. In our experiments, we found five resources located in `/xajax`, `/jscalendar`, and `public.php`, which are not protected by the implemented AC policies (*MisAC* vulnerability). We also found eight resources affected by privilege escalation vulnerabilities: a malicious user can perform unauthorized operations on resources of other users. Some examples of this vulnerability are:

- one of the inferred AC policies for `project_edit.php` states that any user can open the "create new project" page by requesting `project_edit.php?id=0` and creating a new project; according to the GS policy for internal and guest users, this is incorrect.
- one of the inferred AC policies for `user_edit.php` states that a user can change the password of other users provided that the two password attributes `password1` and `password2` have the same value. This policy, however, is incorrect according to the GS policy, which states that only `administrator` users can edit the account details of other users.

We also found an information disclosure issue in `rss.php`: when executing an AC request, it returns an MySQL exception message that discloses the names of some database tables and columns. Lastly, we found an AC defect in `project_edit.php`, in which the response page does not clearly state whether the requests for certain project edit operations coming from low-privilege users are allowed or denied.

[18]https://nvd.nist.gov

WordPress. We found three resources in the `wp-admin` directory affected by AC defects; we discuss these defects below.

- for `user-edit.php`, we detected an inconsistent AC policy. In the GS, it states that only `administrators` can access this resource. However, among the 800 access requests generated as non-administrators by ACMate, 80 of them were allowed to access the resource. Those 80 requests represent high-privileged users such as `editor`. It is not clear to us whether this is due to outdated GS or to an AC implementation error.
- for `admin.php`, we found that the responses to certain access requests are not clear or rather misleading regarding the permission. More specifically, when ACMate sent requests as an `administrator` user to control an administration-related plugin, the server responded with "no access is permitted"; however, according to GS, administrator should be able to access it. With further analysis, we found out the plugin was not actually installed: the AC implementation should clearly state that the plugin does not exist.
- for `edit-comments.php`, when ACMate sent access requests as `subscriber` users, the server responded with HTTP code 403 (Forbidden Access) but with a message requesting to confirm the operation: "Are you sure you want to do this?". Similar to the above, this is rather misleading regarding the permission.

5.5. Limitation and Discussion

The results presented above show the correctness of our approach in terms of the inferred policies and its usefulness for detecting AC issues. However, our approach and its evaluation are affected by the following limitations.

Semi-automated approach. Our approach is a semi-automated approach. One has to manually refine domain input specifications and define application-specific labeling rules, to correctly and effectively infer the implemented AC policies. These manual tasks may be error-prone and require domain knowledge. Absence or improper definition of these inputs may lead to missing or incorrect policies. To mitigate this problem, in our tool we bundle an Xinput schema for specifying domain inputs systematically and provide generic labeling rules as starters. From the experience with our industrial partner, we observed that domain experts were able to provide the required input to our tool based on the bundled artifacts. For the open-source applications, the first author performed these manual steps; we observed that although the first author was not a domain expert, he was able to perform the tasks adequately after using the applications for a few days.

The inferred AC policies are presented to the security engineer, for validation with respect to the intended access rights and for detecting AC issues. This also requires manual work, although this is inherent for the context we considered. To mitigate this problem, we abstract access data using meta-attributes and express the policies using decision trees, which are easy to read and comprehend [14]. As shown in the example in Figure 4, decision trees express the role-based AC policies well. We also provide *prediction confidence* annotations in the leaf nodes of decision trees to highlight possible inconsistent policies as well as guidelines for analyzing the policies. During our experiments, we had several meetings with the security engineer of the industrial application we evaluated. He was able to validate the extracted policies in a matter of minutes. This suggests that the abstracted policies are indeed intuitive and easy to analyse.

Static analysis. Our approach does not perform static analysis of the application source code. Rather, it relies on black-box testing so that it can work on proprietary applications for which the source code is not available. Our test cases may not discover access control policies implemented through sophisticated business logic in the code, which instead static analyses may discover. To mitigate this problem, our tool provides users with a graphical user interface to specify domain inputs so that effective test cases can be generated. Although this requires manual work, as stated above, we observed that it is practical.

