EAI Endorsed Transactions
on Collaborative Computing

Research Article ICST.ORG

Achieving Security Assurance with Assertion-based Application Construction

Carlos E. Rubio-Medrano[1], Gail-Joon Ahn[1,*], Karsten Sohr[2]

[1]Arizona State University, 699 S. Mill Avenue, Tempe, Arizona, 85282, USA
[2]Universität Bremen, Am Fallturm 1, 28359 Bremen, Germany

Abstract

Modern software applications are commonly built by leveraging pre-fabricated modules, e.g. *application programming interfaces* (APIs), which are essential to implement the desired functionalities of software applications, helping reduce the overall development costs and time. When APIs deal with security-related functionality, it is critical to ensure they comply with their design requirements since otherwise unexpected flaws and vulnerabilities may consequently occur. Often, such APIs may lack sufficient specification details, or may implement a *semantically*-different version of a desired security model to enforce, thus possibly complicating the runtime enforcement of security properties and making it harder to minimize the existence of serious vulnerabilities. This paper proposes a novel approach to address such a critical challenge by leveraging the notion of software *assertions*. We focus on security requirements in role-based access control models and show how proper verification at the source-code level can be performed with our proposed approach as well as with automated *state-of-the-art* assertion-based techniques.

Received on 02 March 2015; accepted on 25 August 2015; published on 21 December 2015

Keywords: security assurance, software specification, software assertions, role-based access control, API, SDK

doi:10.4108/eai.21-12-2015.150819

1. Introduction

In recent years, there has been an increasing interest in using *heterogeneous* pre-fabricated software modules, e.g. *application programming interfaces* (APIs) and *software development kits* (SDKs), in order to not only reduce the overall development costs and time in producing high-quality applications, but also minimize the number of incorrect behaviors (*bugs*) observed in the final product. However, recent literature has shown that such modules often lack the proper specification details (in the form of formal or informal documentation) that are essential to guide how a module should be used correctly for implementing security-related functionality [2] [3]. Common pitfalls include missing code assumptions or prerequisites, as well as the lack of foundation on a standardized, well-defined security model that serves as a common reference to help developers understand and correctly implement security-related code. Such a problem may potentially become the source of serious security vulnerabilities, as developers may not be fully aware of the omissions and flaws they may introduce into their applications by failing to implement a security model in a proper way. In order to solve this problem, we propose an assertion-based approach to capture security requirements of security models and create well-defined representations of those requirements. This way, the security features could be effectively understood by all participants in the software development process, in such a way that they can leverage these features when implementing security-related functionalities for multi-module applications, at the same time they engage in a highly-collaborative environment. These *assertion-based* security specifications would be used in conjunction with existing *state-of-the-art* methodologies and tools to verify security properties at the source-code level. In this paper, we choose the well-known *role-based access control* (RBAC) [4] as the security model to enforce access control

★A preliminary version of this paper appeared in the Proceedings of the 2014 IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom) [1].
*Corresponding author. Email: gahn@asu.edu

requirements over an application that is in turn composed of several heterogeneous modules. Concretely, we show how the semantic variations, the lack of proper specification, and the absence of proper verification techniques can lead to the existence of non-trivial access control vulnerabilities in mission-critical applications such as banking applications. Moreover, we provide a well-defined description of RBAC based on the standard provided by the *American National Standards Institute* (ANSI) [5]. We model this reference description by using *assertions* which are later used to provide access control constraints. To inject assertions as part of the documentation devised for software modules, we also adopt *design by contract* (DBC) [6] paradigm. For such a purpose, we leverage the *Java Modeling Language* (JML) [7], a DBC-like specification language for Java, to serve as a vehicle for a*proof-of-concept* implementation of our approach. Also, we utilize existing tools to verify a set of security properties, thus providing a way to locate and possibly correct potential security vulnerabilities in software applications.

This paper is organized as follows: we start by providing some background on the topics addressed in this paper in Section 2. Next, we examine the general problem, as well as the problem instance addressed in this work in Section 3. We then present our approach in Section 4, and a case study depicting three Java-based software applications and our experimental results in Section 5. In Section 6, we provide some discussion on the benefits and shortcomings of our approach as well as some related work. Finally, Section 7 presents directives for our future developments and concludes the paper.

2. Background

Software assertions are commonly described as formal constraints intended to describe the *behavior* of a software system, e.g., what it is expected to do at runtime, and are commonly written as annotations in the system's source code [8]. Using assertions, developers can specify what conditions are expected to be valid before and after a certain portion of code gets executed, e.g. the range of values that the parameter of a given function is allowed to take. *Design by contract* (DBC) [6] is a software development methodology based on assertions and the assumption that the developers and the prospective users (clients) of a given software module establish a *contract* between each other in order for the module to be used correctly. Commonly, such a contract is defined in terms of assertions in the form of *pre* and *post* conditions, among other related constructs. Before using a DBC-based software module *M*, clients must make sure that *M*'s preconditions hold. In a similar fashion, developers must guarantee that *M*'s postconditions hold once it has finished

```
1  public interface Account{
2
3   //@ public instance model double balance;
4
5   //@ public invariant balance > 0.0;
6
7   /*@ public normal_behavior
8     @  requires amt > 0.0;
9     @  assignable balance;
10    @  ensures balance == (\old(balance) - amt);
11    @*/
12  public void withdraw(double amt)
13                       throws SecurityException;
14
15 }
```

Figure 1. An Excerpt of a JML-annotated Banking Application.

execution, assuming its corresponding preconditions were satisfied beforehand. The *Java Modeling Language* (JML) [7], is a *behavioral interface specification language* (BISL) for Java, with a rich support for DBC contracts. Using JML, the behavior of Java modules can be specified using pre- and postconditions, as well as class *invariants*, which are commonly expressed in the form of assertions, and are added to Java source code as the form of comment such as `//@` or `/*@...@*/`. Fig. 1 shows an excerpt of a Java interface named `Account`, which belongs to a banking application and has been annotated with JML specifications.

The contract for the `withdraw` method (shown in lines 7-11) makes use of the *model* field `balance`. In JML, it is possible to define model fields, methods and classes [9], which differ from their regular (*concrete*) counterparts in the sense they are used for specification purposes only, in an effort to better describe a given JML contract in a higher level of abstraction, without worrying about how it is implemented at the source code level. The model field `balance`, shown in line 3, is used to provide an abstract representation of the amount of money held by the bank account represented by interface `Account`. Following JML rules, a given class implementing interface `Account` will be required to provide a suitable implementation for it. The preconditions of method `withdraw` (defined by means of the `requires` keyword) require the value of the method parameter `amt` to be greater than zero, as shown in the assertion depicted in line 8. Postconditions for the same method, which are in turn defined by the `ensures` keyword, guarantee that the new value of model field `balance` will be equal to its previous value before the method was executed (as denoted by the `\old` keyword) minus the value of the method parameter `amt`. The set of memory locations, e.g. instance variables, that are allowed to be modified by the `withdraw` method is specified by means of the `assignable` clause (line 9). Line 5 depicts an assertion representing an *invariant*: the value of model field `balance` must be always greater than zero, before and after each method of interface `Account` executes. Finally, specification contracts that are expected to terminate *normally*, that is, without

diverging nor throwing exceptions at runtime, are defined by means of the `normal_behavior` keyword (line 7). Conversely, contracts that allow a method to throw a runtime exception are specified by means of the `exceptional_behavior` keyword. A summary of the JML features exercised in this paper can be found in [7] and [9].

In recent years, the *American National Institute of Standards* (ANSI) released a standard document that provides well-defined descriptions of the main components and functions that define RBAC [5], and it is mostly based on the well-known Z specification language [10]. In addition, a dedicated profile [11] has been introduced to provide support for expressing RBAC policies by taking both the aforementioned ANSI RBAC standard as a reference foundation as well as the well-known *eXtensible Access Control Markup Language* (XACML), which is a standard language for supporting the distributed definition, storage and enforcement of rich access control policies [12]. Fig. 2 shows an excerpt of an RBAC policy that has been written in the RBAC XACML profile: roles are encoded using so-called *role policy set* (RPS) files (Fig. 2 (a)), which include the name of the role (*teller*, lines 4-13) as well as a reference to a *permission policy set* (PPS) file that includes the set of access rights (permissions) authorized for such a role (lines 16-18), and is in turn shown in Fig. 2 (b). In the RBAC XACML profile, permissions are encoded as XACML rules and role hierarchies are established by allowing a PPS file *P* to reference other PPS files containing the permissions that are assigned to roles that happen to be *junior* to the roles whose RPS files reference *P*. For instance, Fig. 2 (b) (lines 16-18) references the PPS file defined for role *employee* (not shown in Fig. 2 (a)), which happens to be a junior role to *teller*.

3. Problem Description

As mentioned earlier, recent literature includes examples showing that *mission-critical* applications, e.g., banking mobile applications, have suffered from serious security vulnerabilities derived from an incorrect use of their supporting security APIs at the source-code level [2, 3]. Among the possible causes of this problem, insufficient software specifications, including the definition of prerequisites and hidden assumptions, as well as the existence of multiple *semantic* variations of a given security model, e.g., the lack of foundation on a standardized, well-defined model serving as a reference, are cited as common sources of incorrect implementations. Moreover, the problem gets aggravated by the lack of effective software verification procedures at the source-code level, which could affect the chances of identifying and potentially correcting security vulnerabilities exhibited by applications before deployment to a production system. In this paper, we address an instance of this problem by choosing RBAC as the security model to enforce access control requirements in a software application that is in turn composed of several modules. In addition, each of these modules may implement a different version of RBAC whose semantics may or may not strictly adhere to an existing RBAC reference model such as the ANSI RBAC [5]. We therefore aim to verify that such *heterogeneous* modules, when used to build a target application, correctly enforce a well-defined and consistent *high-level* RBAC policy, despite the differences they may exhibit with respect to their inner workings related to RBAC features, which could eventually result in security vulnerabilities.

