Achieving Security Assurance with Assertion-based Application Construction

Carlos E. Rubio-Medrano and Gail-Joon Ahn
Ira A. Fulton Schools of Engineering
Arizona State University
Tempe, Arizona, USA, 85282
{crubiome, gahn}@asu.edu

Karsten Sohr
Center for Computing Technologies (TZI)
Universität Bremen
28359 Bremen, Germany
sohr@tzi.de

***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 be consequently occurred. 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.**

I. Introduction

In recent years, there has been an increasing interest in leveraging *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 specification) that are essential to guide how a given module can be used correctly for implementing security-related functionality [1] [2]. 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 so that they can leverage these features when implementing security-related functionalities for multi-module applications while being engaged in a highly-collaborative environment at the same time. 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) [3] as security model to enforce access control requirements over an application that is in turn composed of several heterogeneous modules. 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 in Section II. Next, we examine the general problem, as well as the problem instance discussed in this paper in Section III. We then present our approach in Section IV, and a case study depicting three Java-based software applications and an experimental process in Section V. In Section VI, we provide some discussion on the benefits and observed shortcomings of our approach as well as some related work. Finally, Section VII presents directives for our future work and concludes the paper.

II. Background

Software assertions are commonly described as formal constraints intended to describe what a software system is expected to do at runtime, and are commonly written as annotations in the system's source code [4]. 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 expected range of values intended for the parameter of a given function. *Design by contract* (DBC) [5] 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 execution, assuming its corresponding preconditions were satisfied beforehand. The *Java Modeling Language* (JML) [6], 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 post conditions, 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

COLLABORATECOM 2014, October 22-25, Miami, United States

DOI 10.4108/icst.collaboratecom.2014.257691

```
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 }
```

Fig. 1: An Excerpt of a JML-annotated Banking Application.

named `Account`, which belongs to a banking application and has been annotated with JML specifications. A summary of the JML features exercised in this paper can be found in [6] and [7].

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 [8], and it is mostly based on the well-known Z specification language [9]. In addition, a dedicated profile 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 and storage & enforcement of rich access control policies [10], [11].

III. 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 [1], [2]. 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 deploying in 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. Each of them possibly implements a different version of RBAC whose semantics may or may not strictly adhere to an existing RBAC reference model such as the one described in [8]. 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.

Fig. 2 (a) and Fig. 2 (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. Fig. 2 (a) shows a Java class `BankAccount`, which leverages the Spring Framework API [12] for implementing an authorization check (lines 7-16). Similarly, Fig. 2 (b) shows another class `DebitBankAccount` depicting an authorization check using the Apache Shiro API [13] (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. 2 (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 to execute the `transfer` method, but are disallowed from executing the `withdraw` method (Fig. 2 (b)) by incorrectly configuring the Spring API depicted in Fig. 2 (a), a given object of class `DebitBankAccount` may be left in an *inconsistent* state, thus also creating a serious security problem.

IV. Our Approach: Assertion-based Application Construction

In order to provide a solution to the problem described in Section III, 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. As it will be described in Section VI, 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,

```
1  import org.springframework.security.core.*;
2  public class BankAccount implements Account{
3
4   public void withdraw(double amt)
5                        throws SecurityException{
6
7    Iterator iter = SecurityContextHolder
8                    .getAuthorities().iterator();
9
10   while(iter.hasNext()){
11    GrantedAuthority auth = iter.next();
12    if (!auth.getAuthority().equals("teller") ||
13        !auth.getAuthority().equals("agent")){
14     throw new SecurityException("Access Denied");
15    }
16   }
17   this.balance -= amt;
18  }
19 }
```

(a) Spring Framework API.

```
1  import org.apache.shiro.*;
2  public class DebitBankAccount{
3
4   public void transfer(double amt, BankAccount acc)
5                        throws SecurityException{
6
7    if(!SecurityUtils.getSubject().hasRole("manager")){
8
9     throw new SecurityException("Access Denied");
10
11   }
12
13   acc.withdraw(amt);
14   this.balance += amt;
15
16  }
17
18
19 }
```

(b) Apache Shiro API.