Test generation. Currently we use a pairwise test generation algorithm for generating access requests. Other test generation strategies, such as fuzz testing, mutation testing, and search-based software testing [31], could be used. Investigating their effect on our approach is left for future work.

Machine learning. In our approach, for learning AC policies, we chose the decision tree algorithm because of its comprehensibility. Nevertheless, there are many other classification algorithms that may offer better capabilities in terms of data processing, classification, and visualization. In future work, we plan to explore other algorithms.

Practical implications. The applicability of our approach is mainly impacted by the cost of the required manual tasks. However, as shown in column *Tot. manual* of Table 8, conducting the manual tasks (i.e., the total of columns *Setup, Browsing, Refining* and *Labeling*) ranges from 20.5 hours (for iTrust) to 34 hours (for WordPress), with an average of $\frac{20.5+30+34+29}{4} = 28.38$ hours. We deem this a reasonable cost for effectively and correctly inferring AC policies, when there is no AC policy specification available.

5.6. Threats to Validity

Our evaluation may be affected by the following threats.

Coverage of implemented AC policies. We did not investigate the completeness of our approach in extracting all the implemented policies because, to do so, we would have had to establish, by manually inspecting the source code, a ground truth in terms of all the policies implemented in each application. This task would have been highly labor-intensive and quite difficult, and impossible to perform for the closed-source SECP application developed by our industrial partner. To mitigate this threat, we sampled some source code of open source applications and checked that all the implemented policies in the sampled code were extracted by ACMate. We were not allowed to inspect the source code of SECP but we had several meetings with the developer and the security engineer of SECP throughout the experiments conducted over 3 months and validated the inferred AC policies together. During the final technology transfer meeting, the developer confirmed that, to the best of his domain knowledge, the coverage of implemented AC policies was complete.

Correctness of inferred AC policies. In Section 5.3, we evaluated the correctness of AC policies inferred by our tool against the actual implementation. To establish the ground truth, i.e., the actual implementation, one of the authors manually exercised the application and checked its access control behavior. The threat is that the author might have misjudged some of the behaviors, especially the complex ones. For example, in our industry partner's application, after running certain operations, the role of a user may change dynamically and hence its access rights. Since there are several roles and resources in the application, it was often challenging to determine the correctness. To mitigate this threat, throughout the experiments, the first three authors conducted weekly meetings and verified the correctness of the inferred AC policies together. We re-ran the test cases and also inspected the source code of the three open-source applications to verify the complex cases. As stated above, we were not allowed to inspect the source code of SECP but we verified the correctness of inferred AC policies over the meetings with the security engineer.

Determining AC issues. In Section 5.4, we reported AC issues that were determined based on the guidelines we proposed in Section 4.6. The threat here is that those issues may be false positives. To mitigate this threat, we confirmed the issues against the known vulnerabilities, listed in the NVD database, of the open source applications we used in our evaluation. Regarding the issues we detected in our industry partner's application, we discussed and confirmed all the cases with its developers.

RBAC. The scope of our approach is limited to the RBAC model. We did not consider other access control models such as attribute-based access control, task-based access control, etc. Therefore, our results will not generalize to extracting access control polices implemented in those other models. On the other hand, even if AC policies are not implemented based on RBAC model, our tool will still extract AC policies in Web applications (following the RBAC model) which may be useful for security testing purposes.

6. Related Work

Our approach is related to work done in the areas of (i) inference of AC policies (from software artifacts) using black-box analysis, (ii) inference of AC policies using code analysis, and (iii) detection of AC defects and vulnerabilities.