As an illustrative example, Fig. 3 (a) and Fig. 3 (b) show a Java-based example where a high-level RBAC policy is enforced at runtime by placing authorization checks before performing security-sensitive operations. In both instances, a policy depicts a role *manager* as a senior role to *teller*, and allows for users, who are assigned to roles that happen to be senior to *manager*, to execute both the *transfer* and *withdraw* operations, whereas users holding *teller* role are allowed to execute the *withdraw* operation only, as shown in Fig. 2. Moreover, Fig. 3 (a) shows a Java class `BankAccount`, which implements the interface `Account` described in Fig. 1 and leverages the Spring Framework API [13] for implementing an authorization check (lines 7-16). Similarly, Fig. 3 (b) shows another class `DebitBankAccount` depicting an authorization check using the Apache Shiro API [14] (lines 7-11). In such a setting, it is desirable to evaluate the correct enforcement of the aforementioned RBAC policy as follows: first, the authorization checks depicted in both examples must correctly specify the roles that are allowed to execute each of the security-sensitive operations. For instance, the authorization check depicted in Fig. 3 (a) incorrectly allows for another role *agent* to also execute the `withdraw` method, which in turn represents a potential security vulnerability. Second, the role *hierarchy* depicted in the high-level policy must be correctly implemented at the source-code level by leveraging both APIs. As roles that happen to be senior to role *manager* should be allowed to execute both the `transfer` and `withdraw` methods, the role hierarchy must be correctly implemented by placing accurate authorization checks within the source code. In addition, the role hierarchy must be also defined correctly in the supporting API configuration files. as an incorrect implementation, e.g. missing role names within the XML files defined for the Spring API, may prevent users with the role *manager* from executing the `transfer` method. A more serious problem may be originated if users with the role *teller* are allowed to execute the `transfer` method. Finally, if users with the role *manager* are allowed

```
<PolicySet PolicySetId="RPS:teller:role" ...>
 <Target>
  <Subjects>
   <Subject>
    <SubjectMatch MatchId="...:string-equal">
     <AttributeValue DataType="...#string">
       teller
     </AttributeValue>
     <SubjectAttributeDesignator
        AttributeId="...:attributes:role"
        DataType="....#string"/>
    </SubjectMatch>
   </Subject>
  </Subjects>
 </Target>
<PolicySetIdReference>
  PPS:teller:role
</PolicySetIdReference>
</PolicySet>
```

(a) An excerpt of a RPS File.

```
<PolicySet PolicySetId="PPS:teller:role" ...>
<Policy PolicyId="Permissions:for:teller" ...>
<Rule RuleId="withdraw:permission" Effect="Permit">
 <Resource>
   <AttributeValue DataType="...#string">
     BankAccount
   </AttributeValue>
  </Resource>
  <Action>
    <AttributeValue DataType="...#string">
         public void withdraw(double amt)
    </AttributeValue>
  </Action>
</Rule>
</Policy>
<PolicySetIdReference>
 PPS:employee:role
</PolicySetIdReference>
</PolicySet>
```

(b) An Excerpt of a PPS File.

Figure 2. A Sample Policy Using the RBACXACMLProfile

```
import org.springframework.security.core.*;
public class BankAccount implements Account{

 public void withdraw(double amt)
                     throws SecurityException{

  Iterator iter = SecurityContextHolder
                  .getAuthorities().iterator();

  while(iter.hasNext()){
   GrantedAuthority auth = iter.next();
   if (!auth.getAuthority().equals("teller") ||
       !auth.getAuthority().equals("agent")){
    throw new SecurityException("Access Denied");
   }
  }
  this.balance -= amt;
 }
}
```

(a) Spring FrameworkAPI.

```
import org.apache.shiro.*;
public class DebitBankAccount{

 public void transfer(double amt, BankAccount acc)
                         throws SecurityException{

  if(!SecurityUtils.getSubject().hasRole("manager")){

   throw new SecurityException("Access Denied");

  }

  acc.withdraw(amt);
  this.balance += amt;

 }


}
```

(b) ApacheShiro API.

Figure 3. Enforcing an RBACPolicy by Leveraging *Heterogeneous* Security Modules.

to execute the `transfer` method, but are disallowed from executing the `withdraw` method (Fig. 3(b)) by incorrectly configuring the Spring API depicted in Fig. 3(a), a given object of class `DebitBankAccount` may be left in an *inconsistent* state, thus also creating a serious security problem.

4. Our Approach: Assertion-based Construction

In order to provide a solution to the problem described in Section 3, we propose an approach that combines the concepts of specification modeling and software assertions for describing security features at the source-code level. These so-called *assertion-based security models* are intended to provide compact, well-defined and consistent descriptions that may serve as a common reference for implementing security-related functionality. Our approach strives to fill in the gap between high-level descriptions of security features, which are mostly abstract and implementation-agnostic, and supporting descriptions focused at the source-code level, which are intended to cope with both security-related and behavioral-based specifications, such as the ones described in Section 2. As it will be described in Section 6, previous work has also explored the use of software assertions and DBC-like contracts for specifying access control policies. However, our approach is intended to leverage the *modeling* capabilities offered by software specification languages using a well-defined reference description of a security model as a source, in such a way that it not only allows for the correct communication, enforcement and verification of security-related functionality, but it also becomes independent of any supporting APIs used at the source-code level, thus potentially allowing for its deployment over applications composed of several heterogeneous modules. Fig. 4 depicts our

proposed approach: an assertion-based security model is intended to be enforced over a target application that is in turn composed of two modules leveraging security APIs and two modules whose security-related functionality has been implemented from scratch. This way, the semantic differences exhibited by such modules, as shown in Section 3, can be effectively mitigated. Moreover, by leveraging state-of-the-art methodologies based on assertions, effective automated verification of security properties at the source-code level becomes feasible, thus providing a means for discovering and possibly correcting potential security vulnerabilities.

To address the problem instance discussed in this paper, we leverage the JML *modeling* capabilities, e.g. model classes [9], to describe the ANSI RBAC standard described in Section 2. Later on, these model classes are used to create assertion-based constraints, which are in turn incorporated into the DBC contracts devised for each module in an application. This way, a high-level RBAC policy can be specified at the source-code level by translating it into assertion-based constraints included in DBC contracts. Following our running example, Fig. 5 shows an excerpt of a model class `JMLRBACRole`, which depicts the role component and some of its related functionalities as devised in the ANSI RBAC standard, e.g. role hierarchies. Such a model class is leveraged in Fig. 7 to augment the JML-based contract depicted in Fig. 1 with security-related assertions restricting the execution of the `withdraw` method to users who activate a role senior to *teller*. We start by defining a model variable `role`, of type `JMLRBACRole` (line 5), which is later used for defining access control constraints in the two specification cases depicted in Fig. 7: the first specification case, depicted in lines 9-14, allows one to properly execute the `withdraw` method, e.g. deducting from the balance of a given account, only if the object stored in the `role` variable represents a role senior to *teller*[1]. The second specification case, shown in lines 16-20, allows for the `withdraw` method to throw a runtime exception if the aforementioned constraint is found to be false. In addition, such a specification case also prevents any modification to the *state* (e.g. private fields) of a given object of type `BankAccount` from taking place.

Fig. 6 depicts our approach: a high-level RBAC policy, which is encoded by means of the dedicated RBAC profile provided by XACML [11], is translated into a series of DBC contracts. Later on, such contracts, along with the source code for a given software application, are fed into JML-based automated tools for verification purposes. Since such an application may be in turn

[1]Following the ANSI RBAC standard, a given role is always *senior* to itself.

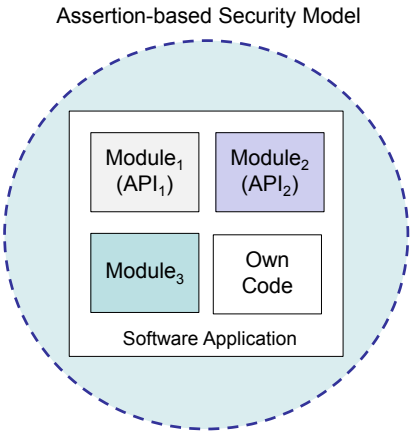

Figure 4. Deploying Assertion-based Security Models over a Multi-module Application.

```
1  package edu.asu.sefcom.ac.rbac;
2  public class JMLRBACRole
3                   extends JMLRBACAbstractRole{
4
5   public boolean isSeniorRoleOf(
6             JMLRBACAbstractRole role){
7
8    if(this.equals(role)){ return true; }
9
10   return getAllJuniorRoles().contains(role);
11  }
12 }
```

Figure 5. An Excerpt of a JML *Model* Class Depicting an ANSI RBAC *Role* Component.

