securityandtrust.lu

ReACP: A Semi-Automated Framework for Reverse-engineering and Testing of Access Control

Ha-Thanh Le, SnT, University of Luxembourg
Cu Duy Nguyen, SnT, University of Luxembourg
Lionel C. Briand, SnT, University of Luxembourg

15 June 2016

978-2-87971-034-1

Abstract

This technical report details our a semi-automated framework for the reverse-engineering and testing of access control (AC) policies for web-based applications. In practice, AC specifications are often missing or poorly documented, leading to AC vulnerabilities. Our goal is to learn and recover AC policies from implementation, and assess them to find AC issues. Built on top of a suite of security tools, our framework automatically explores a system under test, mines domain input specifications from access request logs, and then, generates and executes more access requests using combinatorial test generation. We apply machine learning on the obtained data to characterise relevant attributes that influence access control to learn policies. Finally, the inferred policies are used for detecting AC issues, being vulnerabilities or implementation errors. We have evaluated our framework on three open-source applications with respect to correctness and completeness. The results are very promising in terms of the quality of inferred policies, more than 94% of them are correct with respect to implemented AC mechanisms. The remaining incorrect policies are mainly due to our unrefined permission classification. Moreover, a careful analysis of these policies has revealed 92 vulnerabilities, many of them are new.

1 Introduction

Web-based systems are increasingly adopted in virtually all domains, from e-government, social networks and media, to finance and education. Access control (AC) is a pivotal feature of such systems since they serve different users and manage sensitive data. Unfortunately, AC specifications are often missing or poorly documented, leading to AC vulnerabilities. Moreover, the cost to perform AC testing or auditing can be substantially high due to the lack of such specifications.

In this work, we address the problem by developing a semi-automated framework, namely *ReACP*, for the reverse-engineering of AC policies of web-based systems. Furthermore, upon a tool-assisted systematic assessment of output policies, our framework aids the analyst in detecting AC vulnerabilities.

The primitives elements of AC policies are *resources*, *subjects* (*users*), *access contexts*, and *permissions*. A specific policy governs access permissions to a specific resource for one or more subjects in given contexts. Based on these basic concepts, many AC models have been proposed in the past decades [5], such as Role-Based Access Control (RBAC) model [26], XACML [21], or Attribute Based Access Control (ABAC) [16]. In this paper, we refer to the ABAC model and use key attributes, such as *role*, *resource*, and *context*, to present the policies that are reverse-engineered by our framework.

In the reverse-engineering of AC policies, the outcome AC policies need to meet two qualities: *completeness* and *usefulness*. Completeness concerns the coverage of the elements of AC policies to ensure that we do not leave out any important factor that influences AC. In contrast, *usefulness* entails several properties: (i) the obtained AC policies reflect correctly the actual implementation, (ii) the policies must be easy for the analyst to assess, and (iii) they should help pinpoint AC vulnerabilities if they do exist.

Our preliminary work [18] has paved the first building blocks for the framework, in particular: *resource discovery*, *access analysis*, *AC policy inference* aiming at reverse-engineering AC policy specifications. Based on a security tool suite, our approach automatically exercises a target system with a set of known users and roles. Once assess logs are obtained and analysed, we apply a machine learning technique to infer AC policies, and then support the analyst in assessing them and detecting vulnerabilities.

In this paper, we extend our previous work along the two above-mentioned quality dimensions. First, we enhance the resource discovery capability of *ReACP* by considering meaningful domain data. In web-based systems, many web resources are reachable only when meaningful data are provided in access requests. Otherwise, web servers will deny the requests or return no new useful information for identifying new resources or their attributes.

To help achieve this first objective, we propose an XML schema-like format for the specifications of domain inputs. Our format allows the domain experts to easily specify data types (e.g. string, numeric) and other input specifications using boundaries, enumerations, and regular expressions. Moreover, because the specification format is similar to the XML schema, which is widely used in the industry, the learning cost is expected to be low. More importantly, our *ReACP* framework is capable of mining domain input specifications from execution logs automatically. Hence, manual effort is mainly needed for refinements.

Having domain input specifications, our extended framework uses a combinatorial test generation technique to generate many more and diverse access requests. This helps not only uncover new resources but also learn resource and context attributes that matter in AC policies. Moreover, exercising a resource with an increased number of diverse requests also increases the confidence in the policies that we learn.

To improve the usefulness of the inferred policies, we introduce a *meta-attribute* concept and a method to process intermediate data so that the output policies are more understandable. They are often more abstract, yet at the same time can capture the important relationships among factors that influence AC policies.

Key contributions in this paper include:

◇ A semi-automated approach to reverse-engineer access control policies based on systematic testing and machine learning using data in access logs.
◇ A large scale empirical study based on open source web applications demonstrating the degree of correctness and completeness of the learned policies.
◇ An approach using machine learning classifiers to detect potential access control vulnerabilities.

We have evaluated *ReACP* on three open-source web applications, two of them being widely used in practice. Results are promising and show that *ReACP* could infer correct policies and most of the ones implemented (89%). Moreover, it also helped detecting 92 real vulnerabilities.

The remainder of this paper is organised as follows. Section 2 and Section 2.3 give background knowledge and discuss a motivating example that is used along the description of our framework. Section 4 introduces the key concepts used in our work and presents XInput, our proposed format for the specifications of domain inputs. Section 5 discusses our framework, followed by a section that describes our tool. Section 6 reports on our experiments and results. Finally, Section 8 concludes the paper.

2 Background

2.1 Access Control

Access control is a pervasive security mechanism which is used in virtually all systems [9]. It is concerned with authorising the right resource access permissions to users. The fundamental concepts in an access control model include *users*, *subjects*, *objects*, and *permissions*. *User* refers to human users who interact with a computer system. *Subject* refers to a process or program acting on behalf of a user. *Object* refers to resources accessible from a system (in this paper, *object* and *resource* are used interchangeably). *Permission* refers to the authorisation to perform some actions (e.g., read, update) on objects. In some systems, the notion of *Access Context* is also important. It concerns properties of the subjects (e.g., location, age) attempting access, the states of objects being accessed (e.g., a *paper* cannot be modified after the proceeding has been published), contextual factors when the access is taking place (e.g., working time or holiday), and access methods (e.g, using a trusted device or not).

Over the past decade, a number of access control models have been proposed. The Role-based Access Control (RBAC) model [10] is the most widely adopted. Ac-

cording to a recent study carried out by NIST (the American National Institute of Standards and Technology), RBAC's adoption is increasing steadily, and up to 50% of the surveyed organisations with more than 500 employees [22]. In the model, one of the key concepts is the *Role*, which refers to different privileges on a system. Users and permissions are then assigned to roles; these assignments govern users' accesses to resources.

Because of its flexibility, the Attribute Based Access Control (ABAC) model [16] has recently gained interest and can potentially be used to replace other AC models. ABAC policies can use any type of attributes, such as user attributes, resource attributes, context attributes, or attributes that represent the relationships between users and resources. ABAC controls accesses based on these attributes and their values.

2.2 AC Vulnerabilities

Broken access control is a class of AC vulnerabilities. According to a recent report by the Open Web Application Security Project (OWASP[1]) in 2013, broken access control is involved in three out of the top ten vulnerabilities: *Missing Function Level Access Control*, *Insecure Direct Object References*, and *Broken Authentication and Session Management* [12].

Missing Function Level Access Control. Due to a lack of proper access enforcement, many web applications check access rights before making functionality or system resources visible (via web links, for example). However, the same check must be carried out on the server side when a functionality or a resource is accessed. Otherwise, attackers can forge requests to get access to the resource without being authorised.

Insecure Direct Object References vulnerability refers to the exposure of direct references to internal resources (such as files). If such internal resources are not adequately protected, they can be maliciously accessed. *Broken Authentication and Session Management* relates to the authentication and session management of an access control mechanism. If they are implemented improperly, attackers can compromise user credentials or impersonate other users to access their private resources.

Information Disclosure dues to improper data sanitisation is another class of AC vulnerability. When displaying information to the end users, especially in unexpected circumstances when invalid data are provided or errors occur, sensitive information must properly be sanitised. Unfortunately, many systems in practice still have this kind of vulnerability, disclosing source code or database information to end users.

Faulty AC policy implementations can be vulnerable to *Privilege Escalation* at-

[1]https://www.owasp.org

tacks. In such cases, users with a non-privileged role maliciously access privileged resources, or users access resources own by others without proper permissions.

2.3 Web Crawling

Web crawling, also referred to as *Web Spidering*, is a technique for the retrieval of web resources. Its basic process is rather straightforward, starting from the URLs of one or a few pages (seeding entry pages), a web crawler (also called spider) extracts hyperlinks from the contents of the pages and then iteratively navigates the web pages addressed by those links [23]. Advanced web crawlers can identify and submit web forms to achieve more thorough exploration of web applications. Recent advances in web technologies (i.e., Web 2.0), where Javascript is used extensively to render user interfaces, implement navigation, and to enable asynchronous communications with web servers, have brought great challenges to web crawling as discussed in [29]. Unfortunately, only limited research attention has been paid to improve web crawlers to support Web 2.0, (see, for example, [19, 7] and references therein).

Web crawling is used for a variety of purposes, such as web archiving, building search engines [23], or reverse engineering of web applications where models of web systems are reconstructed and used for maintenance or testing [6]. In security contexts, it helps scan the "attack surface" of web applications through which security test inputs, i.e., attacks, can be submitted to applications.

3 Working Example

We consider a simple web-based document management (SDM) system, depicted in Figure 1, as a working example for our approach. Its server-side code consists of five server pages: *LogIn*, *Main*, *viewDocument*, *manageDocument*, and *manageUser*; they are located in the following folders: *root*, *root*, *root/user*, *root/manager*, *root/admin*, respectively. The folder *root* is the root server directory of SDM.

When a user has an access to the system, she is sent to the *Login* page and, once authenticated, she is redirected to the *Main* page in which she will be provided links to the *viewDocument* and *manageDocument* pages, depending on her roles in the system. The page *viewDocument* has two parameters, *path* and *id*, which indicate the path to a file and the document identifier, respectively. The page *manageDocument* has three Ajax-based functions, namely *ajax-create*, *ajax-update*, and *ajax-approve* to create, update, and approve documents. These functions are also enabled based on users' roles and, once triggered, they open dynamic forms through which documents can be uploaded and updated.