Inference of AC policies using black-box analysis

Martin and Xie [8] proposed an approach to detect discrepancies between a policy specification and its implementation. In this work, the policy specification is used to generate access requests (consisting of subjects, resources, and actions). The dataset containing pairs of access requests and their associated responses from the system under test undergoes a rule inference algorithm called Prism [32] implemented in Weka [27] to infer policy properties. Discrepancies between such properties and the policy specification indicate issues. *ReACP* does not require to have a policy specification a priori; it automatically explores the system under test, infers input specifications, and uses them together with different user credentials to systematically generate access requests.

Gauthier et al. [33] proposed an approach to reverse-engineer RBAC policies in SecureUML [34] models, from XACML [35] implementations of a system. The resulting SecureUML models can then be analyzed using the SPARQL [36] query language. *ReACP* is more generic, since it can extract AC policies hard-coded in Web applications and does not require an XACML implementation. Furthermore, while Gauthier et al. [33]'s approach is based on static analysis, *ReACP* dynamically exercises the application under test to capture additional AC behaviors (which a static approach might miss).

Molloy et al. [37] proposed an approach that mines RBAC policies from existing logs, which reflect access traces of users recorded in the past. Medvet et al. [38], Xu and Stoller [39, 40], Cotrini et al. [41], and Iyer and Masoumzadeh [42] proposed various approaches to mine attribute-based AC (ABAC) policies from existing logs. All these approaches rely only on existing logs; since the latter may only contain a fraction of possible access requests to resources, such approaches could miss many AC policies. For example, if logs contain only traces of legitimate access requests, policies regarding negative authorizations (i.e., access requests that are denied) could be missed. By contrast, our approach does not rely on existing logs; instead, it automatically explores a given Web application to discover its resources and systematically generates various (both legitimate and illegitimate) access requests to those resources for inferring the implemented policies.

Several approaches [39, 42, 43] have proposed to mine AC policies from existing AC implementations, often as part of the migration to a modern access control paradigm; for example, Bui et al. [44, 43] proposed an approach to mine relationship-based AC (ReBAC) policies—an object-oriented extension of ABAC policies—from existing access control lists. Slankas and Williams [45] and Xiao et al. [46] proposed approaches to extract AC policies from requirement documents written in natural language, using NLP (natural language processing) techniques to extract AC concepts like subjects, actions, and resources. The major difference with the above-mentioned approaches is that *ReACP* extracts AC policies from a Web application, treating the latter as a black-box.

Inference of AC policies using code analysis

Alalfi et al. [47] proposed an approach for reverse engineering RBAC models of Web applications. The approach infers UML-based structural models and behavioral models using source code transformation and instrumentation techniques. The inferred models act as inputs for another model transformation technique to construct RBAC models that can be used to check against security properties [48]. The major difference with *ReACP* is that the latter does not require to instrument the application source code. Instead, *ReACP* uses a proxy to capture access traces and performs logs analysis and systematic test generation to infer AC policies. In addition, *ReACP* takes into account several types of server resources, including static files (e.g., PDF documents, images), which are not considered in Alalfi et al.'s approach.

Detection of AC defects and vulnerabilities

Srivastava et al. [49] proposed a static analysis-based approach for detecting AC vulnerabilities regarding the use of APIs in Java applications. RoleCast [50] and FixMeUp [51] applies static analysis to find sensitive program operations, in PHP Web applications, with missing access control checks. Different from our approach, these approaches are not designed to recover implemented AC policies. They only infer operations with missing AC checks. They require code analysis and work on Java or PHP applications only. FixMeUp automatically fixes the defects by inserting code that performs access control checks, but this requires user to specify access control policies.

Scambray et al. [52] proposed a differential analysis approach to detect authorization vulnerabilities in Web applications. The approach involves crawling a system under test using several authenticated and unauthenticated users' sessions to determine which portions of the system are accessible from which users. *ReACP* also involves web crawling with a set of user credentials but, unlike Scambray et al.'s approach, *ReACP* accounts for Javascript and "unlinked" areas of Web applications. Moreover, the goal of *ReACP* is different as we aim at recovering AC policies for Web applications. We consider different resource abstractions and apply machine learning to infer AC policies with a granularity at the role level, not at the user level. As a result, *ReACP* yields high-level policies that can be analyzed for detecting issues and reused for maintenance purposes.