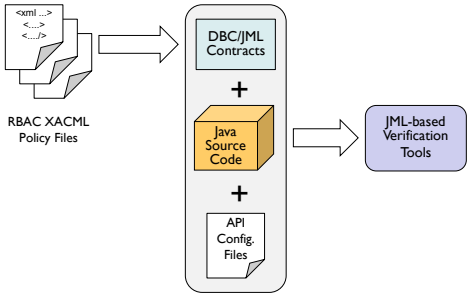

Figure 6. A Framework for Assertion-based Security Assurance.

composed of heterogeneous modules and each of them possibly represents a different API for implementing security-related functionality, e.g. enforcing an RBAC policy, the configuration files for such APIs must be also taken into account when leveraging automated tools for verification, as described in Section 3. In order to automate the creation of DBC contracts such as the ones depicted in Fig. 7, we designed an automated tool that translates RBAC policies encoded in the RBAC XACML profile into JML-based specifications, thus relieving policy designers and software architects from crafting

```
1  //@ model import edu.asu.sefcom.ac.rbac.*;
2  public interface Account{
3
4   //@ public instance model double balance;
5   //@ public instance model JMLRBACRole role;
6
7   //@ public invariant balance > 0.0;
8
9   /*@ public normal_behavior
10    @  requires amt > 0.0;
11    @  assignable balance;
12    @  ensures role.isSeniorRoleOf(
13    @  new JMLRBACRole("teller"))  ==>
14    @  (balance == \old(balance) - amt);
15    @ also
16    @ public exceptional_behavior
17    @  requires !role.isSeniorRoleOf(
18    @  new JMLRBACRole("teller"));
19    @  assignable \nothing;
20    @  signals_only SecurityException;
21    @*/
22   public void withdraw(double amt)
23                       throws SecurityException;
24
25 }
```

Figure 7. Enhancing a DBC contract with Access Control Assertions.

```
1  import org.springframework.security.core.*;
2  public class BankAccount implements Account{
3
4   //@ public represents role <- mapRole();
5
6   /*@ public pure model JMLRBACRole mapRole(){
7     @
8     @  JMLRBACRole newRole = new JMLRBACRole("");
9     @  RBACMonitor monitor  = new RBACMonitor();
10    @
11    @  Iterator iter = SecurityContextHolder
12    @                .getAuthorities().iterator();
13    @
14    @  while(iter.hasNext()){
15    @    GrantedAuthority auth = iter.next();
16    @    if (auth.getAuthority().equals("teller")){
17    @       newRole = new JMLRBACRole("teller");
18    @    }
19    @  }
20    @
21    @  return newRole;
22    @ }
23    @*/
24   ...
25 }
```

Figure 8. An Excerpt Showinga JML *Abstraction* Function.

such contracts manually and eliminating a potential source for errors.

Algorithm *P* shows a procedure for translating a set of XACML files into a set of data structures depicting an ANSI RBAC policy. The algorithm takes as an input the set of RPS and PPS XACML files as introduced in Section 2 and produces the set *R* of roles, the set *P* of permissions, the permission assignment $PA \subseteq R \times P$ relation involving the last two, and the $RH \subseteq R \times R$ relation depicting a role hierarchy between the roles included in *R*. The algorithm starts by initializing the sets/relations to be returned as a result as well as two auxiliary data structures: *REF* and *ROLE-DICT* (lines 1-3). Entries in the *REF* relation store the file references within PPS files that are used to establish a role hierarchy. As an example, an entry of the form (*PPS:teller:role*, *PPS:employee:role*) will be added to *REF* when the policy reference shown in Fig. 2 (b) (lines 16-18)) is processed. Conversely, the *ROLE-DICT* relation is introduced to keep a map between the names of the PPS files being referenced in the *REF* relation and the actual role names depicted in their corresponding RPS files. As an example, an entry of the form (*PPS:teller:role*, *teller*) will added to *ROLE-DICT* when processing the RPS and PPS files belonging to the *teller* role depicted in Fig. 2. The first phase of Algorithm *P* continues by retrieving the pair of RPS and PPS files depicting both the role declaration as well as the set of permissions that are assigned to such role. First, the RPS file is retrieved and the name for a role *r* is extracted. Such a name is then used to populate the *R* set as well as to introduce an initial entry of the form (*r*, *r*) to the *RH* relation indicating that each role is always *senior* or *junior* to itself (lines 5-7). Next, the PPS file corresponding to role *r* is retrieved, an entry to the *ROLE-DICT* relation is added, and each of the permissions included in such PPS file is parsed to populate the *PA* relation (lines 8-14). The first phase ends by processing each of the file references included in the PPS file and adding corresponding entries into the *REF* relation as discussed before. The second phase of Algorithm *P* (lines 18-20) focuses on *expanding* the *RH* relation by adding an entry for each pair of roles in *R* that are in a senior-junior role relationship. We model such calculation as a *graph reachability* problem assuming *RH* to be a *directed* graph. With this in mind, implement a *depth-first search* (DFS) algorithm over all roles in *R*: each entry in the *REF* relation is retrieved, the role name corresponding to the file acting as the senior role is obtained from *ROLE-DICT* and the auxiliary Algorithm *expandRH* is called. Such a algorithm takes an initial role *r* as an input and populates the *RH* relation by recursively obtaining all entries in *REF* and *ROLE-DICT* that belong to roles that happen to be junior to a given role *r*. The runtime performance of the first phase of Algoritm *P* can be regarded to be $O(|RPS| + |PPS|) \approx O(|R| + |P|)$ in the best case, which occurs when every permission in *P* is assigned to only one role in *R*. When several permissions in *P* are assigned to several different roles in *R* then the performance turns out to be $O(|R| + |R| * |P|) \approx O(|R| * |P|)$. In a similar fashion, performance of the second phase can be analyzed as follows: since the DFS algorithm is known to run in $O(E)$ for a graph having *V* nodes and *E* edges, our implementation may then run on $O(V * E) \approx O(|R| * |P|)$ in the best case when $V = R$ and $E = P$, which occurs when every permission in *P* is assigned to only

one role in R. In the case when the same permission is assigned to different roles in R, the running time may be regarded as $O(V * E) \approx O(|R| * (|R| * |P|))$ for $V = R$ and $E = R \times P$.

Taking as an input the data structures produced by Algorithm P, Algorithm $\overline{\sigma}$ produces DBC contracts written in a subset of the JML syntax defined in [15], like the one shown in Fig. 7, by leveraging a *template* in the form of an *abstract syntax tree* (AST), which is shown in Fig. 9. The algorithm starts by exploring the PA relation to obtain the entry depicting the *junior-most* role being assigned to every permission in P (lines 2-6). For such a purpose, line 4 of Algorithm $\overline{\sigma}$ queries the RH structure to determine if there exists a *seniority* relationship between two nodes r_i and r_j ($i \neq j$) in R. Such queries can be potentially answered in constant time ($O(1)$), as it suffices to locate the entry (r_i, r_j) in RH, due to the *expansion* procedure conducted by the second phase of Algorithm P. Finally, the algorithm produces a DBC/JML contract for each of the *junior-most* entries obtained in the previous step (lines 7-13). An alternative approach would include eluding the aforementioned *expansion* procedure carried on by Algorithm P and leaving any further algorithms, e.g., Algorithm $\overline{\sigma}$, with the duty of determining if a role happens to be senior to another one. Such an alternative approach may be beneficial in the case when only a few permissions in P happen to be assigned to more than one role in R, in such a way that the seniority relation between those roles may need to be determined when calculating the *junior-most* role only for such few permissions. In Algorithm $\overline{\sigma}$, since we potentially explore the entries in the PA relation twice, and such a relation may be of size $|R| * |P|$ in the worst case, the runtime performance can be regarded as $O(2 * |R| * |P|) \approx O(|R| * |P|)$. We present an analysis of the correctness of our approach in Appendix A.

As described in Section 1, we aim to support the verification of security properties in mission-critical applications. For such a purpose, we leverage an approach based on automated *unit testing* [16] by adopting JET [16]: a dedicated tool tailored for providing automated runtime testing of Java modules with JML-based assertions, e.g. classes. Using JET, testers can verify the correctness of a Java module by checking the implementation of each method against their corresponding JML specifications. In addition, we also support the detection of potential security vulnerabilities by means of static techniques by leveraging the ESC/Java2 tool [7], which is based on a theorem prover and internally builds *verification conditions* (VCs) from the source code being analyzed, and its corresponding JML-based specifications, which the theorem prover then attempts to prove, thus allowing for the automated analysis of whole code modules without running the applications. In particular, ESC/Java2 uses *modular reasoning* [17], which is regarded as an effective technique when used in combination with static checking since code sections can be analyzed and their JML-based specifications can

Algorithm *P* : Parsing RBAC XACML Files.

Data: Sets *RPS* and *PPS* of RBAC XACML files
Result: Sets *R* of roles, *P* of permissions, and the *PA* and *RH* relations

```
1  Initialize R and P to empty sets;
2  Initialize PA and RH to empty relations;
3  Initialize ROLE-DICT and REF to empty relations;
4  for each file rps in RPS do
5      r = Get role name from rps;
6      R = R ∪ r;
7      RH = RH ∪ (r,r);
8      ref-pps = Get name of permissions file
       referenced by r;
9      ROLE-DICT = ROLE-DICT ∪ (ref-pps, r);
10     pps = Get file from PPS using ref-pps;
11     for each permission p in pps do
12         if p ∉ P then
13             P = P ∪ p;
14         PA = PA ∪ (r, p);
15     JUNIOR-PPS = Get names of files referenced by
       pps;
16     for each junior-pps-name in JUNIOR-PPS do
17         REF = REF ∪ (ref-pps, junior-pps-name);
18 for each (senior-ref, junior-ref) in REF do
19     (senior-ref, role) = Get from ROLE-DICT using
       senior-ref;
20     RH = expandRH(role, senior-ref, RH, REF,
       ROLE-DICT);
21 return R, P, PA and RH;
```

Algorithm *expandRH* : Constructing the *RH* of an RBAC Policy.

Data: A role $r \in R$, a String *key* depicting a file name, the *RH*, *REF* and *ROLE-DICT* relations
Result: The *RH* relation

```
1  Initialize JM and C to empty sets;
2  ENTRIES = Get entries from REF using key;
3  for each (senior-ref, junior-ref) in ENTRIES do
4      (junior-ref, junior-role) = Get from ROLE-DICT;
5      if (role, junior-role) ∉ RH then
6          RH = RH ∪ (role, junior-role);
7      RH = expandRH(role, junior-ref, RH, REF,
       ROLE-DICT);
8  return RH;
```

Algorithm $\overline{\mathcal{C}}$: Transforming an RBAC Policy to DBC/JML Contracts.