Fig. 2: Enforcing an RBAC Policy by Leveraging *Heterogeneous* Security Modules.

in such a way 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 as shown in Fig. 3: 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 III, 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 [7], to describe the ANSI RBAC standard described in Section II. 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. 4 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. 5 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. 5: 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

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

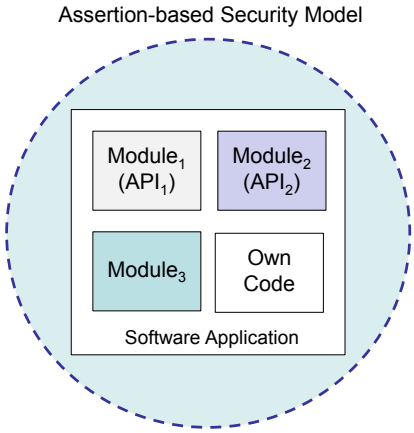

Fig. 3: Deploying Assertion-based Security Models over a Multi-module Application.

```
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 }
```

Fig. 4: An Excerpt of a JML *Model* Class Depicting an ANSI RBAC *Role* Component.

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. 7 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

```
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 }
```

Fig. 5: 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 }
```

Fig. 6: An Excerpt Showing a JML *Abstraction* Function.

for a given software application, are fed into JML-based automated tools for verification purposes. Since such an application may be in turn 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 III. In order to automate the creation of DBC contracts such as the ones depicted in Fig. 5, 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 such contracts manually and eliminating a potential source for errors.

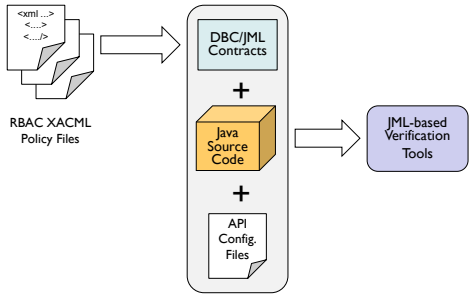

Fig. 7: A Framework for Assertion-based Security Assurance.

As described in Section I, we aim to provide the verification of security properties by leveraging an approach based on automated *unit* testing [14] as well as the JML specifications depicting the assertion-based models described above. For such a purpose, we adopt JET [14], 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. In addition, we also aim to provide support for finding possible security vulnerabilities by means of static techniques. For such a purpose, we leverage the ESC/Java2 tool [6], 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* [15], 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 be proved by inspecting the specification contracts of the methods they call within their method *bodies*. Later, in Section V, 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. 5) and the implementation source code of each heterogeneous module. For such a purpose, we leverage the features offered by the JML *abstraction* functions [7], 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. 6 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. 5.

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. evaluating a given RBAC policy by using techniques such as the ones discussed in [16]. Second, access to all protected resources within a given application, e.g. the `withdraw` operation depicted in Fig. 5, is *guarded* by an

TABLE I: 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. 4).
Security Policy Administrators	Instantiate the security model to be enforced, e.g. specification of an RBAC policy based on the ANSI RBAC standard.
Software Architects	Incorporate the security policy into DBC constructs by specifying assertion-based constraints (See Fig. 5).
Code Developers	Correctly implement the DBC specifications defined by software architects (including security checks). Provide a mapping between the security model and the security APIs used for implementation purposes (See Fig. 6).
Code Testers	Verify both the functional and the security related aspects of a given software application based on their DBC specifications (See Section V).