The administrator of the system has to enter the *admin* suffix manually to the

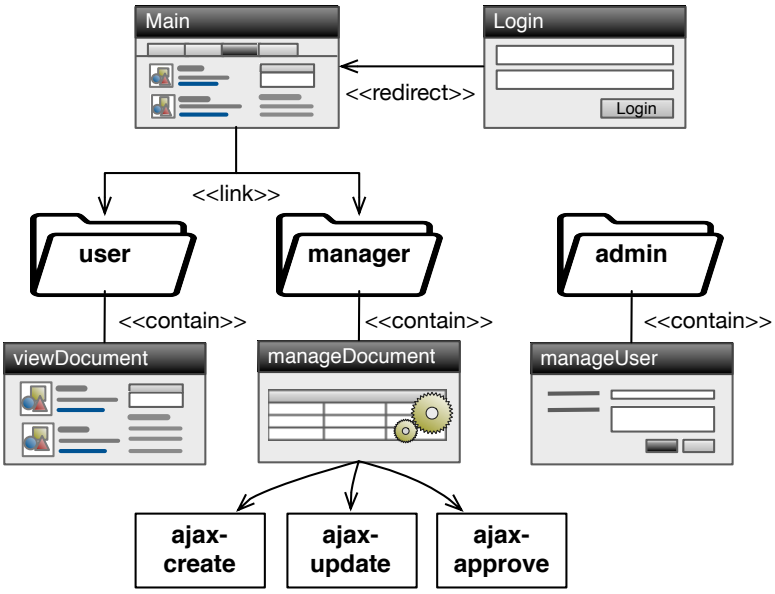

Figure 1: A simple web-based document management system.

URL of SDM in order to access the administrative zone. The page *manageUser* allows him to create, update, delete, and assign roles to users. There is no direct link to the admin area from the *Main* page.

SDM assigns access permissions to roles. In particular, five roles are pre-defined: *adminRole*, *userRole*, *managerRole*, *authorRole*, and *guestRole* with the assigned access permissions as follows:

Role	Access Permissions
userRole	viewDocument
managerRole	viewDocument manageDocument/ajax-approve
authorRole	viewDocument manageDocument/ajax-create manageDocument/ajax-update
adminRole	manageUser viewDocument manageDocument/*

By design, all users can access the *Login* page while only authenticated users can access the *Main* page. From there, depending on the roles of the users, authorised functions (i.e., links) will be provided. For example, users of role *userRole* can only view their documents.

Let us discuss some of commonly encountered AC vulnerabilities (Missing Function Level Access Control) using SDM as an example:
V1: Once logged in, a user is redirected to the *Main* page which only displays links to the allowed pages. For example, assuming a malicious user named Mallory has been assigned to *userRole*, only a list of links of the form *viewDocument?path=...* are displayed on his browser. The developer makes an assumption that by displaying only authorised links a user like Mallory cannot access other pages. However, this assumption is wrong since, based on indexing and suggesting tools like Google, by looking at a browser's history, or by simple guessing, Mallory can easily access other web pages if proper access enforcement is not in place. A malicious user can also try to guess and add some commonly used nouns like *admin*, *backup*, *config* to the main URL in order to access sensitive areas of a web application. Thereby, in the case of SDM, if only the administrator is assumed to know the *admin* suffix and no access control is in place, unauthorised access is likely to occur.
V2: Since web clients can send direct HTTP requests to SDM, a malicious user can create requests to trigger the Ajax-based functions. If accesses are enforced only at the web pages, unauthorised accesses to these functions will occur.
V3: The *path* parameter of the *viewDocument* page can easily be tampered with illegitimate values. For instance, a malicious user can enter a value such as */etc/passwd*, trying to compromise the server. Based on the naming scheme of the parameter, an attacker can guess and access other users' documents. For example, if the values of the path parameter for Mallory are *mallory/1* and *mallory/2*, this pattern can be inferred and Mallory can then try the *alice/[0-9]+* pattern in order to view the documents of the user *alice*.

We propose an approach to address the above issues by discovering resources in web applications like SDM, learning access permissions for different roles in an automated fashion, and then pointing out potential AC problems. We start by introducing some key definitions in the following section.

4 Definitions

In this section, we define the basic concepts used in the paper. We also discuss our proposed XML-based format for the specifications of domain inputs. It facilitates the specifications of web pages, inputs, data types and *input classes* that are used for access testing.

4.1 Basic Concepts

AC Request. AC requests are HTTP requests sent to the system under test (SUT).

Resource. Resources of a web-based SUT are managed through web pages and are identifiable based on URIs (Uniform Resource Identifiers) as well as relevant request parameters and their values. A request parameter is deemed relevant if it affects AC policies, i.e., influences access permissions. In our motivating example, the parameter *id* of the page *viewDocument* is relevant because it helps identify specific documents.
Context. Contexts are situations where AC requests take place. A context is characterised by various factors such as *resource states* (a target resource might have different states that are influencing AC policies), *access history* (preceding access sequences to current AC requests). Besides, other contextual factors like time, geolocation, server availability can also play a role in AC policies.
Access Corpus. The access corpus of a resource contains access instances to that resource. In particular, the access corpus AS of resource r of the SUT is a tuple $AS(r) = (USER, ROLE, ATT, CONTEXT, p)$, where $USER$ is a set of users $\{u_1, u_2, \ldots\}$; $ROLE$ is a set of roles $\{role_1, role_2, \ldots\}$ that are assigned to users; ATT is a tuple of attributes of r: $ATT = (ra_1, ra_2, \ldots)$, where each attribute ra_i can receive a set of values; $CONTEXT$ is a tuple of contextual attributes $CONTEXT = (ca_1, ca_2, \ldots)$, with each attribute capturing a set of concrete context values; and p (for permission) can be assigned one of two string literals: *allowed* and *denied*.

We can also view the access corpus of resource r as a table having the following columns: $USER, ROLE, ra_1, ra_2, \ldots, ca_1, ca_2, \ldots, permission$. For illustration, let us consider two users *Alice* and *Bob* of the SDM in Section 3. Alice is assigned to *userRole* and Bob is assigned to *managerRole*. The resource *viewDocument* has two attributes *id* and *path* representing the unique ID of documents and the dynamically-generated paths, respectively. The access corpus of resource *viewDocument* is presented in Table 1.

Table 1: An example of user access corpus of resource *viewDocument*.

User	Role	ra_1= id	ra_2= path	ca_1 = context_time	Permission
alice	userRole	1	hash1	9AM	allowed
alice	userRole	1	hash2	9PM	denied
alice	userRole	2	hash3	9AM	denied
alice	userRole	2	hash4	9PM	denied
bob	managerRole	1	hash5	9AM	allowed
bob	managerRole	1	hash6	9PM	denied
bob	managerRole	2	hash7	9AM	allowed
bob	managerRole	2	hash8	9PM	denied

4.2 XInput: Specification Language for Domain Inputs

Domain input data have a crucial role for the reverse-engineering of AC policies and for detecting AC issues since, without meaningful domain data, requests sent to a SUT may simply be rejected because they contain invalid data. As a consequence, without the right input data, we may not be able to discover resources, contexts, and their attributes for learning AC policies. Therefore, we systematically consider domain input data, including their specifications and usage, in our approach.

We propose an XML-based language, called *Xinput*[2], for the specifications of domain inputs for web applications. Xinput is inspired by *classification trees* for partition testing [13], but we want to provide the analyst with a flexible and familiar format that enables different and convenient ways of specifying inputs for multiple web pages. Moreover, such format can be used directly, without any additional transformation, to generate executable test cases. To the best of our knowledge, no language exists that fulfils the requirements

In essence, Xinput allows specifying web pages, including URLs, input fields, data types, and *input classes* of each field. A data type indicates a primitive type of input data (e.g, string, boolean, or integer). An input class pertains to two concepts: *equivalence class* and *subtype*. An equivalence class is a concept used in software testing that indicates a family of input values that have the same impact or trigger the same behaviour when they are submitted to the SUT. A subtype denotes concrete values or a specific range of values that an input can receive. Equivalence classes are considered when we have time and domain knowledge to holistically classify input domain into equivalence classes for an input. In practice, however, we may have to work with only a few or a group of input values because of the lack of domain knowledge and time constraint. By supporting both subtypes and equivalence classes, our Xinput format gives input specifications the flexibility of being partial with subtypes or comprehensive with equivalence classes.

In Xinput specifications, a web page is characterised by a unique URL; it may contain one or more input fields. An input field is defined by its name, source (i.e., intended for user or web server), data type (e.g., string or integer), and input classes. As an example, let us consider a web page that asks the end user to select a month. The web page specification in Xinput is as follows, where *month* has 12 input classes corresponding to the 12 months of a year.

[2]The XML schema for Xinput is available at http://people.svv.lu/nguyen/tools/xinput

```
<page urlPath='page.php'>
    <xinput name='month'>
        <dataClz base='string'>
            <enum value='jan'/>
            <enum value='feb'/>
            ...
            <enum value='dec'/>
        </dataClz>
    </xinput>
</page>
```

Listing 1: An example of an Xinput specification using enumerations.

We borrow the well-known restriction specifications in XML Schema Definition [31], including *enumeration*, *boundary*, *length*, and regular *pattern* to define input classes. For instance, when an input has only a limited set of valid values, they are listed as an enumeration. When an input composes of several classes, they can be specified with length restrictions and patterns (for string) or boundaries (for both string and numeric).

```
<page urlPath='book.php'>
 <xinput name='id'>
  <dataClz base='string'>
     <enum. value='1'/>
     <enum. value='2'/>
     ..
     <enum value='10'/>
  </dataClz>
 </xinput>
 <xinput name='title'>
  <dataClz base='string' name='short'>
     <min. value='1'/> <max. value='256'/>
  </dataClz>
  <dataClz base='string' name='long'>
     <min. value='257'/>
  </dataClz>
 </xinput>
 <xinput name='price'>
  <dataClz base='double' name='cheap'>
     <min. value='0.1'/> <max. value='10.0'/>
  </dataClz>
  <dataClz base='double' name='expensive'>
     <min. value='10.0'/>
  </dataClz>
 </xinput>
 <xinput name='ISBN'>
  <dataClz base='string' name='demo'>
     <xs:pattern value='[0-9]{13}'/>
  </dataClz>
 </xinput>
</page>
```

Listing 2: An example of are more complete Xinput specifications.

For example, consider the Xinput specification in Listing 2: a web page called *book.php* has four user inputs *id*, *title*, *price*, and *ISBN*. The first input has 10 classes matching 10 concrete values, the second and third have two classes, characterised by length and boundary specifications. The last input *ISBN* has a class specified by means of a regular pattern for any string of 13 digits[3].

As visible seen from the example, Xinput is flexible, it allows specifying both equivalence classes and subtypes. In the case of the *ISBN* input, we are interested

[3]The example is for demonstration only, it is not the standard ISBN specification.

in only the values that match the defined pattern, i.e., a subtype, and not their equivalence classes. Also, the format is detailed enough to generate concrete values. For instance, for the class *cheap* of the input *price*, we can pick any value within the specified range for price.