Data: The *PA* and *RH* relations depicting an ANSI RBAC Policy
Result: A Set *C* of DBC/JML Contracts

```
Initialize JM and C to empty sets;
for each (r,p) in PA do
    (r',p) = Get entry from JM using p;
    if (r', p) ≠ null and (r',r) ∈ RH then
        JM = JM \ (r'p) ;
    JM = JM ∪ (r,p);
for each (r,p) in JM do
    Create signature from p;
    Get contract from C using signature;
    if contract is null then
        contract = Create using AST and signature;
        C = C ∪ contract;
    Add r to roles in contract;
return C;
```

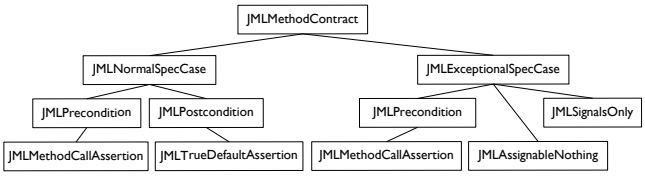

Figure 9. A Sample AST template for Producing JML Syntax.

be proved by inspecting the specification contracts of the methods they call within their method *bodies*.

Later, in Section 5, we present our findings on leveraging both techniques in a set of case studies depicting mission-critical Java applications. In order to support the verification process just described, proper constructs are needed to map the modeling features included in DBC contracts (as depicted in Fig. 7) and the implementation source code of each heterogeneous module. For such a purpose, we leverage the features offered by the JML *abstraction* functions [9], which allow for JML model features to be properly *mapped* to source-code level constructs, thus providing a way to verify that each heterogeneous module implements a given high-level policy correctly. As an example, Fig. 8 shows an excerpt where a JML model method is used to map the source code implementing security features as provided by the Spring Framework API with the model features depicted in Fig. 7.

In general, the correct enforcement of a security model may involve the following cases: first, a high-level security policy, which is based on a well-defined security model definition, should be correctly defined and all policy conflicts must have been resolved, e.g.

Table 1. Distribution of Responsibilities for Enforcing an Assertion-based Security Model In a Collaborative Setting.

Actor	Description of Tasks
Security Domain Experts	Develop an assertion-based security model by using a precise definition as a reference, e.g. using the ANSI RBAC standard. (See Fig. 5).
Policy Administrators	Instantiate the security model to be enforced, e.g. specification of an RBAC policy based on the ANSI RBAC standard. (See Fig. 2).
Software Architects	Incorporate the security policy into DBC constructs by specifying assertion-based constraints (See Fig. 7).
Code Developers	Correctly implement the DBC specifications define by software architects (including security checks). Provide a mapping between the security model and the security APIs used for implementation purposes (See Fig. 8).
Code Testers	Verify both the functional and the security related aspects of a given software application based on their DBC specification (See Section 5).

evaluating a given RBAC policy by using techniques such as the ones discussed in [18]. Second, access to all protected resources within a given application, e.g. the `withdraw` operation depicted in Fig. 7, is *guarded* by an authorization check (adhering to the well-known *principle of complete mediation*). Following our example, authorization checks should depict the RBAC constructs defined in the overall policy, e.g. checking for the correct roles and/or permissions before executing any sensitive operation. Third, supporting components for the security model features are implemented correctly, e.g. RBAC role hierarchies. Finally, we also require that the detection of runtime policy violations is implemented properly, e.g. exception handling and data consistency. With this in mind, for the problem instance addressed in this paper, we make the following assumptions: first, the ANSI RBAC model is well-understood by all participants in the software development process, e.g. policy designers, software architects and developers. Second, the assertion-based specification of the security model is correct: in other words, it has been verified beforehand. Third, any supporting RBAC modules, including security APIs and SDKs, have been implemented correctly, even though their semantics with respect to RBAC may differ, as addressed in Section 3.

Finally, our approach is intended to be carried out by the different participants in the software development process, in such a way that the process of constructing vulnerability-free software becomes a collaborative

responsibility shared by all involved actors, obviously including the source-code level developers. Table 1 shows a summary of the tasks devised for each participant.

5. Case Study

In order to provide a *proof-of-concept* implementation of our approach, we developed a reference description of the security model under study by using a set of JML model classes based on the case illustrated in Fig. 5. Such a reference model contains 960 lines of code grouped in 17 Java classes, including 1,383 lines of JML specifications depicting the functionality desired for RBAC as described in the ANSI RBAC standard. For our case study, we leveraged a pair of open-source Java applications: OSCAR EMR [19], which is a rich web-based software platform tailored for handling *electronic medical records* (EMR). It consists of approximately 35,000 lines of code organized into 110 classes and 35 packages. In addition, we also leveraged JMoney [20], a financial application consisting of 7,500 lines of code grouped into 45 classes. Moreover, we developed a banking application depicting the running examples shown in this paper. Such an application leverages the Apache Shiro and Spring Framework Security APIs, as well as our own RBAC monitor developed for implementing security-related functionality. It consists of 36 classes and contains 1,550 lines of code as well as 1,450 lines of JML specifications, which utilize our JML model classes in DBC contracts, as shown in Fig. 7.

In addition, we performed an evaluation over the automated translation tool described in Section 4 that takes as an input a set of XACML files depicting an ANSI RBAC policy and produces a set of DBC contracts in the JML language. Such a tool consists of 6,246 lines of code grouped in 25 classes in Java, and implements the Algorithms labeled as P, $\overline{v}$ and *expandRH*, also shown in Section 4 as well as the AST structure shown in Fig. 9. In order to evaluate the effectiveness of the tool we designed an experiment tailored to measure the overall processing time in milliseconds taken by the tool to process XACML-based policies and produce their corresponding DBC/JML contracts. In such experiments, we varied the policy *size* by varying the number of roles included in the policy as well as the number of permissions being assigned to each role. In addition, we controlled the number of different role hierarchies depicted by each policy as well as the number of permissions that were simultaneously assigned to the same role. Fig. 10 shows the results of our experimental approach when allowing the tool to process synthetically-created policies to produce DBC/JML contracts such as the one shown in Fig. 7. We produced 4 different policies varying the number of roles from 5 to 20, as well as the number of permissions

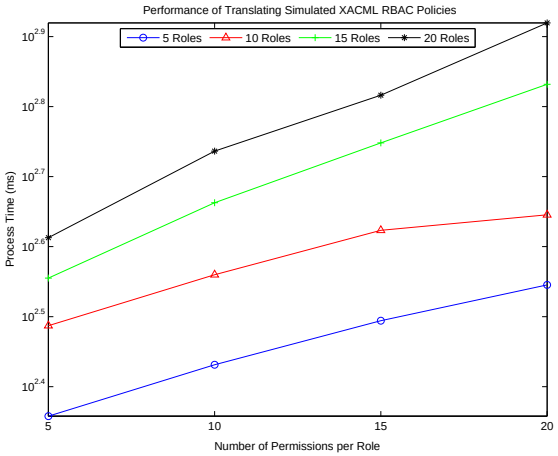

Figure 10. Runtime performance of our Translation Tool.

included on each role. All roles in the policy belonged to the same role hierarchy and the same permission was allowed to be assigned to at most two roles. As expected, the execution time depicted by our tool remains linear as the size of the policy given as an input is varied.

As described in Section 4, our approach is intended to identify inconsistencies in the implementation of security models that can eventually become security vulnerabilities. In the context of the security model addressed in this paper, an incorrect implementation of an RBAC policy may end up introducing non-trivial vulnerabilities to applications. Based on the description of RBAC depicted by the ANSI RBAC standard, inconsistencies on the implementation of RBAC policies can be described as follows: first, an incorrect mapping between access rights (permissions) and sensitive operations performed by applications. Sensitive operations should be properly *guarded* by permissions, in such a way that the execution of such operations is only allowed when the requesting entity is found to be granted the permissions devised for them. Failing to identify such sensitive operations as well as the need to secure them by properly requesting for the permissions, may also result in non-trivial security vulnerabilities. Following our running example, the `transfer` operation featured in Fig. 3 (b) must be identified as security-sensitive and being guarded by a permission to be assigned only to users holding the *manager* role. Second, failures in the assignment of permissions to roles may exist. As an example, incorrect assignment of a given permission P to a role R may allow R and roles that happen to be senior to it to execute the unintended operations *guarded* by P. On the other hand, unintended removal of P from the list of permissions devised for R will deprive such a role and all other roles senior to it from exercising P and its related operations, possibly causing an *availability* problem by restricting the number of operations that such roles can execute in the context

of a given application. Third, there may be failures in the implementation of role hierarchies. As an example, the introduction or removal of a role or a set of roles at a given *level* of the hierarchy may produce vulnerabilities: introduction of an unintended role R in a given hierarchy may allow for R to unintentionally *inherit* the permissions assigned to all roles that happen to be junior to R, and it may also allow for senior roles to R to obtaining the permissions assigned to R. Conversely, removal of a role R in a given hierarchy may deprive roles senior to R from obtaining the permissions assigned to it, which can yield vulnerabilities such as the state inconsistency problem described at the end of Section 3. In the context of applications composed of heterogeneous modules, the aforementioned inconsistencies can be potentially introduced either in the source code of the application itself, or by incorrect configuration of policy files. As an example, failures in the implementation of authorization checks, as the ones depicted in Fig. 3, can be regarded as a common source of potential inconsistencies at the source code level. In addition, state-of-the-art security APIs, as the ones depicted throughout this paper, leverage text files for configuring security features. With respect to RBAC, our featured APIs provide configuration files depicting the assignment of permissions to roles. In summary, an incorrect configuration of those files may also introduce security vulnerabilities.