Noseevich and Petukhov [53] extended differential analysis with the role concept and the notion of use cases, representing roles' actions and their dependencies; the latter are inputs that should be provided by a human operator. The approach considers sequences of use cases, iterates through these sequences, and applies differential analysis for each user case to detect insufficient access control. *ReACP* differs in a number of aspects. First, our goal is to infer AC policies that, on the one hand, can be inspected to detect AC issues and, on the other hand, can be used for other purposes such as regression testing or software maintenance. Second, we deal with Javascript and unlinked resources, which are not supported in [53].

Several approaches [54, 55, 2, 3, 4, 56, 5, 57, 58, 59, 60, 61] generate access control test cases from AC policy specifications to detect AC defects in the implementations; policy specifications are defined using XACML [54, 2, 3, 59] or declarative access control rules [4, 57]. In general, these approaches are model-based and generate test models or abstract test cases that cover various (combinations of) AC rules embedded in policy specifications. The scope of these approaches also vary: generating executable test cases (e.g., [57], prioritizing test case execution based on a given test budget constraints (e.g., [54]), localizing faults (e.g., [58]), generating test oracles (e.g., [60, 61]); other approaches [62, 59] focus on proposing coverage criteria, such as AC rule coverage, rule pair coverage, and modified condition/decision coverage, to measure and ensure the quality of (AC) test suites. Daoudagh et al. [6] conducted a systematic literature review on 20 studies that focus on testing of the usage and access control systems inside DevOps processes. The survey reported that many critical AC issues arise from ambiguous or faulty AC specifications. In contexts where policy specifications are faulty, ambiguous, or unavailable (either because they are missing or hard-coded in the implementation), any AC testing approach requires a reverse-engineering approach like ours, which focuses on extracting and abstracting AC policies from the implementation. Hence, our approach is complementary to the above-mentioned approaches.

7. Conclusion

Broken access control (AC) is a widely recognized security issue in Web applications. As Web applications and their AC mechanisms become increasingly complex, AC vulnerabilities become significant threats. Testing and validation to detect AC problems are thus crucial but usually rely on documented and regularly updated AC specifications. In practice, however, it is common for such specifications to be missing or outdated. For many systems, AC policies are even hard-coded in the implementation without proper documentation at all.

In this paper, we have proposed *ReACP*, a semi-automated framework to infer AC policies of Web applications from their implementation. *ReACP* rests on the integration of a popular security tool suite, domain input specifications, combinatorial testing, and a machine learning algorithm.

We have evaluated *ReACP* on four Web applications (three open-source and a proprietary one built by our industry partner). The results show that 97.8% of the inferred policies are correct with respect to the actual AC implementation; the analysis of these policies led to the discovery of 64 AC issues that were reported to the developers. These results suggest that *ReACP* can correctly infer many policies, which can in turn help detect actual AC issues.

As part of future work, we plan to extend *ReACP* to support more expressive AC models, such as GemRBAC [12] (and its contextual extension [63]) and ABAC [64].

References

[1] R. S. Sandhu, P. Samarati, Access control: Principles and practice, IEEE Communications Magazine 32 (1994) 40–48.

[2] E. Martin, T. Xie, A fault model and mutation testing of access control policies, in: Proceedings of the 16th international conference on World Wide Web (WWW'07), ACM, New York, NY, USA, 2007, pp. 667–676.

[3] J. Hwang, E. Martin, T. Xie, V. C. Hu, Testing access control policies, Encyclopedia of Software Engineering 1 (2010) 673–683.

[4] Y. L. Traon, T. Mouelhi, B. Baudry, Testing security policies: going beyond functional testing, in: Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE'07), IEEE, IEEE, 2007, pp. 93–102.