5 ReACP Framework

We propose a comprehensive framework, namely **ReACP**, for supporting the reverse-engineering of AC policies and detecting AC issues in web applications. The main structure of *ReACP* is depicted in Figure 2, and consists of seven steps:

1. *Exploratory Access Testing*, where the SUT is exercised to obtain AC logs for subsequent analysis.
2. *Mining and Refining Domain Input Specifications*, where input specifications are mined and revised. Input specifications will be used for AC request generation to further exercise the SUT.
3. *Determining Permission Filters*, where permission filters are specified to help classify AC responses as allowed or denied.
4. *Access Request Generation*, where we use input specifications and combinatorial test generation to generate more diverse AC requests.
5. *Access Testing*, where AC requests are sent to the SUT on behalf of different users.
6. *Inferring AC Policies*, where AC policies are learned after obtaining and analysing AC logs.
7. *Assessment and AC Issue Detection*, where we assess the inferred policies for correctness and use them to detect AC issues.

We discuss these seven steps in detail in the next sections.

5.1 Step 1: Exploratory Access Testing

The goal of this step is to explore the SUT and exercise its functionality in order to populate access logs for subsequent steps. The inputs for this step include a set of user credentials having different privileges with respect to the SUT.

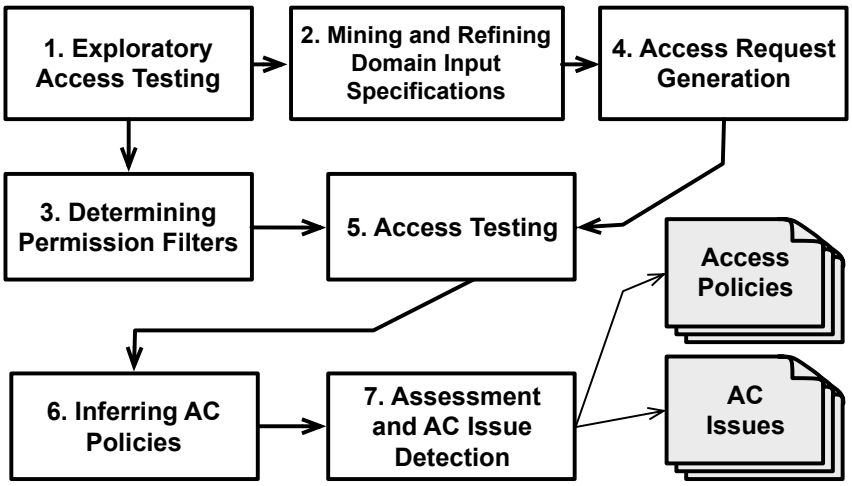

Figure 2: Seven steps of ReACP to reverse-engineer AC policies.

For this step, we rely on a security tool suite called *BurpSuite*[4] (more specifically its spider and proxy) and manual browsing. Firstly, the automated crawling component BurpSuite's Spider starts crawling the SUT with different user credentials to discover web resources. Typically, the spider is provided with one or a few seeding pages to start the crawling process, it then analyses the content of visited web pages to determine links and forms, and follows them to visit more pages. Web crawlers might, sometimes, have problem with the SUTs that have isolated web pages, i.e., there is no link among them. To deal with this, for SUTs that have web root directories (where server code is stored) accessible for analysis, in our approach, we consider all server static resources including server pages, images, folders, and all other files in the directories as seeding pages for spidering. This helps improve the crawler's performance.

Secondly, since crawlers in general have no knowledge about business logics or domain data, manual browsing is useful to further explore the SUT in areas the spider cannot reach, such as the web pages that demand meaningful input data. To do this, a tester can use a browser and interact with the SUT; all requests and their corresponding responses are recorded through the BurpSuite's proxy. Notice that the proxy can also be used to capture execution logs when the SUT is used by end users in production or when the SUT is functionally tested during development. In addition, other spiders with different crawling capabilities such as Crawljax [19] that supports Javascript, can be used in tandem with the proxy to further explore the SUT. Figure 3 shows how different components are integrated for exploratory access testing. The

[4]http://portswigger.net/burp

results of this first step are the resource access logs, which include HTTP requests and responses captured by the BurpSuite's proxy.

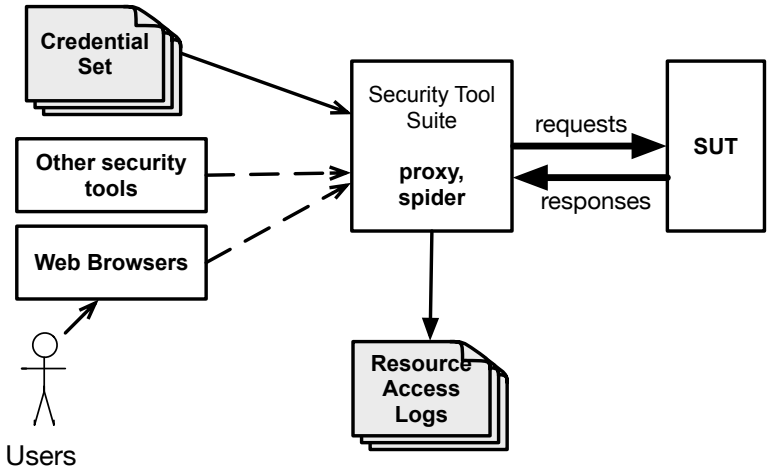

Figure 3: Automated exploration of resource accesses for the SUT. Dashed arrows indicate optional involvement of the other security tools or end users.

5.2 Step 2: Mining and Refining Domain Input Specifications

In Step 1, manual browsing with user-provided input data can help explore and reach the parts of the SUT that the spider cannot. Moreover, such data are also useful to generate meaningful access requests to further exercise the SUT so as to learn AC policies precisely and completely, and to detect AC problems. Therefore, in *ReACP*, we automatically mine partial input specifications from logged data to produce input specifications in the Xinput format and then involve domain experts for refinements.

In the following we discuss how input specifications, including web pages, their inputs, and the classes of each input, are mined. The next section will discuss how to use such specifications to generate access requests.

First, crawling logs consisting of HTTP requests and responses from all exploratory sessions are consolidated to extract: (i) *URL paths* of web pages, (ii) input *identifiers*, i.e. names of parameters extracted from URL query strings and bodies of HTTP requests, and from hyper-links and web forms of HTTP responses, and (ii) all *values* of each input. Values of inputs are extracted as follows:

- *Query strings*: Query strings are a part of URLs in requests, they are parameter-value pairs. Data values in query strings are extracted for the corresponding parameters. For example, in URL *http://examp.le/p?q=10*, from the query string *q=10* we can extract the value 10 for the parameter *q*.

- *Hyper-links*: Hyper-links are in HTTP responses. We consider links that are within the scope of the SUT to extract data values. Extracting data from hyper-links is the same as from query strings.

- *Web forms*: Web forms in HTTP responses can contain hidden inputs with server-generated or predefined values (e.g, in *select* elements). The example given in Listing 3 shows a form in which the input field "month" has only some predefined values. They are extracted for mining purposes.

- *Bodies of POST requests*: POST requests are the results of form submissions with user inputs. The bodies of the requests contain parameter-value pairs that are extracted for specification mining.

```
<html>
<body>
 <form action=‘‘page.php’’>
  <select name=‘‘month’’>
   <option value=‘‘jan’’></option>
   <option value=‘‘feb’’></option>
   <option value=‘‘mar’’></option>
  </select>
 </form>
</body>
</html>
```

Listing 3: An example of an HTML web forms from which we can extract data for the "month" input.

We propose the following five rules to mine input specifications:

- *Rule 1: type*: if all logged values of an input are numeric, then the data type of the input is *numeric* (integer or double, depending on whether all values are integer or not, respectively), otherwise it is *string*.

- *Rule 2: enumeration*: inputs having a limited number of data values (no more than 10 in our current setting) or inputs listed in select elements should be specified as enumerations, where each value represents a distinct class. For example, if in a web page we have an input having a *select* form element that lists

all the countries in the world, then the classes of the input are an enumeration of the countries.

- *Rule 3: length*: if the type of an input is string and Rule 2 is not applicable (i.e., it has more data values), the minimum and maximum length of its data values are used for the specification of the input.

- *Rule 4: boundary*: if the type of an input is numeric and Rule 2 is not applicable, meaning that we might have many numeric data values (at least 10), then we resort to data clustering to learn classes. The idea is to group similar input values into the same class so the analyst can save time revising only a few mined classes instead of considering all the specific values. The clustering technique used is the Expectation-Maximization (EM) algorithm [1], which can estimate the number of clusters in addition to providing the clusters themselves. Each obtained cluster is, then, considered as a class for the input, characterised by the boundaries of the data in the cluster (min, max).

- *Rule 5: pattern*: this rule is an alternative to Rule 3 and can be applied for strings. There exist different approaches proposed in the literature to learn regular patterns from string samples (e.g., [4, 8]), which can be applied to mine input specifications. However, such approaches require a sufficiently large sample as otherwise learned patterns may not be general. We leave the implementation of this rule open for future work.

  The domain analyst can manually analyse the string samples and define classes by mean of regular patterns for the input. We subsequently support the generation of concrete values from regular patterns, as discussed in the next section.

By applying these rules on logged data we can learn initial specifications in which, all web pages, inputs of the pages, their types and initial classes are ready for refinement, where needed.

The output of this step is an Xinput file containing input specifications that will be used in Step 4 for the generation of AC requests.

5.3 Step 3: Determining Permission Filters

This step focuses on determining access permissions ("allowed" or "denied") for AC requests by analysing the responses in the access logs. We define a set of filters that specify how the SUT grants or denies permissions to AC requests.

Access permission of a user to a resource is determined based on HTTP responses, more specifically the standard HTTP status codes [30], and the HTML contents. In addition to using HTTP status codes, some web systems gracefully deny unauthorised

accesses with an "OK Status" web page (HTTP code = 200) but stating, for instance, "Access is not allowed". Because these denial messages usually follow certain patterns, we use regular expressions in our approach to match HTML contents and classify them as *denial* or *normal*, indicating the corresponding requests are denied or allowed, respectively.

The following table represents general filters to classify responses and determine access permissions:

Condition	Permission
HTTP Code = 4xx,5xx,301, any content	denied
HTTP Code = 200,302,304, *denial* content	denied
HTTP Code = 200,302,304, *normal* content	allowed

We may need to customise these filters to specific applications to correctly classify AC responses. These filters will be used later on in Step 5 during access testing.

5.4 Step 4: Access Request Generation

In this step, the *ReACP* uses the Xinput specification file from Step 2 to simulate more access requests to further exercise the SUT. The goal is to determine how the SUT responds to new requests with new data or combined data from different users. The results from these requests will help us learn detailed policies and detect vulnerabilities where one can illegitimately access resources of others.