With this in mind, we modeled implementation inconsistencies of the RBAC security model by leveraging an approach inspired by *mutation testing* [21]: we inserted variations (also known as *mutants*) in both the source code and the API configuration files of the applications considered in our study. As an example, Fig. 11 shows different mutants introduced to the RBAC policy shown in Table 2: first, the original policy is modified to add an unintended permission (*transfer*, (*t*)) to a role *employee* (Fig. 11 (a)). Such a modification creates a potential security vulnerability as it allows *employee*, and all other roles senior to it, e.g. *agent* and *teller*, to execute an operation that was originally intended only for a role *manager*. A configuration file depicting such modification is shown in Fig. 12 (lines 14, 19, 23). Similarly, Fig. 11 (b) shows a permission (*deposit*, (*d*)) being removed from the *employee* role. Such a modification produces an inconvenience to such a role and all other roles that happen to be senior to it, as execution of the deposit operation will be denied at runtime. Fig. 11 (c) shows another example where the original role hierarchy of the RBAC policy is modified to introduce an unintended role (*supervisor*, (*S*)). This way, the newly-introduced role creates a pair of security vulnerabilities: first, it inherits the permissions from all junior roles in the hierarchy, thus allowing for the execution of unintended operations. Fig. 13 shows an example depicting such modification (lines 6,7). Second, it also allows for a senior role in the hierarchy to obtain an extra permission (*audit*, (*a*)), thus possibly allowing them to perform unintended operations as well. Fig. 13 shows an excerpt of an XML configuration file depicting the role hierarchy modification shown in Fig. 11 (c) (lines 6-8). Finally, Fig. 11 (d) shows a case when a role is removed from a role hierarchy: *teller* is left aside by removing the relationships with both the *manager* (senior) and the *employee* (junior) roles. It exposes an inappropriate permission revocation not only to users holding the role *teller* as it is prevented from getting the permissions of its junior roles (e.g. *deposit*, (*d*)), but also to senior roles since revocation prevented them from getting the permissions assigned to *teller* (e.g., *withdraw*, (*w*)) including all other permissions that could be obtained from junior roles to *teller*.

Table 2. A Sample RBAC Policy for Evaluation Purposes.

Role	Junior Roles	Allowed Operations
Employee	-	`deposit`
Teller	Employee	`withdraw, deposit`
Agent	Employee	`close, deposit`
Manager	Teller, Agent	`transfer, withdraw, deposit, close`

In the rest of this section, we describe the experimental procedure we have conducted on the sampled software applications by leveraging existing assertion-based tools to detect (*kill*) mutant-based inconsistencies in the implementation of RBAC policies like the ones described above, in an effort to show the suitability of our approach for the effective verification of security properties. In particular, we present the results when applying both the dynamic and the static approach to the aforementioned case-study applications. Later, in Section 6, we highlight some shortcomings we have identified in this experimental process, and propose some alternative solutions.

5.1. Applying Dynamic Analysis Techniques to Assertion-based Verificatio

As described in previous sections, we aim to support the verification of security properties by leveraging an approach based on automated *unit* testing [16] as well as the JML specifications depicting our assertion-based models that were introduced in Section 4. For such a purpose, we adopted JET [16], which is a dedicated tool tailored for providing automated runtime testing of Java modules with JML-based assertions, e.g. classes. Using JET, testers can verify the correctness of a Java module by checking the implementation of each method against their corresponding JML specifications. As an example, the contract of a given method M is used as a *test oracle*, by first translating it into

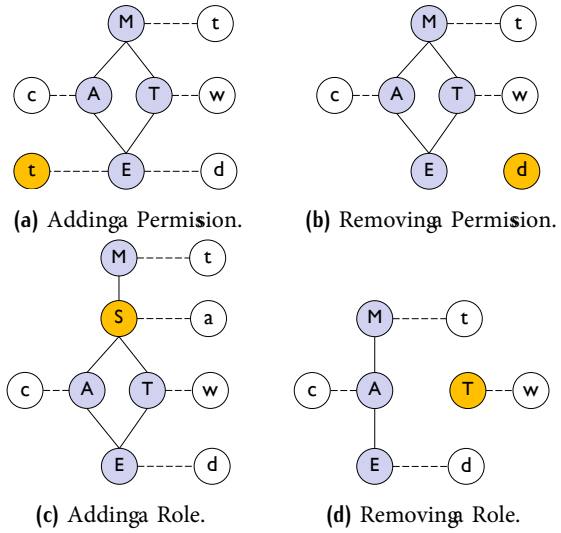

Figure 11. Introducing *Mutants* in the RBAC Policy shown in Table 2: roles are labeled using their uppercase initial. Permisions are shown using lowercase initials and dotted lines. E.g., in Fig. 11 (a), role *Manager* is shown as a colored circle labeled as *M*. Permision *transfer* is shown as a white circle labeled as *t*. The assignment of permision *transfer* to the role *Manager* is shown as a dotted line.

runtime assertion checking (RAC) code. Then, proper values (of either primitive or reference data types) are *randomly* created for each of *M*'s formal parameters, and compared for compliance against the RAC code created for *M*'s precondition. If such a precondition is satisfied, a *valid* test case is said to be created [2], and *M*'s body is executed. If any exception not devised for *M* is thrown, the test case is regarded as *failed*. Otherwise, the RAC code for *M*'s postcondition is executed. If such a postcondition is satisfied, the test is regarded as a *success*, otherwise, it is regarded as *failed* as well. Many different test cases can be created for *M*, as soon as different combinations of values for *M*'s parameters are created by JET.

Following the automated testing approach just described, we conducted a set of experiments to measure the effectiveness of our assertion-based models, along with our enhanced DBC contracts, in detecting the mutations introduced into the applications tested in our case study. Such experiments were carried out on a PC equipped with an Intel Core Duo CPU running at 3.00 GHZ, with 4 GB of RAM, running Microsoft Windows 7 64-Bit Enterprise Edition. First, we measured the impact of our approach in the average execution time of the applications. In order to provide a mapping between the modeling features included in JML contracts (as depicted in Fig. 7) and the implementation code of each heterogeneous module, we leveraged the

[2] Otherwise, the test case is said to be *meaningless*, so it is discarded.

```
1  public static Ini getStaticIni(){
2   Ini ini = new Ini();
3   ini.addSection("roles");
4   ini.setSectionProperty("roles",
5                          "manager",
6                          "p:deposit, "  +
7                          "p:withdraw, " +
8                          "p:close, " +
9                          "p:transfer" );
10  ini.setSectionProperty("roles",
11                         "agent",
12                         "p:close, " +
13                         "p:deposit" +
14                         "p:transfer");
15  ini.setSectionProperty("roles",
16                         "teller",
17                         "p:deposit, " +
18                         "p:withdraw" +
19                         "p:transfer");
20  ini.setSectionProperty("roles",
21                         "employee",
22                         "p:deposit " +
23                         "p:transfer");
24  return ini;
25  }
```

Figure 12. Introducing *mutants* in an Apache Shiro configuration.

```
1  <?xml ...>
2   <beans:bean id="roleHierarchy" ...>
3    <beans:property name="hierarchy">
4     <beans:value>
5       manager    > supervisor
6       supervisor > teller
7       supervisor > agent
8       teller     > employee
9       agent      > employee
10    </beans:value>
11  ...
```

Figure 13. Introducing *Mutants* in Spring Framework.

features offered by the JML *abstraction* functions [9]: we enhanced our supporting tool described in Section 4 to also produce abstraction functions for the referred Spring Framework and Apache Shiro APIs. We then executed a sample trace of the Java methods exposed by our three applications and calculated the average execution time over 1,000 repetitions. Such a trace was created to contain representative operations for each application, e.g. the trace created for the OSCAR EMR application that contains Java methods used to update a patient's personal data as well as information about medical appointments and prescriptions.

As shown in Table 3, the introduction of RAC code has a moderate impact on the performance, which is mostly due to the overhead introduced by the RAC code generated to process both the JML contracts as well as the abstraction functions. We then recorded the results obtained by our tool while attempting to detect (*kill*) the mutants introduced in both the configuration of the Security APIs as well as the authorization checks guarding each of the Java methods contained in our sample traces, following the approach depicted in Fig. 11. Table 3 shows a report on the number of generated test cases, including the number of

Table 3. Experimental Data Using JET.

	Banking	JMoney	OSCAR
Total methods	46	136	125
Analysis time per method/s	4.56	17.32	15.4
Total analysis time/s	209.76	2355	1925
Runtime overhead/s	0.97	2.34	1.78
Generated test cases	1000	1000	1000
Meaningful test cases	150	250	225

Table 4. Experimental Data Using ESC/Java2.

	Banking	JMoney	OSCAR
Analysis time per method/s	0.43	2.07	0.5
Total analysis time/s	19.66	281.41	63.00

meaningful ones produced by the tool. [3] Our *meaningful* test cases were able to *kill* all the mutants inserted into our case study applications.

In an additional experiment, we compared the time taken by our JML model classes to detect each of the mutant generation techniques depicted in Fig. 11. Once again, we used a trace of Java methods depicting the main functionality for each application, and used the automated mutant-generation tool described before to generate different variations to an original RBAC policy. The results, as shown in Fig. 14, show that adding/removing a role to a given hierarchy is the most costly mutation to be detected by the RAC code through processing our assertion-based JML classes. This is mostly due to the way how role hierarchies are implemented in our JML classes, by using a series of `java.util.ArrayList` objects to store references to each senior/junior role in a given hierarchy, and allowing for such references to be inspected recursively when determining if there is a seniority relationship between two given roles.