[5] D. Xu, L. Thomas, M. Kent, T. Mouelhi, Y. Le Traon, A model-based approach to automated testing of access control policies, in: Proceedings of the 17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, ACM, New York, NY, USA, 2012, pp. 209–218.

[6] S. Daoudagh, F. Lonetti, E. Marchetti, Continuous development and testing of access and usage control: A systematic literature review, in: Proceedings of the 2020 European Symposium on Software Engineering, ESSE 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 51–59. URL: `https://doi-org.libproxy.smu.edu.sg/10.1145/3393822.3432330`. doi:`10.1145/3393822.3432330`.

[7] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman, Role-based access control models, IEEE Computer 29 (1996) 38–47.

[8] E. Martin, T. Xie, Inferring access-control policy properties via machine learning, in: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06), IEEE, IEEE, 2006, pp. 4–pp. –238.

[9] H. T. Le, C. D. Nguyen, L. C. Briand, B. Hourte, Automated inference of access control policies for web applications, in: Proceedings of the 20th ACM Symposium on Access control Models and Technologies (SACMAT'15), ACM, ACM New York, NY, USA, 2015, pp. 27–37.

[10] H. T. Le, D. C. Nguyen, L. Briand, ReACP: A Semi-Automated Framework for Reverse-engineering and Testing of Access Control Policies of Web Applications, Technical Report, University of Luxembourg, 2016.

[11] D. F. Ferraiolo, D. R. Kuhn, R. Chandramouli, Role-Based Access Control, Computer Security Series, 2nd edition ed., Artech House, 685 Canton Street, Norwood, MA 02062, 2007.

[12] A. Ben Fadhel, D. Bianculli, L. Briand, A comprehensive modeling framework for role-based access control policies, Journal of Systems and Software 107 (2015) 110–126. URL: `http://www.sciencedirect.com/science/article/pii/S0164121215001041`.

[13] R. Sandhu, D. Ferraiolo, R. Kuhn, The NIST model for role-based access control: towards a unified standard, in: Proceedings of the Fifth ACM Workshop on Role-based Access Control (RBAC 2000), ACM, ACM New York, NY, USA, 2000, pp. 47–63.

[14] I. H. Witten, E. Frank, Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, 2005.

[15] J. R. Quinlan, C4.5: Programs for Machine Learning, volume 1, Morgan Kaufmann Publisher, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, 1993.

[16] C. Olston, M. Najork, Web crawling, Foundations and Trends in Information Retrieval 4 (2010) 175–246.

[17] BurpSuite, Web vulnerability scanner, `https://portswigger.net/burp`, 2018.

[18] A. P. Dempster, N. M. Laird, D. B. Rubin, Maximum likelihood from incomplete data via the em algorithm, Journal of the Royal Statistical Society. Series B (Methodological) 39 (1977) 1–38.

[19] C. D. Nguyen, A. Marchetto, P. Tonella, A language for the specification of domain inputs, Technical Report, Fondazione Bruno Kessler, 2011. `http://se.fbk.eu/sites/se.fbk.eu/files/tr-fbk-se-2011-8.pdf`.

[20] S. Elbaum, G. Rothermel, S. Karre, M. Fisher II, Leveraging user-session data to support web application testing, IEEE Transactions on Software Engineering 31 (2005) 187–202.

[21] P. Flores, Y. Cheon, Pwisegen: Generating test cases for pairwise testing using genetic algorithms, in: Proceedings of IEEE International Conference on Computer Science and Automation Engineering (CSAE'11), volume 2, IEEE, IEEE, 2011, pp. 747–752.

[22] J. Yan, J. Zhang, Backtracking algorithms and search heuristics to generate test suites for combinatorial testing, in: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC '06), volume 1, IEEE Computer Society, IEEE, 2006, pp. 385–394.

[23] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, Y. L. Traon, Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines, IEEE Transactions on Software Engineering 40 (2014) 650–670.