In Xinput specifications, each web page contains a set of input fields and each input field contains a set of specified input classes. In turn, each of them is characterised by a boundary and length, or a pattern specification. In the case where an input is specified by an enumeration, each enumerated value is considered a distinct class. For instance, the example in Listing 2 has four inputs *id*, *title*, *price*, *ISBN*, which have 10, 2, 2, and 1 class, respectively.

In practice, a web page might have many input fields and each of them can have many input classes. Trying to exercise all combinations of input classes would lead to an exponentially large number of requests to handle. For example, a simple web page with three inputs, each with 10 classes, would result in 10^3 combinations. As a result, in the paper we apply *combinatorial testing* to generate combinations of input classes and achieve an adequate degree of combinatorial coverage while minimising the number of requests to be executed.

Standard combinatorial testing strategies aim, for example, at generating combinations that cover all pairs or all triples of input classes [11, 34]. Several such algorithms and other more recent ones that apply meta-heuristic search [15, 17] have been proposed and developed to efficiently generate minimal sets of combinations.

Moreover, there exist many available tools that support the generation of test combinations. In *ReACP*, we opt for a tool called AllPairs[5], which implements pairwise generation following a greedy strategy, for its ease of integration and fast execution. 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 meta-heuristic search algorithms [24].

In AllPairs, each input class is represented by a label that corresponds to enumerated values or the name of the class. Considering the example in Listing 2, there are four inputs and their corresponding class labels are:

id:	$id_1, id_2, \ldots, id_{10}$
title:	*short*, *long*
price:	*cheap*, *expensive*
ISBN:	$ISBN$

Dealing with all class combinations for the inputs would lead to a total of 2*2*10*1 = 40 combinations. Instead, AllPairs generates only 20 pairwise combinations, including those listed below.

1:	id_1, *short*, *cheap*, *isbn*
2:	id_2, *long*, *expensive*, *isbn*
3:	id_3, *long*, *cheap*, *isbn*
4:	id_4, *short*, *expensive*, *isbn*
5:	id_5, *short*, *expensive*, *isbn*
..	..
18:	id_3, *short*, *expensive*, *isbn*
19:	id_2, *short*, *cheap*, *isbn*
20:	id_1, *long*, *expensive*, *isbn*

Once the combinations are obtained, we concretise them by deriving concrete values from the class specifications to generate actual requests. The concretisation rules defined for different types of specifications are provided below. For illustration, we explain these rules using a combination generated by AllPairs in the above example $< id_1, short, cheap, isbn >$.

- *enumeration*: simply pick the value corresponding to the enumerated element. For instance, we pick value 1 for the input *id*.
- *length*: generate a random string of characters that satisfies the length specifications. For instance, the *short* class represents strings having their length in between 1 and 256. Therefore, our framework generates a random string "abc" for the *title* input.

[5]By McDowell, previously available at http://pairwise.org/tools.asp

- *boundary*: generate a random numeric value that satisfies the boundary specifications. For instance, the *cheap* class requires values in between 0.1 and 10.0. Therefore, our framework generates a random number (e.g., 5.0) within the range for the *price* input.

- *pattern*, generate a random string that matches the defined pattern. In *ReACP*, we use a tool called Xeger[6] for this type of specifications. It converts regular expressions to automata and generates strings (in other words, sequences of labels) from the automata. For instance, for *isbn* the tool generates a string of 13 digits (e.g., "0123456789123") for the *ISBN* input.

As a result, the first combination of classes can be concretised to an HTTP request having four parameters: *id* = 1, *title* = "abc", *price* = 5.0, and *ISBN* = "0123456789123".

5.5 Step 5: Access Testing Execution

This step consists of establishing access contexts (or just *context* for brevity), submitting access requests, and determining access permissions.

5.5.1 Setting up Access Contexts

Contexts are defined by the states of resources being requested and the environment (specified by the time, location, and states of other resources that interact with the targeted resources). Contexts can affect access permissions in many cases and we therefore need to establish the right contexts before submitting any access request. In web applications, contexts are materialised in client states, server states, access time, and location.

For client states, although the HTTP protocol is stateless, web applications have to maintain the state of each and every client in order to provide services according to a business process. For example, an e-commerce system has to ensure whether a user has logged in and added some items to her shopping cart before allowing her to check-out. On the client side, states are stored in cookies, hidden form fields, and query strings with their values set by the server upon interactions with the client. We propose a systematic approach to deal with client states. First, *ReACP* adds an attribute called *source* to the input fields and automatically assigns its value to *SERVER* or *USER* during the mining procedure at Step 2. Cookie and hidden input fields are assigned *SERVER*, otherwise they are *USER* (only inputs of *USER* type needs classification mining and refinements). Second, for any web page that has an

[6]Maintained at https://github.com/bluezio/xeger

input field with a *SERVER* source, a set-up script is prepared to guarantee that the client is at the right state for incoming requests to the page.

Depending on the SUT, server states are defined by means of database or file systems. However, these states should be initialised before executing a request and rolled back after completing a test session or a request when the executed request results into database or file changes. We have observed that, in many cases, such access requests are denied because of missing data which were deleted by previous requests. Therefore, ensuring that the server is in the right state is important as otherwise we might miss the opportunity to learn useful AC policies. To this end, our framework supports executing external scripts for setting up purposes, e.g., populate a database or reset file contents.

For some applications in which the time and location are considered in the AC, server time and client location must also be controlled during the execution of access requests. Changing server time can be done by external scripts similar to those for setting up database or file systems that our framework supports. Likewise, client location can be manipulated by our framework when submitting AC requests.

5.5.2 Test Execution

In *ReACP*, there are three sources to obtain requests for access testing: the spider (Step 1), the end-users through manual browsing (Step 1), and the requests generated from input specifications using pairwise testing (Step 4). Here we want to exercise the SUT to evaluate how it controls accesses by submitting these requests with different users. Recall that we have a set of user credentials, having different privileges, that use when performing exploratory access testing at Step 1, which we can reuse here. Moreover, the tester can easily add more user credentials, if budget permitting, keeping in mind that having more users will help us learn useful information such as whether the SUT differentiates access policies according to users even when they belong to the same role (due to ownership, authorship, specific profiles the users might have, or other reasons that concern AC policies).

Given a user, our framework performs necessary authentication and also ensures that proper context is established before executing access requests. Responses received from the SUT are analysed with respect to the permission filters defined in Step 3, to determine whether the corresponding requests are allowed or denied.

The result of each request execution with a user credential is represented by input values, context values, and the permission labelled as *allowed* or *denied*. The overall test results are a set of access instances AS (corpus) for each resource. Recall that an access corpus is a table of users, roles, resource attributes, contexts and permission.

5.6 Step 6: Inferring AC Policies

Based on detailed access and permission data, for each and every resource of the SUT, we aim at learning implemented access control policies at the right level of abstraction. In some cases, we can apply machine learning directly on the data collected, as described above, to infer AC policies. However, for some resources, pre-processing the data is required to help the machine learning algorithm learn more insightful policies at a more appropriate level of abstraction, as described next.

5.6.1 Pre-processing Data

Recall that our goal is to learn AC policies for analysis purposes, e.g., to determine their compliance with specified or expected policies. Hence, it is desirable that the learned AC policies explicitly capture the key factors that impact access control and abstract away unimportant details. For instance, in our motivating example, a 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 learn, at an abstract level, how document ownership influences AC policies.

More generally, before learning policies, we employ data pre-processing for the resources with which concrete data should be abstracted, to only retain attributes or their relationships that really matter in AC policies. To support pre-processing, we rely on a concept called *meta-attribute*. A meta-attribute represents a relationship between two or more attributes. In the context of access control, we observe common meta-attributes in many systems, including *ownership*, *authorship*, *assignment*. The values of such meta-attributes can often be mapped to a Boolean value, indicating whether the underlying relationship exists or not. The analyst may consider these as a starting point to define meta-attributes.

In *ReACP*, we propose some templates for rules to add meta-attributes to the access dataset and to determine their values:

```
IF att1 IN {set 1} AND att2 IN {set 2}
                   THEN meta-att-name = value-t
                   ELSE meta-att-name = value-f
IF att1 IN {set 1} OR att2 IN {set 2}
                   THEN meta-att-name = value-t
                   ELSE meta-att-name = value-f
```

These rules read as follows: given an access dataset, for access instance, add a new meta-attribute *meta-att-name*, then check the values of attributes *att1* and *att2*, for that instance, to determine if they are contained in the value sets *set 1* and *set 2*,

respectively. Last, assign the corresponding value (*value-t* or *value-f*) to the meta-attribute.

For example, consider the access data presented in Table 1. We choose to capture document ownership with a meta-attribute called *isOwned* by defining this rule to obtain the extended access corpus in Table 2.

```
IF USER IN {alice} AND id IN {1, 3, 5, 7}
                   THEN isOwned = 1
                   ELSE isOwned = 0
```

Table 2: An example of an extended user access dataset for resource *viewDocument* where the meta-attribute *isOwned* is added.

User	Role	id	isOwned	path	time	Permission
alice	userRole	1	true	h.1	9AM	allowed
alice	userRole	1	true	h.2	9PM	denied
alice	userRole	2	false	h.3	9AM	denied
alice	userRole	2	false	h.4	9PM	denied
alice	userRole	3	1	h.9	9AM	allowed
alice	userRole	3	1	h.10	9PM	denied
alice	userRole	5	1	h.11	9AM	allowed
alice	userRole	5	1	h.12	9PM	denied
	..	..	..	..	..	..
alice	userRole	8	false	h.19	9AM	denied
alice	userRole	8	false	h.20	9PM	denied
bob	managerRole	1	false	h.5	9AM	allowed
bob	managerRole	1	false	h.6	9PM	denied
bob	managerRole	2	false	h.7	9AM	allowed
bob	managerRole	2	false	h.8	9PM	denied

5.6.2 Learning AC Policies

The number of access requests to a resource is often very large because it can have many attributes and can be accessed in various contexts. Manually analysing them to reverse-engineer AC policies is impractical. Therefore, we propose to apply a classification technique [32] that yields *decision trees* to learn AC policies from access data. Considering the permissions of access requests as class labels (i.e., *allowed* or *denied*), such classification technique helps identify automatically the attributes (e.g. role, resource attributes) that characterise AC policies. Moreover, the outputs are structured in a decision tree form, making it easy for human experts to analyse them.

In particular, we use the J48 classifier that implements the C4.5 algorithm [25] in Weka [14], to learn decision trees for inferring AC policies. There exists a number of classification algorithms that can produce decision trees, e.g., REPTree, RandomTree [14, 32], but we choose J48 since it supports binary splitting [32] that often results in more concise outputs, reducing the time needed for validation.