5.2. Applying Static Analysis Techniques to Assertion-based Verificatio

As mentioned in previous sections, we also leverage the ESC/Java2 tool for providing verification guarantees based on static analysis techniques and our proposed approach. However, despite the support provided for JML-based constructs by such a tool, some challenges must be addressed: first, in order to prove the correctness of a certain source code *C* against its corresponding JML contracts, the tool additionally requires that the JML specifications of

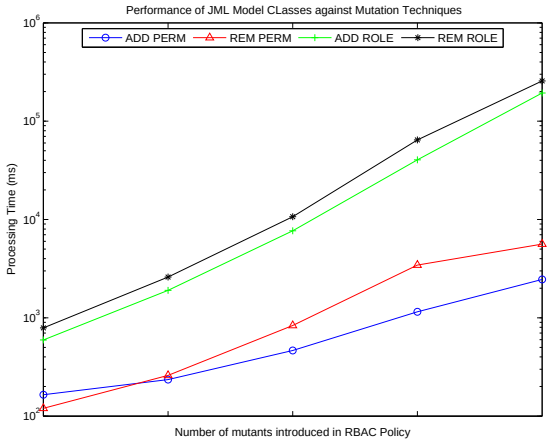

Figure 14. Runtime performance of our Verificatio Approach.

each library called within *C* are available, including the specifications of additional libraries the original ones may eventually call later on. In some cases, such a requirement may notoriously increase the amount of VCs that need to be proved by the tool, so the verification process becomes prohibitively expensive, resulting in the *specification-creep* problem [17]. Second, an additional problem arises from the lack of support offered by the current tool for advanced JML concepts, such as the JML model classes introduced in Section 4 and the JML abstraction functions depicted in in Fig. 8, as the internally-produced VCs are too complex for the tool to handle, which limits the applicability of our assertion-based models.

Subsequently, we present an approach that addresses these challenges while still providing verification guarantees for our assertion-based approach. First, we addressed the specification-creep problem. In particular, as described in Section 4, we assumed the Security APIs leveraged within our case study have been implemented correctly and previously verified elsewhere. Therefore, there is no need to include their corresponding source code in our verification process. Based on this observation, we provided *specification stubs* for the leveraged Security APIs whose JML-based annotations are trivially satisfied. Fig. 15 shows the translated JML specifications for the method `hasRole` of class `Subject`, which implements an authorization check in the Apache Shiro API, as shown in Fig. 3 (b). One can see that a trivial method body has been provided; for the task of static checking a Shiro module only the contract and not the specification of `hasRole` is needed by ESC/Java2. The process of providing specification stubs can be carried out by security domain experts (see Table 1) for the Security APIs and must only be revised when new API versions are released.

Second, as mentioned before, the JML model classes, which are a core part of the approach shown in

[3] In JET, a test case *T* for a given method *M* is said to be *meaningful* if the tool is able to randomly create values for *M*'s formal parameters in such a way that *M*'s preconditions involving such parameters are satisfied. Otherwise *T* is said to be *meaningless*.

```
1  public class Subject{
2
3  /*@ public normal_behavior
4    @  requires  true;
5    @  ensures \result == true || \result == false;
6    @ also
7    @ public exceptional_behavior
8    @  requires  false;
9    @  assignable \nothing;
10   @  ensures true;
11   @*/
12   public /*@ pure @*/ boolean hasRole(String r){
13     return true;
14   }
15 }
```

Figure 15. Specification Stubs for the ApacheShiro API.

```
1  public interface Account{
2
3   /*@ public normal_behavior
4     @  requires amt > 0.0;
5     @  assignable balance;
6     @  ensures
7     @   (SecurityUtils.getSubject()
8     @                     .hasRole("teller") ||
9     @   SecurityUtils.getSubject()
10    @                     .hasRole("manager"))
11    @   ==> ...
12    @*/
13 public void withdraw(double amt)
14                      throws SecurityException;
15 }
```

Figure 16. Translating Model JML Classes.

Section 4, are beyond the current capabilities of ESC/-Java2. To overcome this limitation, we provided JML specifications that do not employ the JML model classes and use low-level JML concepts instead. Table 5 shows the implementation-independent model classes and their corresponding low-level specifications for each framework. As an example, the aforementioned `hasRole` and `getAuthority` methods are directly called rather than using JML model classes: the role hierarchy depicted in Table 2 and Fig. 7, which checks that the current user is granted a role senior to *teller* (e.g. *manager*), can be translated into the JML contracts shown in Fig. 16 (lines 7-10): the references to the model class `JMLRBACRole` have been substituted for the `hasRole` method of class `Subject` provided by the Apache Shiro API, and are integrated together by using the operator `||` in JML, applied to all relevant senior roles (e.g., the *manager* role in line 10). We call this technique *unrolling* the role hierarchy. It is also supported by the JML-based translation tool described in Section 4 by automatically translating XACML policies to JML specifications suitable for ESC/Java2 and inserting them into the corresponding source code. This step relieves the software architect (see Table 1) from manually providing JML contracts, which is an error-prone process. Software architects can now leverage an automated tool for this purpose.

In addition, web-based Java software frameworks, e.g. Spring Framework, provide support for declarative access control in addition to programmatic access control (e.g. by using authorization checks). Declarative authorization allows a developer to define role-based restrictions on access to certain protected resources such as a given Java method (see also Section 3). As an example, in the Spring Framework we can define access restrictions in an XML configuration file as follows:

```
<sec:protect method="BankAccount.withdraw"
             access="teller"/>
```

In the dynamic analysis approach described before, such declarative rules are implicitly considered because the Spring framework enforces this role assignment under the hood. However, in a static approach, we must also include such configuration files to obtain a complete picture about the access control features implemented in the web application. Otherwise, we would produce false positives because ESC/Java2 would falsely report that an authorization check is missing although it has been defined in the XML configuration file (and not in the code). To implement this additional feature, we parse the XML configuration files to retrieve the access control statements and insert their corresponding JML `assume` statements in the body of the referenced Java methods—the JML `assume` statement lets ESC/Java2 unconditionally assume a constraint without checking it [17]. For instance, the XML configuration shown above can be translated into the following specification that is to be inserted in the implementation body of the `withdraw` method of class `BankAccount` (Fig. 8):

```
//@ assume GrantedAuthority.getAuthorities()
//@                         .contains("teller");
```

After the preparing steps, we applied our analysis technique to the applications under our case study, by following the mutation-based approach described before. All mutants were automatically detected by ESC/Java2 even if they were hidden within the many methods of the real-world case studies JMoney (125 methods) and OSCAR EMR (136 methods). As an example, in the following authorization check included in the JMoney application

```
if(!currentUser.hasRole("accountant"))
   throw new SecurityException("Permission denied!");
```

a user with the *acountant* role is permitted to execute the method. If the user, however, has assumed a role senior to *accountant* (e.g. *owner*), a security exception is thrown, since the Apache Shiro library call `hasRole` provides no native support for implementing role hierarchies, which must be in turn encoded into a series of nested `hasRole` calls listing all the roles that are authorized to execute a given method.

We used a conventional Lenovo Thinkpad T510 laptop (Intel Core i7-620M Processor, 2.66GHz, 8 GB

Table 5. Mapping between the JML RBAC and Security-API-based specifications

	`JMLRBACRole.equals(new JMLRBACRole(r))`	`JMLRBACRole.isSeniorOf(new JMLRBACRole(r))`
Spring Framework	`GrantedAuthority.getAuthorities().contains(r)`	`GrantedAuthority.getAuthorities().contains(r) ||...|| GrantedAuthority.getAuthorities().contains(mostSeniorRole)`
Apache Shiro	`Subject.hasRole(r)`	`Subject.hasRole(r) ||...|| Subject.hasRole(mostSeniorRole)`
RBAC Monitor	`RBACMonitor.hasRole(r)`	`RBACMonitor.hasRole(r) ||...|| RBACMonitor.hasRole(mostSeniorRole)`

RAM) for our experiments with the static analysis technique, in an effort to provide increased RAM capabilities to the theorem prover serving as a back-end for ESC/Java2. The runtime of the three applications under our case study is given in Table 4, which confirmed that the preparation steps enabled us to use ESC/Java2 efficiently. Specifically, we avoided the expensive analysis of container classes, e.g. `java.util.Collection`, by applying the aforementioned *specification stub* technique. For example, the method `getAuthorities()-.contains()` uses a container class of the Java type `Collection` and by leveraging the stub technique we succeeded in eliminating this problem.

6. Discussion and Related Work

In order for the approach presented in Section 4 to properly detect implementation flaws at the source-code level, a correct and sound translation from such a model into our assertion-based constructs is needed. With this in mind, a formal proof must include the following: first, only the access rights depicted in the original policy must be present in their JML-based counterpart, that is, no potential security vulnerabilities are introduced by adding extra access rights in the resulting specifications. Second, all access rights included in the original policy must be present in the translated specifications, that is, no *inconveniences*, e.g. preventing a legitimate access from taking place, are introduced by missing access rights included in the original policy. We present an sketch of such a proof in Appendix A.

The experimental results depicted in Section 5.1 and 5.2 support our claim that our approach can effectively expose the set of security vulnerabilities caused by the incorrect source-code level implementations of security models. In our approach, we have selected Java for our *proof-of-concept* implementation due to its extensive use in practice. Moreover, we have also chosen JML as the specification language for defining our assertion-based security models due to its enhanced tool support as well as its language design paradigm, which supports rich behavioral specifications. At the same time it strives to handle the complexity of using complex specification constructs, in such a way that it becomes suitable for average developers to use [7] (see Table 1).

We believe that our approach can be extended to other programming languages/development platforms. For instance, Spec# [22] provides rich DBC-based specifications for the C# language, depicting an approach similar to JML. Moreover, our approach can be also applied to other Java-based frameworks such as JEE [23] or Android [24], which may help implement authorization checks for guarding access to its core system services.