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 is 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 III.

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 I shows a summary of the tasks devised for each participant.

V. 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. 4. 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 [17], which is a rich web-based software platform tailored for handling *electronic health records* (EMR). It consists of approximately 35,000 lines of code organized into 110 classes and 35 packages. In addition, we also leveraged JMoney [18], a financial application consisting of 7,500 lines of code grouped into 45 classes. Finally, 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. 5.

TABLE II: A Sample RBAC Policy for Evaluation Purposes.

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

In order to verify the effectiveness of our approach for detecting faulty implementations of the RBAC security model, we followed an approach inspired in *mutation testing* [19]: we inserted variations (also known as *mutants*) in both the source code and the API configuration files of the applications considered in our study, in an effort to introduce inconsistencies in the implementation of their corresponding RBAC Policies. As an example, Fig. 8 shows different mutants introduced to the RBAC policy shown in Table II: first, the original policy is modified to add an unintended permission (*transfer*, (*t*)) to a role *employee* (Fig. 8 (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*. Similarly, Fig. 8 (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. 8 (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. 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. 9 shows an excerpt of an XML configuration file depicting the role hierarchy modification shown in Fig. 8 (c) (lines 6-8). Finally, Fig. 8 (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 expose an inappropriate permission revocation to not only users holding the role *teller*, as such a role is prevented from getting the permissions of its junior roles (e.g. *deposit*, (*d*)), but also senior roles since it prevented from getting the permissions assigned to *teller* (e.g., *withdraw*, (*w*)) including all other permissions that could be obtained from junior roles to *teller*.

A. Assertion-based Verification

Following the automated testing approach described in Section IV, 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. As described in [14], the JML-based specifications depicting our model classes are translated into *runtime assertion checking* (RAC) code, which is then executed along with the original application code for verification purposes. In order to provide a mapping between the modeling features included in JML contracts (as depicted in Fig. 5) and the implementation code of each heterogeneous module, we leveraged the features offered by the JML *abstraction* functions [7]: we enhanced our supporting tool described in Section IV 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 patient's personal data as well as information about medical appointments and prescriptions. As shown in Table III, 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. 8. Table III shows a report on the number of generated test cases, including the number of *meaningful* ones produced by the tool. [2] 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. 8. 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. 12, 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.

[2]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 M's preconditions involving such parameters are satisfied. Otherwise T is said to be *meaningless*.

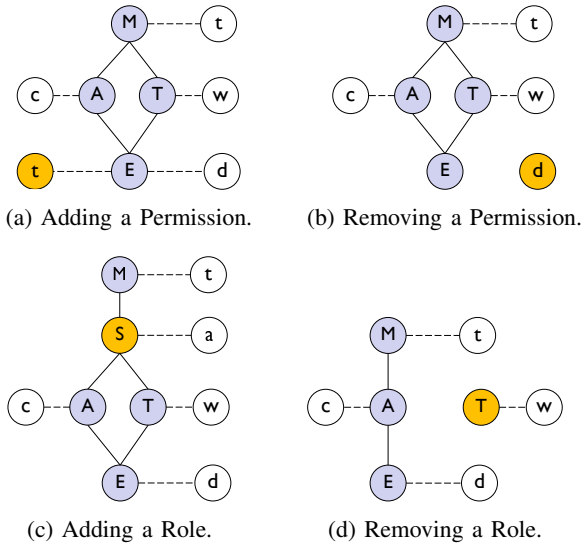

Fig. 8: Introducing *Mutants* in an RBAC Policy.