A decision tree can be represented graphically in which the root and intermediate nodes represent attributes, the outgoing edges from the nodes are specific values for these attributes, and each leaf node represents a class label (*allowed* or *denied*). In the decision tree, the paths from the root to the leaves represent classification rules. In our context, they match the AC policies that we need to infer. For example, Figure 4 depicts the decision tree learned from the access data dataset of the resource *viewDocument* in Table 1. The classifier identified three attributes to be relevant for access control: *context_time*, *ROLE*, and *id*.

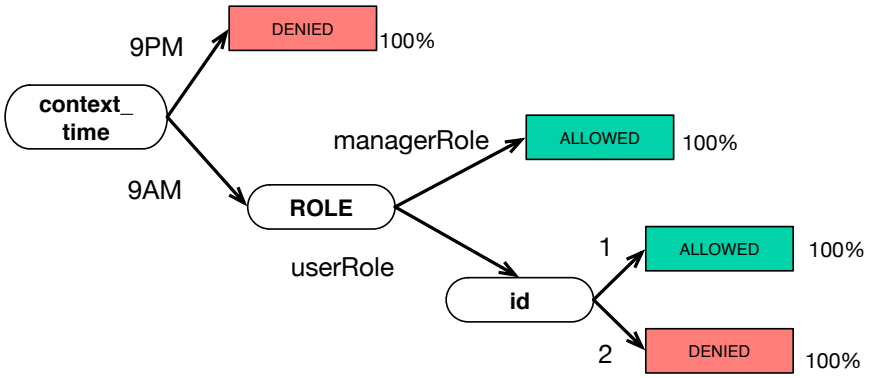

Figure 4: An example of decision tree learned from the access data corpus of the *viewDocument* resource.

Four concrete policies corresponding to the leaves can be derived from the tree include:

```
P1: IF resource = 'viewDocument'
      AND context_time = '9PM' THEN denied

P2: IF resource = 'viewDocument'
      AND context_time = '9AM'
      AND ROLE = 'managerRole' THEN allowed

P3: IF resource = 'viewDocument'
       AND context_time = '9AM'
       AND ROLE = 'userRole'
```

```
        AND id = ‘1‘ THEN allowed
P4: IF resource = ‘viewDocument’
     AND context_time = ‘9AM’
       AND ROLE = ‘userRole’
       AND id = ‘2‘ THEN denied
```

In our context, our objective is to use decision trees to accurately characterise and explain the logged access data (training data); the decision trees are not used for prediction as it is often the case. We are therefore not concerned with over-fitting the training data but rather with capturing as many AC rules as possible. Therefore, we recommend to turn off the pruning [32] option of J48 to generate the most precise tree possible.

From the above example, we observe that a set of policies will tend to grow large since the *id* attribute with concrete values (id = ‘1‘) appear in the policies. There could be many distinct *id* values in an access dataset. Now, if we apply machine learning on the data in Table 2, with one extended meta-attribute (*isOwned*), we can learn more abstract and meaningful policies that do not depend on the number of *id* values. For instance, the following policy advantageously replaces the set of policies above:

```
P1’:
     IF resource = ‘viewDocument’
      AND context_time = ‘9AM’
      AND isOwned = true THEN allowed
```

5.7 Step 7: Assessment and AC Issue Detection

A classifier also provides a *prediction confidence* for each leaf node of the trees. This measure indicates the homogeneity of the data sample that falls into a leaf node in terms of the ratio of accesses that are allowed or denied. In this respect, we define a *consistent* rule as a rule that predicts allowed or denied with 100% confidence given its corresponding leaf node. The other rules are referred to as *inconsistent*, meaning that in their corresponding leaf nodes some accesses are allowed while others are denied. In the context of access control, such inconsistent rules may indicate potential problems in the AC implementation or a lack of information regarding the attempted accesses (e.g., history, parameters).

In an ideal situation where there is no noise in the training data (e.g., wrong labelling or error in attribute values), we should expect all rules to be consistent since the algorithm can grow each branch of the tree deeply enough to perfectly classify the training examples. Inconsistencies may be due to implementation errors that lead to inconsistent responses of the SUT, even for the same request. Another reason is that some relevant information in the training data might be missing. For

instance, if the contextual attribute *access-time* is relevant in AC policies, some actual access instances in the training data corresponding to the same request might have different labels because they are issued at different times. However, since *access-time* is not included in the training data, the leaf node pertaining to those instances is inconsistent.

In *ReACP*, we take several measures to mitigate the missing information problem. First, in Step 1 - Exploratory Access Testing, we rely not only on automated crawlers but also enable the analyst to interact and explore the SUT manually. In this way, web pages that are difficult for the crawlers to reach (because they demand meaningful domain inputs) can also be exercised. Second, in Step 2 - Specification Mining, we automatically mine domain specifications and solicit all resource attributes. This helps avoid missing attributes by human mistakes when such specifications are defined manually from scratch. Finally, in Step 5 - Access Testing, access contexts are also carefully prepared such that if they affect AC policies, they will be included in the training data for inferring policies.

Eventually, all the inferred rules should be validated to identify potential AC problems and discrepancies with expected AC requirements. Toward this end, since their number could be large, we suggest guidelines for the analyst to pay special attention to the following types of resources and their corresponding access rules:

- Resources that allow access to all users; we have observed many cases where such resources have been left unprotected by the implemented AC mechanism.
- Resources that are related to the database, configuration, installation, backup files, and other static documents. The development team might have forgotten to remove them or properly protect them before deployment. There could potentially be insecure direct accesses (without authentication) to them.
- Resources with corresponding inconsistent AC rules. The analyst should check whether inconsistency is due to AC implementation mistakes or other reasons that affect access control.

5.8 Tool Implementation

We have developed a tool called ACMate[7] with graphical user interfaces to implement and support the adoption of ReACP. Figure 5 is a screen shot of ACMate. The tool is integrated with BurpSuite to leverage its features, including its proxy, spider, and site map. Key features of ACMate include: (1) exploring the web applications to uncover resources, (2) generating the AC test requests and performing AC testing on behalf of various users, and (3) inferring AC policies.

[7]Available at http://people.svv.lu/nguyen/tools/acmate/

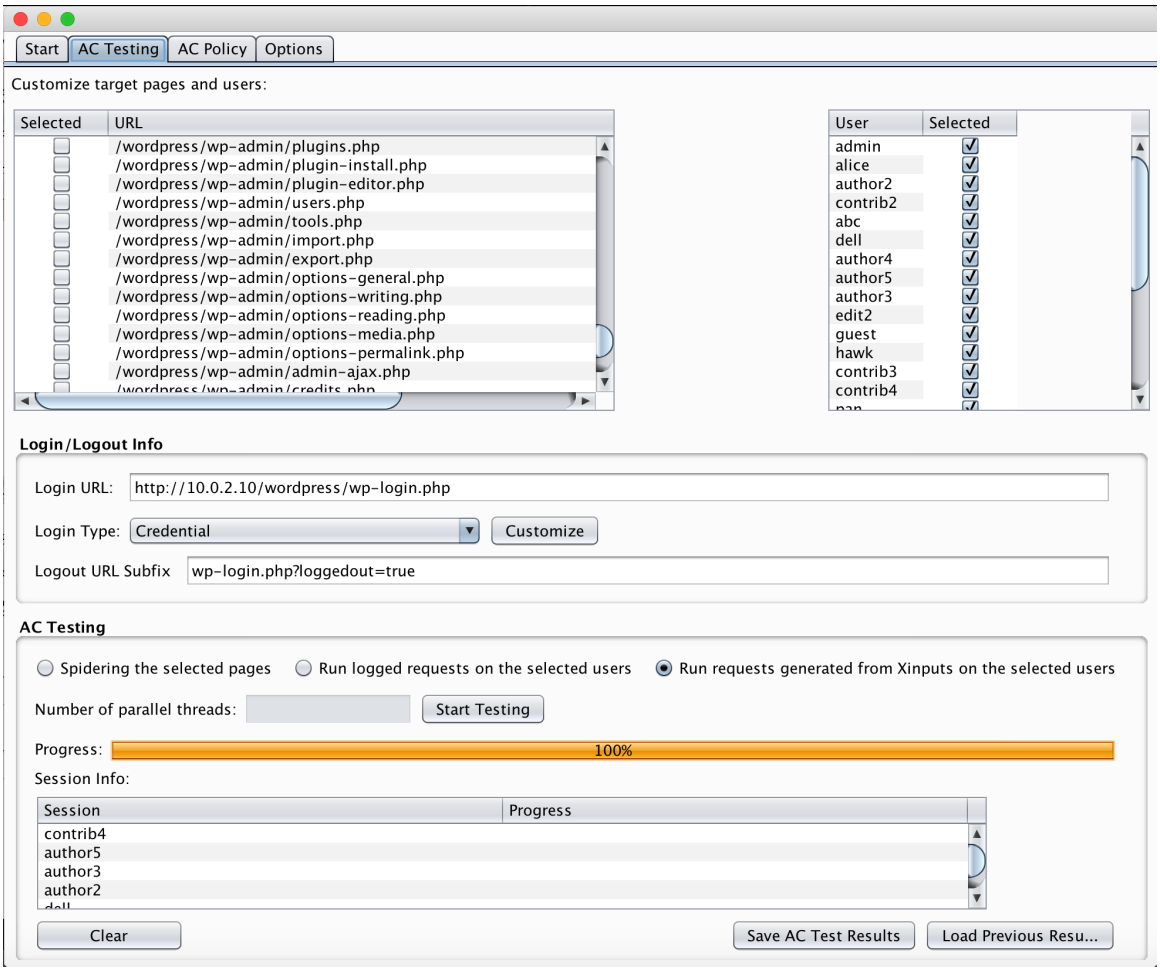

Figure 5: ACMate: a tool suite for the reverse engineering and testing of access control policies.

ACMate provides the user with an interface to specify input configuration, consisting of a list of users' credentials (*username* and *password*) and a set of permission filters for classifying access responses.

ACMate can also mine input specifications automatically. Domain input specifications are extracted from proxy logs and saved in an XML-based XInput file. The analyst can easily work with the file to check and refine the mined specifications. During AC testing, ACMate uses such specifications to generate test requests and execute them.

In our tool, the decision tree algorithm J48 [14] is used to infer the AC policies from the access logs obtained after performing AC testing.

6 Experimental Evaluation

In this section, we discuss the experiments that we have carried out to evaluate our approach. We investigate whether the proposed approach is effective in inferring correct AC rules and in helping the detection of AC vulnerabilities. We also assess the usefulness of input specification mining and the role of combinatorial testing in the proposed approach.