Despite our success, some issues still remain in the verification process. In particular, ESC/Java2 may produce *false positives* (in case the built-in theorem prover cannot prove a VC) and *false negatives* (e.g., restrictions on loop unrolling). To deal with this situation, a possible solution may consider a runtime testing approach, like the one we have described using the JET tool, for all methods raising warnings by ESC/-Java2, thus showing a way in which both techniques can be used to provide stronger guarantees for the verification. Second, as shown in Table 3, the number of *meaningful* test cases produced by the JET tool is considerably less than the number of test cases created, which may affect the test coverage provided by the tool and could allow for potential security vulnerabilities to remain hidden during the verification process. This is mostly due to the limitations on the automated testing technique [16]. A possible solution would adopt a static approach for those methods whose test coverage is found to be below a given threshold. Finally, we have found that extended static checking is valuable when analyzing applications with respect to checking the implementation of an assertion-based access control model. In particular, we supported different concepts depicted in our model, which are in turn based on the ANSI RBAC standard, such as role hierarchies.

Furthermore, extended static checking represents a promising approach as there is no need to provide dedicated test cases nor implement a complete *running* system, as software modules can be tested in isolation by using modular reasoning techniques. Although ESC/Java2 is quite mature as a research prototype, some shortcomings still exist with respect to its current tools and development kits. As there are currently ongoing efforts for building a new extended static checker for Java within the OpenJML initiative[4], we hope this approach can be applied to larger case studies in industrial contexts, supporting advanced

[4] `http://jmlspecs.sourceforge.net/`

JML concepts, such as *model* features, as well as complex data structures. This newer extended static checker is expected to leverage more powerful backend SMT solvers such as Yices [25]. At the time of this writing, however, this tool does not completely support advanced JML specifications, which we use in our analysis and which are well-supported by ESC/-Java2. For example, heavyweight JML specifications, i.e. specifiations that contain normal and exceptional behavior, are not correctly implemented by OpenJML's extended static checker as our early experiments have shown. However, we use heavyweight specifications as the basis for our assertations (for example see Fig. 7).

Our work is related to other efforts in software security: Architectural risk analysis [26] attempts to identify security flaws on the level of the software architecture and hence is unrelated to the source-code level addressed in this approach. Language-based security approaches in the sense of Jif [27] allow software to be verified against information flow policies rather than supporting specific security requirements for different Security APIs. In addition, formal verification of RBAC properties has been already discussed in the literature [18]. These approaches are mostly focused on verifying the correctness of RBAC models without addressing their corresponding verification against an implementation at the source-code level.

The work closely related to ours involves the use of DBC, which was explored by Dragoni, et al. [28]. In addition, Belhaouari et al. introduced an approach to the verification of RBAC properties based on DBC [29]. Both approaches, while using DBC for checking RBAC properties, do not include the use of reference models to better aid the specification of DBC constraints in the security context. Moreover, no support is provided as API-independent constructs, such as the JML model capabilities discussed in our approach.

Other works that apply a DBC approach based on JML in the security context are presented by Lloyd et al. (biometric authentication system) [30], Cataño et al. [31] (smart card system), and Mustafa et al. [32] (Android system services). These works, however, do not cover applications consisting of heterogeneous modules and do not use the combination of dynamic and static analysis technique for assertion-based verification.

7. Conclusions and Future Work

In this paper, we have addressed the problem originated by the existence of security vulnerabilities in software applications. We have shown how such vulnerabilities, which may exist due to the lack of proper specification and verification of security checks at the source-code level, can be tackled by using well-defined reference models with the help of software assertions, thus providing a reference for the correct enforcement of security properties over applications composed of heterogeneous modules such as APIs and SDKs. Future work would include the introduction of assertion-based models to better accommodate other relevant security paradigms, e.g., the correct usage of cryptography APIs. We also plan to refine our proposed RBAC model introduced in Section 4 by introducing an automated translation from the specifications depicted in the ANSI RBAC standard, which are written in the Z specification language, to our supporting assertion-based language JML. In addition, we plan to refine the translation Algorithms shown in Section 4, in such a way their runtime efficiency can be considerably improved.

Appendix A. Analyzing the Correctness of our Proposed Approach

In this appendix, we present an analysis of the correctness of our approach as presented in Section 4. Recall from Fig. 4, our approach is based on the translation of ANSI RBAC security policies expressed in the well-known XACML policy language into DBC contracts written in the JML specification language. Then, such contracts, along with the source code of the software modules and any other supporting configuration files are fed into JML-based tools for automated verification. With this in mind, an analysis for correctness may need to take into account the following: first, the correct implementation of our proposed JML-based Model Classes, which depict the ANSI RBAC standard, must be verified. As mentioned in Section 2, such a standard contains functional specifications written in the Z language that unambiguously describe the inner components of RBAC as well as the interactions between them. We have provided a manual translation of those Z-based specifications into the JML language, in such a way the implementation of our referred model classes can be guided by them. Since we have proposed in this paper to use of JML-tools for verification purposes, a natural step will include to use such tools for verifying the correct implementation of our model classes, in an approach similar to the one we have described in Section 5. We plan to carry on such process as a part of future work, which may also focus on providing an automated translation of Z-based specifications into JML, in such a way any errors or redundancies introduced by our manual translation effort can be detected and resolved. Second, the correlation between our approach, the ANSI RBAC standard, and the semantics of the JML language needs to be explored. Concretely, a rigorous analysis involving the semantics of DBC contracts written in JML must be carried on to guarantee that a given RBAC policy is correctly enforced at runtime by a software

module. For such a purpose, the work of Bruns [33] may serve as a reference for a formal description of the semantics of the JML language. We plan to work on such challenging endeavor in the future as well. Finally, in the remainder of this section, we focus on showing the correctness the translation of an ANSI RBAC policy into DBC/JML contracts. As mentioned before, our approach is mostly concerned with verifying that a given policy P is correctly enforced by a set of heterogeneous modules that are used to build up a software application S. More concretely, our approach must guarantee that every role in P can potentially *exercise* in S only the permissions that were originally intended in the aforementioned policy P. With this in mind, we base our correctness claims by showing that the set of permissions that a given role in P can exercise at runtime, namely *effective* runtime permissions, are the same in both the original policy P as well as in the produced DBC/JML contracts that are later used for software verification as described in this paper.

A.1. Basics

As described in Section 4, an ANSI RBAC XACML Policy can be parsed into the following: a set R of roles, a set P of permissions, the permission assignment relation ($PA \subseteq$ R $\times$ P) and the role hierarchy $RH \subseteq R \times R$ where $(r_i, r_j) \in RH$ if and only if r_i is senior to role r_j. In addition, a role r_i is always senior to itself, e.g., $(r_i, r_i) \in RH$. For simplicity, and without loss of generality, let us assume only a single permission exists for executing each method in a given Java module. As an example, given our sample policy depicted in Table 2, we have: R = {Manager, Agent, Teller, Employee}, P = {Transfer, Withdraw, Deposit, Close}, PA = {(Manager, Transfer), (Manager, Withdraw), (Teller, Withdraw), (Agent, Close), (Employee, Deposit)}, RH = {(Manager, Manager), (Agent, Agent,) (Teller, Teller), (Employee, Employee), (Manager, Agent), (Manager, Teller), (Manager, Employee), (Agent, Employee), (Teller, Employee)}. Finally, the set of DBC/JML contracts can be modeled as a relation $C \subseteq R \times M$ when R is the set of roles as described before and M is the set of Java methods included in a given application being the subject of a verification process. For each role name enlisted in a given contract, an entry in C is produced. As an example, the contract shown in Fig. 7 can be modeled as an entry of the form (Teller, *withdraw(double)*).

A.2. AuxiliaryAlgorithms

In order to support our analysis, we introduce two auxiliary algorithms: first, Algorithm $\mathfrak{R}$ takes a set of DBC/JML contracts and produces the PA' relation obtained from the roles and permissions enlisted in the contracts provided as an input, thus potentially *reversing* the transformation procedure carried on by our proposed Algorithm $\overline{\mathfrak{C}}$. Second, Algorithm EP calculates the set of effective runtime permissions for a given role r in the set of roles R belonging to an ANSI RBAC policy whose PA and RH relations are provided as an input. Recall that following the ANSI RBAC standard, the set of effective permissions that are available to a given role r are those defined in the PA relation of the policy plus all other permissions that are also assigned to roles that happen to be *junior* to it. As an example, executing Algorithm EP on the role *Manager* defined in the policy described in Table 2 will lead to the following permissions: {Transfer, Withdraw, Deposit, Close}.

Algorithm $\mathfrak{R}$: Reconstructing an ANSI RBAC policy from DBC/JML Contracts.

Data: A Set C of DBC/JML Contracts
Result: the PA' relation depicting an ANSI RBAC Policy

```
1 Initialize PA' to an empty relation;
2 for each (r, m) ∈ C do
3     p = Get permission from m;
4     if (r, p) ∉ PA' then
5         PA' = PA' ∪ (r, p);
6 return PA';
```

Algorithm *EP* : Obtaining the set of effective runtime permissions of a role in an ANSI RBAC Policy.