[24] Y. Jia, M. B. Cohen, M. Harman, J. Petke, Learning combinatorial interaction test generation strategies using hyperheuristic search, in: Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), volume 1, IEEE, IEEE, 2015, pp. 540–550.

[25] J. Petke, M. B. Cohen, M. Harman, S. Yoo, Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection, IEEE Transactions on Software Engineering 41 (2015) 901–924.

[26] W3C, Hypertext transfer protocol - http/1.1, RFC, 1999.

[27] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The weka data mining software: An update, ACM SIGKDD Explorations Newsletter 11 (2009) 10–18.

[28] OWASP, OWASP 10 Most Critical Web Application Security Risks, Technical Report, The OWASP Foundation, 2013.

[29] H. T. Le, ACMate - a tool for Reverse-engineering and Testing of Access Control Policies of Web Applications, `https://github.com/lehathanh/acmate`, 2017.

[30] S. Heckman, K. T. Stolee, C. Parnin, 10+ years of teaching software engineering with itrust: The good, the bad, and the ugly, in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET '18, ACM, New York, NY, USA, 2018, pp. 1–4.

[31] Y.-F. Li, P. K. Das, D. L. Dowe, Two decades of web application testing—a survey of recent advances, Information Systems 43 (2014) 20–54.

[32] J. Cendrowska, Prism: An algorithm for inducing modular rules, International Journal of Man-Machine Studies 27 (1987) 349–370.

[33] F. Gauthier, E. Merlo, E. Stroulia, D. Turner, Supporting maintenance and evolution of access control models in web applications, in: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME'14), IEEE, IEEE, 2004, pp. 506–510.

[34] T. Lodderstedt, D. A. Basin, J. Doser, Secureuml: A uml-based modeling language for model-driven security, in: Proceedings of the 5th International Conference on The Unified Modeling Language, UML '02, Springer-Verlag, London, UK, UK, 2002, pp. 426–441.

[35] A. Anderson, XACML profile for role based access control (RBAC), 2004. OASIS Access Control TC committee draft.

[36] E. Prud'hommeaux, A. Seaborne, SPARQL query language for RDF, 2008. URL: `http://www.w3.org/TR/rdf-sparql-query/`.

[37] I. Molloy, Y. Park, S. Chari, Generative models for access control policies: applications to role mining over logs with attribution, in: Proceedings of the 17th ACM symposium on Access Control Models and Technologies, ACM, 2012, pp. 45–56.

[38] E. Medvet, A. Bartoli, B. Carminati, E. Ferrari, Evolutionary inference of attribute-based access control policies, in: International Conference on Evolutionary Multi-Criterion Optimization, Springer, 2015, pp. 351–365.

[39] Z. Xu, S. D. Stoller, Mining attribute-based access control policies, IEEE Transactions on Dependable and Secure Computing 12 (2015) 533–545.

[40] Z. Xu, S. D. Stoller, Mining attribute-based access control policies from logs, in: IFIP Annual Conference on Data and Applications Security and Privacy, Springer, 2014, pp. 276–291.

[41] C. Cotrini, T. Weghorn, D. Basin, Mining ABAC rules from sparse logs, in: IEEE European Symposium on Security and Privacy, 2018, pp. 31–46.

[42] P. Iyer, A. Masoumzadeh, Mining positive and negative attribute-based access control policy rules, in: Proceedings of the 23Nd ACM on Symposium on Access Control Models and Technologies, SACMAT '18, ACM, 2018, pp. 161–172.

[43] T. Bui, S. D. Stoller, J. Li, Greedy and evolutionary algorithms for mining relationship-based access control policies, Computers & Security 80 (2019) 317 – 333.

[44] T. Bui, S. D. Stoller, J. Li, Mining relationship-based access control policies, in: Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies, SACMAT 2017, Indianapolis, IN, USA, June 21-23, 2017, 2017, pp. 239–246.

[45] J. Slankas, L. Williams, Access control policy extraction from unconstrained natural language text, in: International Conference on Social Computing (SocialCom '13), IEEE, IEEE, 2013, pp. 435–440.