Since ReACP aims at learning policies that have actually been implemented in the systems under test, we need to validate the policies to assess whether or not they reflect correctly the implementation. Therefore, the first research question can be stated as follows:

RQ1: *Does ReACP correctly infer the AC policies that are implemented in the SUT?*

We check the inferred policies for inconsistency and validate them against stakeholders' AC requirements to identify discrepancies, as discussed in Section 5.7. 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 issued by authorised bodies). Therefore, it is interesting to evaluate whether the policies inferred with ReACP help pinpoint AC issues.

RQ2: *Do the inferred AC policies help detecting AC issues?*

In ReACP, we propose to leverage domain input specifications and combinatorial testing to generate more requests and increase their diversity. The goal is to exercise the SUT more thoroughly, with meaningful data, to uncover resources and their relevant attributes in order to learn a more complete set of policies. Hence, we investigate the role of these factors in the next research question:

RQ3: *Do input specifications and pairwise test generation help learn a more complete set of policies in terms of the coverage of policy attributes (roles, users, resource attributes)*?

Manually specifying the domain inputs is expensive, especially when we test large SUTs with many inputs that might be poorly documented. Therefore, our implemented tool for ReACP supports the analyst in specifying domain inputs. It mines automatically input specifications from access logs into the XInput format, which contains web page URLs, input parameter identifiers, types, and classes. Our fourth research question addresses the usefulness of input specification mining.

RQ4: *To what extent is input specification mining helpful for developers to specify domain inputs?*

In the following sections, we first describe the selected subject applications and the procedure of setting and executing the experiments. Then, we present and discuss the experimental results for the above research questions.

6.1 Subject Applications and Settings

We evaluate ReACP on three open-source Web applications: WordPress[8], TaskFreak[9], and iTrust[10]. WordPress is a widely used content management system (CMS) for blogging and news publishing. TaskFreak is a popular open-source system for project management. We deploy the latest multi-user version of TaskFreak (v0.6.4). iTrust is a healthcare system used in educational courses about access control and testing. WordPress and TaskFreak are deployed on an Apache HTTP Server while iTrust is deployed on Apache Tomcat under a Linux server. While all the applications support the role concept and assign permissions to roles, the implementation of AC enforcement is different among applications. For instance, WordPress allows permission administration at runtime while TaskFreak and iTrust hard-code role permissions. WordPress and TaskFreak take into account ownership relationship between users and resources, while iTrust does not.

Table 3 shows the size of the subject applications in terms of number of server pages (ending with .PHP or .JSP, which include also library files), and number of lines of code. The data in the table were collected using *cloc*[11], a popular tool used by developers for measuring code metrics. Apart from having different sizes and

[8]https://wordpress.org
[9]http://www.taskfreak.com
[10]http://agile.csc.ncsu.edu/iTrust/wiki/doku.php
[11]https://github.com/AlDanial/cloc

being developed in different languages, the applications also implement different AC enforcement mechanisms. These make them interesting subject applications for our evaluation.

Table 3: The open-source Web applications used in the experiments.

SUT	Language	Number of server pages	LOCs
WordPress	PHP	482	133,023
iTrust	JSP	220	23,859
TaskFreak	PHP	180	19,448

Each application comes with a set of hard-coded roles having different privileges. For example, the developers of TaskFreak (used for project management) have predefined four roles, *administrator*, *manager*, *intern*, and *guest*, which are assigned to users depending on their roles in managed projects. Similarly, iTrust has eight roles corresponding to various duties in a hospital, WordPress provides five roles for authoring and editing blog posts or news articles. In particular, WordPress also allows the administration of user-role assignments. Therefore, we consider a special role, namely *no-assignment*, for users who own an active account in the application but for which the account has not been assigned to any role. This is to check whether such users can carry out any unauthorised operations.

For each SUT, we create a set of users and assign them to the available roles. We also populate data for each user (e.g. creating new posts, set up the ownership to the posts). As discussed in Section 5, various factors such as users' roles and resource ownership can influence AC policies. Therefore, in our experiments, we prepare data and ensure the coverage of users' profiles, user-role assignments, and add at least two users per role, each of them having different ownership settings on resources. This process led to the number of users per role reported in Table 4.

We also need to account for interferences among testing sessions where, after testing with one user, resource states might be changed and affect the access permissions of other users. To do so, we use a technology offered by virtual machine platforms (e.g., VirtualBox or Parallels Desktop) to quickly save and restore states of the systems before and after each test session.

6.2 Procedure

In our experiments, we follow the seven steps of *ReACP*. We use the automated features of our tool, ACMate, to support the tasks in these steps. Specifically, for each subject under test, in Step 1 - *Exploratory Access Testing*, we collect data about

Table 4: Number of users per role used in the experiments.

SUT	Role	Number of users
WordPress	administrator	1
	subscriber	5
	author	5
	contributor	5
	editor	5
	no-assignment	3
TaskFreak	administrator	2
	manager	3
	intern	3
	guest	3
iTrust	administrator	2
	er	2
	hcp	3
	lt	2
	patient	4
	pha	2
	uap	2
	tester	1

the resources and accesses to the SUT by using a web browser to initiate the discovery. We enable the proxy option in the browser and set it to direct the requests/responses to BurpSuite's proxy. We use the browser to authenticate the SUT website with a test user credential so that a valid session is set up before using the spidering mode in ACMate for automated crawling. After that, we use the same browser to manually interact with the SUT by exercising all functions provided and we simulate different access situations for the links which are missed in the initial step (see Section 5.1). At the end of this step, we use ACMate's mining feature to analyse the resulting output and generate an Xinput file.

We revise the Xinput items to check whether Xinput data classes are valid and relevant to be used in our experiments. For example, ACMate automatically mines the logs of *addTelemedicineData.jsp*, a web page of for iTrust, and extracts an enumeration (Rule 2, Section 5.2) containing two concrete values, 25 and 60, for its *diastolicBloodPressure* attribute. 25 is an invalid value which was randomly entered by the spider. We, therefore, update this attribute value class to match the valid range $[40, 150]$. The revised Xinput is later used to generate the test requests in Step 4.

For correctly classifying access permissions, based on the generic AC permission filter patterns discussed in Section 5.3, we define five content patterns for iTrust. Typically, the responses having HTTP codes 301, 40* or 50* with specific content patterns (such as "Server Error" or "Request timeout") and the HTTP 200 responses containing "You are not authorized" clearly state that the corresponding access requests are *denied*. We also isolate the requests causing error messages (e.g. "ITrustException" or "NullPointerException") and investigate them separately. They are often related to AC issues because of exposing sensitive information, such as source code. Similarly, we define four content patterns for TaskFreak. For WordPress, we need 17 content patterns in its permission filter set.

For resources where relationships among attributes can help characterise AC policies, we apply the pre-processing step and define meta-attributes representing the relationships. In our experiments, the mapping templates described in Section 5.6.1 are sufficient for the selected applications. Finally, we use ACMate's AC policy inference function to infer the AC policies.

To assess the inferred policies when addressing research questions RQ1 and RQ2, it is important to differentiate some key artefacts and their links. As depicted in Fig. 6, since inferred policies are learned by executing the actual implementation with some AC configurations that are defined by the system administrator for controlling accesses, we inspect the inferred policies and compare them against the implementation for correctness (RQ1). Moreover, we define *Gold Standard (GS) Policies* as policies that the stakeholders expect the system to enforce. We derive GS policies of an application from its requirements documents, when they exist, and from the AC configurations. For finding AC issues (RQ2), we compare the inferred policies with

GS to detect discrepancies and then analyse their causes.

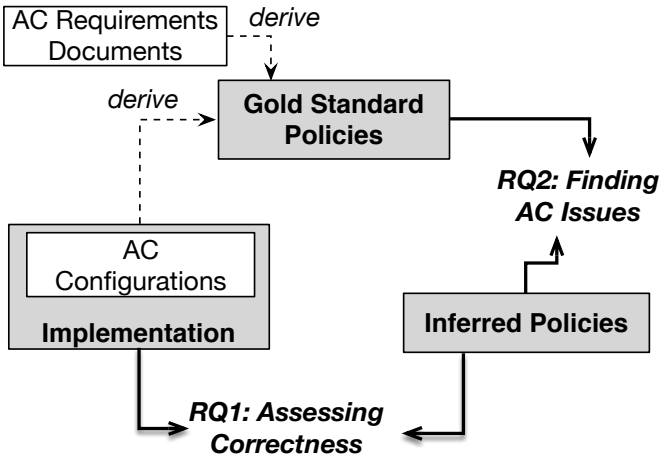

Figure 6: Three key artefacts in assessing the research questions RQ1 and RQ2: Gold Standard Policies, Implementation, and Inferred Policies.

6.3 Results for RQ1: Inferring Correct AC Policies

Recall that each resource is associated with an access dataset. The decision tree learned from the data characterise the access policies relevant to the resource. The paths from the root of the tree to its leaves represent specific policies (examples are given in Section 5.6.2). To address RQ1, we consider only the main resources (i.e., web pages) that are intended for user interactions, excluding all library pages[12]. Finally, we count the number of inferred policies and assess their correctness.

We consider the policies that are learned from all AC requests, including those from automated crawling and those that are generated using domain input specifications and pairwise test generation. Moreover, we have also carried out data pre-processing for 4 (out of 35) resources of WordPress, 3 (out of 6) resources of TaskFreak, and 10 (out of 205) resources of iTrust. Such resources initially led to a large number of detailed policies, at the wrong level of abstraction, thus making them hard to understand.

To illustrate the effect of pre-processing, let us consider a resource of TaskFreak, *project_edit.php*. If we do not apply pre-processing, 471 AC policies would have been inferred. However, when we use a meta-attribute representing the project-user

[12]Library pages of the selected applications control accesses in the same manner, all users are allowed or denied, and *ReACP* learns them correctly.

assignment relationship (*isAssigned*), the number of AC policies reduces to three, without loss of information, as shown below:

```
1: IF isAssigned = 0 AND ac-role = manager
                                  THEN allowed
2: IF isAssigned = 0 AND ac-role != manager
                                   THEN denied
3: IF isAssigned = 1 THEN allowed
```

These more abstract policies are much easier to understand: if a user is assigned to a project, certain operations can be performed. Otherwise, the permission is *allowed* if and only if the role of the user is *manager*.

Table 5: The correctness of inferred policies.

Application	#Resources	#Inferred Policies	#Correct Policies	%
WordPress	35	1,391	1,388	99.78
iTrust	205	1,518	1,441	94.93
TaskFreak	6	65	62	95.38

Table 5 shows the total numbers of inferred AC policies and the number of correct ones corresponding to the three applications, WordPress, iTrust, and TaskFreak. As we can observe, the percentage of correctly inferred policies for the considered resources is high, thus suggesting that our *ReACP* framework is effective in learning correct policies.