Data: A role r ∈ R, the PA and RH relations
Result: the set EP of effective runtime permissions

```
1 Initialize EP to the empty set;
2 for each (r', p) ∈ PA do
3     if (r, r') ∈ RH then
4         EP = EP ∪ p;
5 return EP;
```

A.3. Correctnes Analysis

We start our analysis by showing that the set of produced DBC/JML contracts contains no extra permissions other than the ones defined in the original ANSI RBAC XACML policy. This way, we guarantee that no security *vulnerabilities* are introduced by our translation procedure into the generated set of DBC/JML contracts by allowing a given role to execute at runtime a permission not intended in the original policy. We formalize such requirement in the following:

Lemma 1. All effective runtime permissions present in the produced DBC/JML contracts are included in the original ANSI RBAC XACML policy. Formally, given an

XACML RBAC policy encoded as (R, P, PA, RH), $\forall r \in R$, $\nexists\, p \in P$ s.t. $p \notin EP\ (r, PA, RH) \wedge p \in EP\ (r, \Re\ (\mathcal{T}\ (PA, RH)), RH)$.

Proof. Let us assume $\exists\, p \in P$ s.t. $p \notin EP\ (r, PA, RH) \wedge p \in EP\ (r, \Re\ (\mathcal{T}\ (PA, RH)), RH)$ for some $r \in R$. In order to have $p \in EP\ (r, \Re\ (\mathcal{T}\ (PA, RH)), RH)$, following Algorithm EP (lines 2-4), there must be $(r', m) \in C$ for some $(r, r') \in RH$ and $m = p$ (Algorithm $\Re$, lines 2-5). Moreover, since $(r', m) \in C$, then, following Algorithm $\mathcal{T}$ (lines 7-13), there must be $(r', p) \in JM$ such that $m = p$. If $(r', p) \in JM$, then $(r', p) \in PA$ since $JM \subseteq PA$, following Algorithm $\mathcal{T}$ (lines 2-6). If $(r', p) \in PA$ and $(r, r') \in RH$, then $p \in EP\ (r, PA, RH)$, which contradicts our assumption that $p \notin EP\ (r, PA, RH)$. □

A.4. Soundness Analysis

In addition, we must also show that all the permissions included in the original ANSI RBAC policy are also included in the set of DBC/JML contracts. This way, we also guarantee that no security *vulnerabilities* are introduced into the produced contracts by missing to include one or more permissions included in the original policy. As described at the end of Section 3, failing to execute a permission originally included in a given policy may be the source of non-trivial vulnerabilities by leaving applications in an *inconsistent* state. We formalize this requirement as follows:

Lemma 2. All effective permissions included in the original RBAC XACML policy are included in the produced DBC/JML contracts. Formally, given an XACML RBAC policy encoded as (R, P, PA, RH), $\forall r \in R$, $\nexists\, p \in P$ s.t. $p \in EP\ (r, PA, RH) \wedge p \notin EP\ (r, \Re\ (\mathcal{T}\ (PA, RH)), RH)$.

Proof. Let us assume $\exists\, p \in P$ s.t. $p \in EP\ (r, PA, RH) \wedge p \notin EP\ (r, \Re\ (\mathcal{T}\ (PA, RH)), RH)$ for some $r \in R$. If $p \in EP\ (r, PA, RH)$ then $(r', p) \in PA$ for some $(r, r') \in RH$ following Algorithm EP lines 2-4. If $(r', p) \in PA$, then $(r', p) \in JM$ following Algorithm $\mathcal{T}$ (lines 2-6). Moreover, since $(r', p) \in JM$, then, following Algorithm $\mathcal{T}$ (lines 9-11), there must be $(r', m) \in S$ such that $m = p$. Since $(r', m) \in S$, then $(r', p) \in PA'$ following Algorithm $\Re$ (lines 2-5). Subsequently, if $(r', p) \in PA'$, then $p \in EP\ (r, \Re\ (\mathcal{T}\ (PA, RH)), RH)$ following Algorithm EP (lines 3-4). This contradicts our assumption that $p \notin EP\ (r, \Re\ (\mathcal{T}\ (PA, RH)), RH)$. □

A.5. Final Remarks

Finally, following the topics discussed in the beginning of this Appendix, we formalize the correctness claims of our translation approach by means of the following:

Theorem 1. The set of effective runtime permissions of each role listed the original XACML RBAC policy and the set of effective runtime permissions from the same role obtained from the translated DBC/JML contracts are the same. Formally, given an XACML RBAC policy encoded as (R, P, PA, RH), $\forall r \in R$, $EP\ (r, PA, RH) \equiv EP\ (r, \Re\ (\mathcal{T}\ (PA, RH)), RH)$.

Proof. The theorem follows from Lemma 1 and Lemma 2, as those two cases are sufficient to show that the set of effective runtime permissions from the original ANSI RBAC XACML policy and the ones from the DBC/JML are the same. □

Acknowledgement. The work of Carlos Rubio-Medrano and Gail-Joon Ahn was partially supported by a grant from the US Department of Energy (DE-SC0004308). The work of Karsten Sohr was supported by the German Federal Ministry of Education and Research (BMBF) under the grant 16KIS0074.

References

[1] Rubio-Medrano, C.E. and Ahn, G.J. (2014) Achieving security assurance with assertion-based application construction. In *Proc. of the 10th Int'l Conf. on Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom)* (IEEE): 520–530.

[2] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D. and Shmatikov, V. (2012) The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proc. of the ACM Conf. on Computer and comm. security*: 38–49.

[3] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B. and Smith, M. (2012) Why eve and mallory love Android: an analysis of Android SSL (in)security. In *Proc. of the ACM Conf. on Computer and communications security*: 50–61.

[4] Sandhu, R.S., Coyne, E.J., Feinstein, H.L. and Youman, C.E. (1996) Role-Based Access Control Models. *IEEE Computer* **29**(2): 38–47.

[5] American National Standards Institute Inc. (2004), Role Based Access Control. ANSI-INCITS 359-2004.

[6] Hoare, C.A.R. (1969) An axiomatic basis for computer programming. *Communications of the ACM* **12**(10): 576–580.

[7] Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K. *et al.* (2003) An overview of JML tools and applications. In *Proc. 8th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*: 73–89.

[8] Rosenblum, D.S. (1995) A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.* **21**(1): 19–31.

[9] Cheon, Y., Leavens, G., Sitaraman, M. and Edwards, S. (2005) Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.* **35**(6): 583–599.

[10] Spivey, J.M. (1989) *The Z notation: a reference manual* (Upper Saddle River, USA: Prentice-Hall, Inc.).

[11] OASIS (2014), XACML v3.0 Core and Hierarchical Role Based Access Control (RBAC) Profile Version

1.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cd-03-en.html.

[12] OASIS (2014), eXtensible Access Control Markup Language (XACML) TC. https://www.oasis-open.org/committees/xacml/.

[13] Pivotal, Inc. (2013), Spring security 3.1.2. http://static.springsource.org/spring-security/site/index.html.

[14] The Apache Software Foundation (2013), Apache shiro 1.2.1. http://shiro.apache.org/.

[15] G. T. Leavens and E. Poll and C. Clifton and Y. Cheon and C. Ruby and D. Cok and J. Kiniry (2004), JML Reference Manual. http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html.

[16] Cheon, Y. (2007) Automated random testing to detect specification-code inconsistencies. In *Proc. of the 2007 Int'l Conf. on Software Engineering Theory and Practice* (Orlando, Florida, U.S.A.).

[17] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B. and Stata, R. (2002) Extended static checking for Java. In *Proc. of the ACM SIGPLAN Conf. on Prog. language design and implementation*: 234–245.

[18] Hu, H. and Ahn, G.J. (2008) Enabling verification and conformance testing for access control model. In *Proc. of the 13th ACM Symp. on Access Control Models and Technologies*: 195–204.

[19] OSCAR EMR (2014), OSCAR Electronic Medical Records System. http://oscar-emr.com/.

[20] J. Gyger and N.L Westbury (2014), JMoney Financial System. http://jmoney.sourceforge.net/.

[21] Jia, Y. and Harman, M. (2011) An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* **37**(5): 649 –678.

[22] Barnett, M., Leino, R. and Schulte, W. (2005) The spec# programming system: An overview. In *Proc. of the 2004 Int'l Conf. on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (Berlin: Springer-Verlag): 49–69.

[23] Oracle Inc. (2014), Java Platform Enterprise Edition. http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[24] Google Inc. (2014), Android. http://www.android.com.

[25] Dutertre, B. and de Moura, L. (2006) A fast linear-arithmetic solver for dpll(t). In *Proc. of the 18th Int'l Conf. on Computer Aided Verification*, CAV'06 (Berlin: Springer): 81–94.

[26] McGraw, G. (2006) *Software Security: Building Security In* (Addison-Wesley).

[27] Sabelfeld, A. and Myers, A.C. (2003) Language-based information-flow security. *IEEE J. Selected Areas in Communications* **21**(1): 5–19.

[28] Dragoni, N., Massacci, F., Naliuka, K. and Siahaan, I. (2007) Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Public Key Infrastructure* (Springer Berlin), *LNCS* **4582**, 297–312.

[29] Belhaouari, H., Konopacki, P., Laleau, R. and Frappier, M. (2012) A design by contract approach to verify access control policies. In *17th Int'l Conf. on Engineering of Complex Computer Systems (ICECCS)*: 263 –272.

[30] Lloyd, J. and Jürjens, J. (2009) Security analysis of a biometric authentication system using UMLsec and JML. In *MoDELS* (Springer), *Lecture Notes in Computer Science* **5795**: 77–91.

[31] Cataño, N. and Huisman, M. (2002) Formal specification of Gemplus's electronic purse case study. In *FME 2002* (Springer), **LNCS 2391**: 272–289.

[32] Mustafa, T. and Sohr, K. (2014) Understanding the implemented access control policy of android system services with slicing and extended static checking. *International Journal of Information Security* : 1–20doi:10.1007/s10207-014-0260-y, URL http://dx.doi.org/10.1007/s10207-014-0260-y.

[33] Bruns, D. (2010) Formal semantics for the java modeling language. In *Informatiktage* (GI), *LNI* **S-9**: 15–18.