[46] X. Xiao, A. Paradkar, S. Thummalapenta, T. Xie, Automated extraction of security policies from natural-language software documents, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12), ACM, ACM, ACM New York, NY, USA, 2012, p. 11.

[47] M. H. Alalfi, J. R. Cordy, T. R. Dean, Automated reverse engineering of uml sequence diagrams for dynamic web applications, in: Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW '09), IEEE, IEEE, 2009, pp. 287–294.

[48] M. H. Alalfi, J. R. Cordy, T. R. Dean, Recovering role-based access control security models from dynamic web applications, in: M. Brambilla, T. Tokuda, R. Tolksdorf (Eds.), Proceedings of the International Conference on Web Engineering (ICWE'12), volume 7387 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, Springer Berlin Heidelberg, 2012, pp. 121–136.

[49] V. Srivastava, M. D. Bond, K. S. McKinley, V. Shmatikov, A security policy oracle: Detecting security holes using multiple api implementations, ACM SIGPLAN Notices 46 (2011) 343–354.

[50] S. Son, K. S. McKinley, V. Shmatikov, Rolecast: finding missing security checks when you do not know what checks are, in: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, 2011, pp. 1069–1084.

[51] S. Son, K. S. McKinley, V. Shmatikov, Fix me up: Repairing access-control bugs in web applications., in: NDSS, 2013.

[52] J. Scambray, V. Liu, C. Sima, Hacking Exposed Web Applications: Web Application Security Secrets and Solutions, 3rd edition ed., McGraw-Hill, Berkeley, CA, USA, 2011.

[53] G. Noseevich, A. Petukhov, Detecting insufficient access control in web applications, in: Proceedings of the First SysSec Workshop (SysSec '11), SysSec2011, IEEE Computer Society, Washington, DC, USA, 2011, pp. 11–18.

[54] A. Bertolino, S. Daoudagh, D. El Kateb, C. Henard, Y. Le Traon, F. Lonetti, E. Marchetti, T. Mouelhi, M. Papadakis, Similarity testing for access control, Information and Software Technology 58 (2015) 355–372.

[55] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, Testing access control policies against intended access rights, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, ACM, 2016, pp. 1641–1647.

[56] A. Pretschner, T. Mouelhi, Y. Le Traon, Model-based tests for access control policies, in: Software Testing, Verification, and Validation, 2008 1st International Conference on, IEEE, 2008, pp. 338–347.

[57] D. Xu, M. Kent, L. Thomas, T. Mouelhi, Y. L. Traon, Automated model-based testing of role-based access control using

predicate/transition nets, IEEE Transactions on Computers 64 (2015) 2490–2505.
[58] D. Xu, Z. Wang, S. Peng, N. Shen, Automated fault localization of xacml policies, in: Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, 2016, pp. 137–147.
[59] D. Xu, R. Shrestha, N. Shen, Automated coverage-based testing of XACML policies, in: Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, SACMAT 2018, Indianapolis, IN, USA, June 13-15, 2018, 2018, pp. 3–14.
[60] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, An automated model-based test oracle for access control systems, in: 2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST), IEEE, 2018, pp. 2–8.
[61] S. Daoudagh, F. Lonetti, E. Marchetti, Xacmet: Xacml testing & modeling, Software Quality Journal 28 (2020) 249–282.
[62] E. Martin, T. Xie, T. Yu, Defining and measuring policy coverage in testing access control policies, in: Information and Communications Security, Springer, Springer Berlin Heidelberg, 2006, pp. 139–158.
[63] A. Ben Fadhel, D. Bianculli, L. Briand, B. Hourte, A model-driven approach to representing and checking RBAC contextual policies, in: Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY 2016), New Orleans, LA, USA, ACM, 2016, pp. 243–253.
[64] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, Guide to Attribute Based Access Control (ABAC) Definition and Considerations, Technical Report 800-162, National Institure of Standards and Technology (NIST), 2014.