```
<?xml ...>
...
 <beans:bean id="roleHierarchy" ...>
  <beans:property name="hierarchy">
   <beans:value>
     manager    > supervisor
     supervisor > teller
     supervisor > agent
     teller     > employee
     agent      > employee
   </beans:value>
  </beans:property>
 </beans:bean>
...
```

Fig. 9: Introducing *Mutants* in Spring Framework.

TABLE III: Experimental Data on Using JET and ESC/Java2.

	Banking	**JMoney**	**OSCAR**
Total methods	46	136	125
JET			
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
ESC/Java2			
Analysis time per method /s	0.43	2.07	0.5
Total analysis time /s	19.66	281.41	63.00

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 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

```
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   @*/
11   public /*@ pure @*/ boolean hasRole(String r){
12     return true;
13   }
14 }
```

Fig. 10: Specifications Stubs for the Apache Shiro 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    @*/
14 public void withdraw(double amt)
15                       throws SecurityException;
16 }
```

Fig. 11: Translating Model JML Classes.

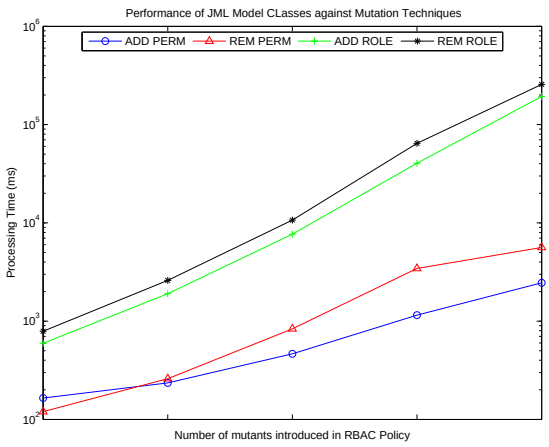

Fig. 12: Runtime performance of a Dynamic Verification Approach.

that need to be proved by the tool, so the verification process becomes prohibitively expensive, resulting in the *specification creep* problem [15]. 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 IV and the JML abstraction functions also described before, 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 IV, 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. 10 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. 2 (b). This process can be carried out by security domain experts 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 Section IV, 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. For example, the role hierarchy depicted in Table II and Fig. 5, 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. 11 (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).

After the preparation steps, we applied our analysis technique to the applications under our case study, by following the mutation-based approach described before. We used a conventional Lenovo Thinkpad T510 laptop (Intel Core i7-620M Processor, 2.66GHz, 8 GB RAM). All mutants were automatically detected by ESC/Java2 even if they were hidden within the many methods of our case studies. The runtime of the three applications under our case study is given in Table III.

VI. Discussion and Related Work

The experimental results depicted in Section V-A 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 it becomes suitable for average developers to use [6]. (see Table I). We believe our approach can be extended to other programming languages/development platforms. For instance, Spec# [20] 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 [21] or Android [22], 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 to provide stronger guarantees for the verification. Second, as shown in Table III, 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 [14]. A possible solution would adopt a static approach for those methods whose test coverage is found to be below a given threshold.

Our work is related to other efforts in software security: Architectural risk analysis [23] 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 [24] allow software to be verified against information flow policies rather than supporting specific security requirements for different Security APIs. Formal verification of RBAC properties has been already discussed in the literature [16]. 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. [25]. In addition, Belhaouari et al. introduced an approach for the verification of RBAC properties based on DBC [26]. 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.

VII. 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. Also, we plan to refine our proposed RBAC model introduced in Section IV 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 language JML.

Acknowledgment

This work was partially supported by a grant from the US Department of Energy (DE-SC0004308) .

References

[1] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "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*, 2012, pp. 38–49.

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

[3] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[4] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Trans. Softw. Eng.*, vol. 21, no. 1, pp. 19–31, Jan. 1995.

[5] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct 1969.

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

[7] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards, "Model variables: cleanly supporting abstraction in design by contract: Research articles," *Softw. Pract. Exper.*, vol. 35, no. 6, pp. 583–599, May 2005.

[8] American National Standards Institute Inc., "Role Based Access Control," 2004, ANSI-INCITS 359-2004.

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

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

[11] OASIS, "XACML v3.0 Core and Hierarchical Role Based Access Control (RBAC) Profile Version 1.0," 2014, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cd-03-en.html.

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

[13] The Apache Software Foundation, "Apache shiro 1.2.1," 2013, http://shiro.apache.org/.

[14] Y. Cheon, "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., 2007.

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

[16] H. Hu and G.-J. Ahn, "Enabling verification and conformance testing for access control model," in *Proc. of the 13th ACM Symp. on Access Control Models and Technologies*, 2008, pp. 195–204.

[17] OSCAR EMR, "OSCAR Electronic Medical Records System," 2014, http://oscar-emr.com/.

[18] J. Gyger and N.l Westbury, "JMoney Financial System," 2014, http://jmoney.sourceforge.net/.

[19] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649 –678, 2011.

[20] M. Barnett, R. Leino, and W. Schulte, "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, 2005, pp. 49–69.

[21] Oracle Inc., "Java Platform Enterprise Edition," 2014, urlhttp://www.oracle.com/technetwork/java/javaee/overview/index.html.

[22] Google Inc., "Android," 2014, http://www.android.com.

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

[24] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[25] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan, "Security-by-contract: Toward a semantics for digital signatures on mobile code," in *Public Key Infrastructure*, ser. LNCS. Springer Berlin, 2007, vol. 4582, pp. 297–312.

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