We investigated the policies that are incorrect and found two causes. The first one is due to incomplete context initialisation during test execution. As discussed in Section 5.5.1, contexts affect access permissions and missing contextual information leads to denial of accesses. For example, WordPress uses a dynamic parameter called *nonce* to control the order of AC requests and all requests without *nonce* are denied. For some of our executed requests, their contexts were not completely set and incorrect policies were learned.

The second cause for incorrect policies is related to the use of permission filters. Currently, our filters rely only on response codes and contents i.e., containing some specific texts that indicate access permission. They fail to recognise the denial cases in which we must take into account the relationship between the requesting users and the target resources in classifying access permissions. For example, in TaskFreak, when a user adversarially requests profile information of other users, TaskFreak returns his own profile instead of replying with a denial message. Only analysing the profile content cannot help classify such request correctly. Regarding the permission filters of WordPress and iTrust, their filters are not refined enough to properly classify some

responses with HTTP code 200. For this particular response family, one needs to check the content of the responses for "denial" messages (discussed in Step 3 of the framework). We have missed some of these messages, thus leading to their responses being wrongly classified and therefore learning incorrect policies.

6.4 Results for RQ2: Detecting AC Issues Using Inferred AC Policies

To answer RQ2, we derive gold standard (GS) policies from the AC configurations defined by system administrators and requirements documentations that come with the applications. The details of the GS policies of each application are described in Appendix. Inferred policies are then compared to GS to find AC issues, including errors in enforcement implementation or vulnerabilities. We inspect each suspected issue by testing and analysing the applications with the corresponding requests to ascertain whether they are errors or vulnerabilities.

Table 6: Vulnerabilities found thanks to *ReACP*.

SUT	AC Vulnerability	Vulnerable Resource
iTrust	Insecure Direct Object References	44
	Missing Function Level Access Control	38
TaskFreak	Privilege Escalation	4
	Information Disclosure	5
WordPress	Privilege Escalation	1

Table 6 summarises the number of vulnerabilities found. As a student project, iTrust has many vulnerabilities (82 in total). In contrast, since TaskFreak is a popular open-source project management system, it was unexpected to find seven new AC vulnerabilities that had not been published, to the best of our knowledge. We have informed the developer of TaskFreak and submitted these vulnerabilities to the NVD database (US National Vulnerability Database) [13]. For WordPress, we found a known vulnerability that has been detected and fixed by the subsequent version of WordPress. We can thus conclude that, by assessing the inferred policies and investigating those that differ from GS, many new vulnerabilities have been found. This supports the usefulness of the inferred policies in detecting AC issues.

In what follows, we describe in detail the vulnerabilities.

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

6.4.1 iTrust

In iTrust, we found 44 resources that are *Insecure Direct Object References* vulnerabilities. Those resources were accessible by all users. They returned Java exceptions showing implementation errors. The source code and database information is also disclosed in the error contents.

We also found that 38 resources located in important directories (*util*, *errors* and *DataTables*) were not protected by the implemented AC rules (*Missing Function Level Access Control* vulnerability). They did not check user authorisation. These resources can leak important information, such as showing database tables or transactional logs which can be used in privilege escalation attacks.

Among the unprotected resources for end users, we found three critical privilege vulnerabilities. First, the */util/resetPassword.jsp* page allows any user (even an anonymous user without authentication) to change passwords of any other users. Second, the */util/getUser.jsp* page is accessible without proper authorisation enforcement and displays users private data. Third, the */errors/reboot.jsp* page allows anyone to reboot the web server, which might render the system inaccessible to all users.

6.4.2 TaskFreak

In our experiment with TaskFreak, we found five pages containing nine AC vulnerabilities. Two pages, *project_edit.php* and *user_edit.php*, contain four privilege escalation vulnerabilities; a malicious user can perform unauthorised operations on resources of others. All the five pages can result in information disclosure.

- *project_edit.php* has three privilege escalation vulnerabilities. One of the inferred AC policy shows that any user can open the "create new project" page by requesting *project_edit.php*?$id = 0$ and create a new project. This is incorrect when comparing with the GS policies for intern and guest users - projects). Another AC policy for the same resource also shows that *manager* users can view any projects, including projects that are not assigned to him. This is against the GS policies for *manager*. In the last policy, we found that the a *moderator* user can assign himself to a higher position (e.g., *leader*). This is also against the GS policies.

- *user_edit.php* has one escalation vulnerability. Amongst the inferred policies, one policy suggests that a user can change password of other users provided that the two password attributes *password*1 and *password*2 have the same value. This AC policy, however, is incorrect with the GS policy, stating that only *administrator* users can edit other users' account.

- *rss.php* has one information disclosure vulnerability. One of the inferred policy shows that any user can see the list of tasks associated with other users. This is incorrect when comparing with the GS policies for *guest* users.

Apart from *rss.php*, other four pages (*user_list.php*, *user_edit.php*, *project_list.php*, *project_edit.php*) are also vulnerable to disclose information. When they are requested with invalid data (such as out-of-range values), they return error pages showing detailed database queries.

6.4.3 WordPress

In the case of WordPress's *edit.php* page, when analysing its inferred policies we realised a discrepancy between them and the GS policies. The inferred AC policies show that the operation to change a post's status to "published", which is reserved to the administrator and editor, is also allowed to the users of role *contributor*. This violates the GS and, hence, indicates a privilege escalation vulnerably in the version of WordPress used in the experiment (3.8.1).

Such a vulnerability permits malicious contributors to post content without approval. It was detected thanks to our use of combinatorial testing that combines a user of role *contributor* with the operation to change posts' status, which is not available to the user through the user interface. Note that the vulnerability has been fixed in subsequent versions.

6.5 Results for RQ3: The Role of Input Specifications and Pairwise Test Generation

For a given resource, a set of policies is complete if it covers all the attributes that affect the access permissions to the resource. In other words, we assess the completeness of the policies related to a resource if all user roles, contexts, and attributes of the resource are accounted for by the policies.

We investigate the role of domain input specifications and pairwise test generation (XInput for brevity) in learning more complete policies. In *ReACP*, we propose to take meaningful domain inputs into account by mining input specifications and involving the analyst to revise them. Based on such information, we apply pairwise test generation to obtain diverse AC requests which consider the interactions among input parameters and execute them.

To answer RQ3, we compare the output policies that we can learn from two scenarios. The first one considers only the AC logs obtained from the exploratory testing step of *ReACP*. These logs are the result of automated crawling and manual usage (logs from users' interaction with applications). The second scenario (with XInput)

also mines input specifications and exercises the SUTs with additional requests using combinatorial testing. We consider here the inferred policies for the same resources as in RQ1: 35 resources of WordPress, 205 resources of iTrust, and 6 resources of TaskFreak.

Table 7: The number of executed tests and inferred policies without or with combinatorial testing.

Subject	Exploratory Testing Only		With Combinatorial Testing	
	#Tests	#Policies	#Tests	#Policies
WordPress	1,859	149	8,943	1,391
iTrust	122,343	672	222,840	1,518
Taskfreak	59,629	41	342,517	65

Table 7 summarises the number of access requests (aka test cases) executed and the number of policies inferred without and with combinatorial testing. We can observe that the number of tests increases by a large extent thanks to the use of domain input specifications and combinatorial testing, indicating that the subjects are exercised much more thoroughly. Also, the number of inferred policies increases significantly when combinatorial testing is considered. Given that the same number of resources is considered in both cases, the increase indicates that more attributes and increased diversity in their values have been taken into account to produce more policies. Therefore, input specifications and additional combinatorial testing are beneficial. Based on a careful inspection of the inferred policies for TaskFreak, and random samples of 10 resources of WordPress and iTrust, we could confirm that domain input specifications and pairwise test generation actually led to more complete policies.

The following concrete example shows the decision tree learned only from exploratory testing logs for $getPersonnelID.jsp$, a resource in iTrust.

```
forward = staff/editPersonnel.jsp
| ac-role = patient
| | ac-method = GET: allowed
| | ac-method = POST: denied
| ac-role != patient
| | ac-role = tester
| | | ac-method = GET: allowed
| | | ac-method = POST: denied
| | ac-role != tester
| | | ac-role = PHA
```

```
| | | | ac-method = GET: allowed
| | | | ac-method = POST: denied
| | | ac-role != PHA
| | | | ac-role = ER
| | | | | ac-method = GET: allowed
| | | | | ac-method = POST: denied
| | | | ac-role != ER: allowed
forward != staff/editPersonnel.jsp: allowed
Number of Leaves : 10
```

In the above decision tree, many values of the *forward* attribute that determines access permission of AC requests and many roles of iTrust, e.g., *administrator*, *hpc*, are not present. Therefore, the 10 policies that could be extracted from the tree are incomplete. Moreover, some policies are incorrect as well. For instance, the last one states that accesses in which *forward* differs from $staff/editPersonnel.jsp$ are allowed for all users, which is not correct. When considering Xinput, we learned a more complete decision tree, producing 24 correct policies, in which all roles and values of the *forward* attribute were considered.

6.6 Results for RQ4: Input Specification Mining

We evaluate here the usefulness of the input specification mining feature of *ReACP*. We look at the size of the mined Xinput files produced automatically by our tool to estimate the manual effort that would have otherwise been needed. We also count the number of Xinput elements (input fields, classes) that need to be refined to evaluate the quality of the mined specifications: the better the mined specifications, the lower the number of refinements.

Table 8: The size of mined specifications and the number of refined elements.

Subject		Mined	Need refinement
WordPress	#Input fields	1,627	177
	#Classes	6,643	1,857
iTrust	#Input fields	662	16
	#Classes	1,058	46
Taskfreak	#Input fields	34	18
	#Classes	512	327

Table 8 shows the results for RQ4. We can observe that the number of input parameters of the subject applications is as high as 1627. If we had had to specify them manually, i.e., identifying their web pages, names, data types, and input classes,

it would have taken an unrealistic amount of time. Thanks to our specification mining feature, this laborious task has been largely automated. Moreover, the number of parameters that need refinement is small: 16 for TaskFreak, 18 for iTrust, and 177 (out of 1627) for WordPress. The number of classes that need refinement is also significantly smaller than their total number. This indicates that the mined specifications are largely reusable as generated. From the above we conclude that with its mining feature, *ReACP* helps substantially reduce the effort needed for input specifications.

7 Related Work

Our work belongs to the research domain focusing on applying dynamic analysis to infer program specifications and detect security vulnerabilities. In the area of access control analysis, a similar technique called "differential analysis" has been proposed to detect authorisation vulnerabilities in web applications [27]. It involves crawling a system under test using different authenticated users' sessions and unauthenticated ones in order to determine which portions of the system are accessible from which users. Our approach also involves web crawling with a set of user credentials but, unlike the work in [27], our approach accounts for Javascript and "unlinked" areas of web applications. Moreover, the goal of our approach 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 for roles, not users. As a result, our approach is more scalable and can work with more complex web applications.

Noseevich *et al.* extended differential analysis with the role concept and the notion of user cases, which represents roles' actions and their dependencies [20]. These are inputs defined by a human operator. The approach, then, considers sequences of use cases, iterates through these sequences and applies differential analysis for each user case in order to detect insufficient access control. Our approach differs in a number of aspects. First, our goal is to infer AC policies that, on the one hand, can be validated 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 omitted in the related work.

Alalfi *et al.* have worked on the reverse engineering of RBAC models for web applications. Source code transformation and instrumentation techniques and tools were proposed to recover UML-based structural models and behavioural models [2]. Such models act as inputs for another model transformation technique to construct RBAC models that can be used to check against security properties [3]. Our approach uses a proxy to obtain access traces, and hence, we do not need to modify application source code. In addition, we take into account all types of server resources, includ-

ing static files (e.g., PDF documents, images), which are not considered in Alalfi's approach.

Slankas *et al.* [28] proposed an approach to extract access control policies from documents written in natural language. Natural language processing techniques are used to extract access control concepts like subjects, actions, and resources. One of our applications used for evaluation is also used in [28]. Xiao *et al.* [33] proposed an approach called the Text2Policy to extract automatically AC policies in XACML format from software documents and resource access information written in natural language.

8 Conclusion

Broken access control (AC) is a widely recognised security issue in web applications. As web applications and their AC mechanisms are becoming 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.

We propose in this paper a testing-based, semi-automated framework, called *ReACP*, to infer the AC policies of web applications from their implementation. We have also developed a tool available to practitioners. The proposed approach rests on the integration of a popular security tool suite, domain input specifications, combinatorial testing, and a machine learning algorithm. It first explores a SUT automatically to obtain initial access logs. They are, then, analysed and mined for domain input specifications. These specifications are refined by analysts and then used to generate, in a systematic way using a combinatorial strategy, more access requests to further exercise the SUT. Finally, machine learning is applied by using data in access control logs to infer AC policies.

We have evaluated *ReACP* on three representative, open-source web applications. The results are promising and show that *ReACP* can infer correct and complete policies, which can in turn help detect actual AC vulnerabilities.

References

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

[2] M. Alalfi, J. Cordy, and T. Dean. Automated reverse engineering of uml sequence diagrams for dynamic web applications. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 287–294, April 2009.

[3] M. Alalfi, J. Cordy, and T. Dean. Recovering role-based access control security models from dynamic web applications. In M. Brambilla, T. Tokuda, and R. Tolksdorf, editors, *Web Engineering*, volume 7387 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin Heidelberg, 2012.

[4] A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, and E. Sorio. Automatic generation of regular expressions from examples with genetic programming. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 1477–1478. ACM, 2012.

[5] N. Damianou, A. Bandara, M. Sloman, and E. Lupu. A survey of policy specification approaches. *Department of Computing, Imperial College of Science Technology and Medicine, London*, 2002.

[6] G. Di Lucca, M. Di Penta, G. Antoniol, and G. Casazza. An approach for reverse engineering of web-based applications. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 231–240, 2001.

[7] C. Duda, G. Frey, D. Kossmann, and C. Zhou. Ajaxsearch: Crawling, indexing and searching web 2.0 applications. *Proc. VLDB Endow.*, 1(2):1440–1443, Aug. 2008.

[8] H. Fernau. Algorithms for learning regular expressions. In *Algorithmic Learning Theory*, pages 297–311. Springer, 2005.

[9] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based access control - 2nd edition*. Artech House, 2007.

[10] D. Ferraiolo and R. Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[11] P. Flores and Y. Cheon. Pwisegen: Generating test cases for pairwise testing using genetic algorithms. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, volume 2, pages 747 –752, june 2011.

[12] T. O. Foundation. Owasp 10 most critical web application security risks. Technical report, OWASP, 2013.

[13] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[14] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and 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(7):650–670, July 2014.

[16] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, et al. Guide to attribute based access control (abac) definition and considerations. *NIST Special Publication*, 800:162, 2013.

[17] Y. Jia, M. B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 540–550, May 2015.

[18] H. T. Le, C. D. Nguyen, L. Briand, and 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, pages 27–37, New York, NY, USA, 2015. ACM.

[19] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[20] G. Noseevich and A. Petukhov. Detecting insufficient access control in web applications. In *SysSec Workshop (SysSec), 2011 First*, pages 11–18, July 2011.

[21] OASIS. Extensible access control markup language (xacml). Technical report, OASIS, 2003.

[22] A. C. O'Connor and R. J. Loomis. Economic analysis of role-based access control. *RTI International report for NIST*, 2010.

[23] C. Olston and M. Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010.

[24] J. Petke, M. B. Cohen, M. Harman, and S. Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Transactions on Software Engineering*, 41(9):901–924, Sept 2015.

[25] J. R. Quinlan. *C4.5: Programs for Machine Learning*, volume 1. Morgan kaufmann, 1993.

[26] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[27] J. Scambray, M. Shema, and C. Sima. *Hacking Exposed Web Applications: Web Application Security Secrets and Solutions*. McGraw-Hill San Francisco, 3rd edition, 2011.

[28] J. Slankas and L. Williams. Access control policy extraction from unconstrained natural language text. In *Social Computing (SocialCom), 2013 International Conference on*, pages 435–440, Sept 2013.

[29] P. Tonella and F. Ricca. Dynamic model extraction and statistical analysis of web applications: Follow-up after 6 years. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 3–10, Oct 2008.

[30] W3C. Hypertext transfer protocol – http/1.1, 1999.

[31] W3C. W3c xml schema definition language, 2012.

[32] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[33] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *ACM SIGSOFT FSE'12*, page 12. ACM, 2012.

[34] J. Yan and J. Zhang. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, volume 1, pages 385 –394, sept. 2006.

Gold Standard AC Policies

Gold standard policies are policies that the stakeholders want the SUTs to enforce. They are often documented in system requirements and/or specified by system administrators in the form of AC configurations. This appendix presents the gold standard policies that we extracted from the AC configurations and documentations of WordPress, TaskFreak, and iTrust.

Extracting GS policies is challenging, though. They might be embedded in textual descriptions of system requirements. They might be hard-coded in configuration files, for instance, iTrust defines access rights in it *web.xml* file (at run-time, these rights must be enforced). Information regarding GS policies can also be stored in database, e.g., role definitions, user and resource assignments. As a result, for each application, we need to go through these sources to extract GS policies.

Specifically, for WordPress, we use its role and capability specifications[14]. Table 9 details the roles and capabilities of WordPress. It associates each role to a set of capabilities. For instance, rule *Administrator* has all the defined capabilities, e.g., *create_users*, *edit_posts*, while *Subscriber* has only one *read* capability.

Table 9: WordPress roles and capabilities (excerpt). Labels: at a cell, 1 indicates the role on the same column is *allowed* to perform the operation on the same row; and 0 means the *denied*

Capability Role	**Administrator**	**Editor**	**Author**	**Contributor**	**Subscriber**
create_users	1	0	0	0	0
delete_users	1	0	0	0	0
edit_page	1	1	0	0	0
publish_page	1	1	0	0	0
publish_posts	1	1	1	0	0
edit_posts	1	1	1	1	0
read	1	1	1	1	1

For TaskFreak, we must extract its GS policies from its configuration file, i.e, $config.php$. The full AC specifications of TaskFreak are detailed in Tables 10 and 11. TaskFreak manages access policies based not only on roles but also on the position of

[14]https://codex.wordpress.org/Roles_and_Capabilities

users in assigned projects. It enforces access rights by combining these two sources of information.

Table 10: TaskFreak's role-based global AC specification. Labels: at a cell, 1 indicates the role on the same column is *allowed* to perform the operation on the same row; and 0 means the *denied*

		administrator	**manager**	**intern**	**guest**
Resource	Operation				
users	see all	1	1	1	0
	create new	1	1	0	0
	edit any	1	0	0	0
	delete any	1	0	0	0
projects	see all	1	0	0	0
	create new	1	1	0	0
	edit any	1	0	0	0
	delete any	1	0	0	0
	change status	1	1	0	0
tasks	create own tasks	1	1	1	0
	view internal tasks	1	1	1	0
	view public task	1	1	1	0
	edit any task	1	0	0	0
	system settings	1	0	0	0

For iTrust, we extract its GS policies from its web configuration file *web.xml*. They are detailed in Table 12. iTrust predefines eight roles, access permissions of roles to folders are specified in the table. The administrator can change such specification by editing the configuration file.

To this end, each of the subject applications has a specific way of specifying AC policies. We extract and use them as gold standards for identifying problematic inferred policies. Many of them have led to the detection of AC vulnerabilities.

Table 11: TaskFreak role-based project access rights

		leader	**moderator**	**member**	**official**	**extern**	**user not associated to project**
Resource	Operation						
comments	see all	1	1	1	1	1	0
	add new comment	1	1	1	1	0	0
	edit any	1	1	0	0	0	0
	delete any	1	1	0	0	0	0
tasks	create new	1	1	1	0	0	
	edit any	1	1	0	0	0	
	change status	1	1	0	0	0	
	delete any	1	1	0	0	0	
	view tasks	1	0	0	0	0	
project	manage	1	1	0	0	0	
	change status	1	1	0	0	0	
	edit info	1	0	0	0	0	
	delete	1	0	0	0	0	

Table 12: iTrust roles and access capabilities to specific role-based authorization folders. A cell labelled as 1 indicates the folder and files on the corresponding row can be accessed by the role on the corresponding column.

Folders&Pages / Role	**admin**	**er**	**hcp**	**lt**	**patient**	**pha**	**uap**	**tester**
/auth/*	1	1	1	1	1	1	1	1
/auth/hcp/*	0	0	1	0	0	0	0	0
/auth/hcp-uap/*	0	0	1	0	0	0	1	0
/auth/hcp-er/*	0	1	1	0	0	0	0	0
/auth/er/*	0	1	0	0	0	0	0	0
/auth/uap/*	0	0	0	0	0	0	1	0
/auth/admin/*	1	0	0	0	0	0	0	0
/auth/staff/*	1	0	1	1	0	0	1	0
/auth/patient/*	0	0	0	0	1	0	0	0
/auth/lt/*	0	0	0	1	0	0	0	0
/auth/tester/*	0	0	0	0	0	0	0	1
/util/getUser.jsp	1	1	1	1	1	1	1	1