Verifying Access Control Properties with Design by Contract: Framework and Lessons Learned

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

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

***Abstract*—Ensuring the correctness of high-level security properties including access control policies in mission-critical applications is indispensable. Recent literature has shown how immaturity of such properties has caused serious security *vulnerabilities*, which are likely to be exploited by *malicious* parties for compromising a given application. This situation gets aggravated by the fact that modern applications are mostly built on previously developed *reusable* software modules and any failures in security properties in these *reusable* modules may lead to vulnerabilities across associated applications. In this paper, we propose a framework to address this issue by adopting *Design by Contract* (DBC) features. Our framework accommodates security properties in each application focusing on access control requirements. We demonstrate how access control requirements based on ANSI RBAC standard model can be specified and verified at the source code level.**

***Index Terms*—security, access control, formal verification**

I. Introduction

Recent literature has shown severe consequences of mission-critical applications containing serious security *vulnerabilities*, which are believed to be caused by the *misuse* of the several *reusable* software modules such as application programming interfaces (APIs) that modern software applications rely on to provide security-based functionality [1], [2]. For instance, a secure communication channel is realized as a set of APIs in the software applications. Among the possible causes of this misuse problem, researchers have found the overall design and the informal specifications of such modules are often insufficient or complicated for developers to understand correctly. Consequently, they even fail to fully understand what the modules *do* and how their configuration parameters should be manipulated. Hence, the integration of their *self-developed* code and reusable modules may not be fully achieved. The problem gets aggravated by the fact that modern software applications are expected to make use of these *reusable* modules as much as possible, in an effort to reduce both the development costs and the production time. Unfortunately, this situation opens the door for the propagation of security vulnerabilities among several applications as a result of the incorrect enforcement of security properties in *reusable* software modules and threatens the security and safety of applications as a whole.

In order to cope with this problem, we propose a framework tailored for use of specification techniques, such as *design by contract* (DBC) [3], to provide high-level *abstract* descriptions of the security properties devised for a *reusable* software module, in such a way they can be found easy to understand and follow by future developers. In addition, we believe these specifications can be also used for providing automated verification techniques so the correct implementation of the security properties at the source code level can be verified. In this paper, we focus on security properties intended to provide access control guarantees for sensitive resources by means of a well-known *role-based access control* (RBAC) model [4]. As RBAC has emerged as the leading access control model for defining access control constraints, e.g. *who* is allowed to access *what*, correct implementation of these constraints becomes crucial to effectively enforce the access control model devised for a given software application. Throughout this paper, we adopt DBC which is an effective technique for defining, communicating and verifying the correct enforcement of RBAC constraints specified with the *Java Modeling Language* (JML), a DBC-based behavioral interface specification language for Java [5]. Using JML features, we model a set of classes from the main components of RBAC, as defined in ANSI RBAC standard model [6]. Using this new set of classes, we show how RBAC constraints can be defined in JML, relating them to other *behavioral* specifications also written in JML, in such a way it helps developers understand what a JML-specified software module is expected to *do* at runtime as well as corresponding RBAC constraints. Moreover, since our approach is mostly based on a well-defined standard, the RBAC constraints defined in our JML classes become independent of any supporting software module that is used to implement security features at the source code level. In other words, it also enables a seamless integration when the JML-specified module is *reused* to implement RBAC constraints differently. In order to verify that the RBAC constraints are correctly enforced at the source code level, we also propose an extension to *JET* [7], a JML-based tool providing automated runtime testing for Java modules. The contributions of this paper are as follows: First, it proposes a solution to the problem of misusing mission-critical software modules, by introducing a framework tailored for the specification and verification of

RBAC constraints based on the RBAC ANSI standard model (Section III). Second, using our approach, we introduce a set of different ways to specify RBAC constraints, in such a way that the specific RBAC requirements for applications can be better handled by both policy architects and software developers (Sections III-C1 III-C2 III-C3). Third, we provide a methodology to formally map these RBAC constraints with source code level constructs, in such a way the verification of the JML-based constraints against the source code can be achieved (Section III-D). Finally, we provide a customization of an existing tool to automatically carry out the verification task with the runtime testing (Section IV). We provide a summary of future work and concluding remarks in Sections V and VI.

II. Background

A. Design by Contract and JML

Design by Contract (DBC) [3] has been extensively explored in literature as a software development methodology based on the assumption that *implementers* and *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 *pre* and *post* conditions, among other related constructs. Before using a DBC-specified software module *M*, *clients* must make sure *M*'s preconditions hold. In a similar fashion, implementers must make sure *M*'s postconditions hold once *M* has finished execution and *M*'s preconditions were satisfied. The *Java Modeling Language* (JML) [5], is a behavioral interface specification language (BISL) for Java, with a rich support for DBC contracts. Using JML, the behavior of Java modules, e.g., what a Java class or interface is expected to *do* at runtime, can be specified using *pre*, *post* conditions, and class *invariants*, which are commonly expressed in the form of *assertions*, and are added to Java source code as comments of the form `//@` or `/*@...@*/`.

B. RBAC ANSI standard

Role-based Access Control (RBAC) is nowadays regarded as the leading access control model for defining and enforcing access control properties in software systems, mainly due to its flexibility, manageability and economy of use [4]. As RBAC started gaining popularity as a suitable solution for access control, there was a need to precisely define its main features and components, in an effort to allow for both research and commercial products to rely on a standardized reference other than in *custom-made* solutions. With this in mind, the *American National Standards Institute* (ANSI) [8] released a standard model that provides a well-defined yet flexible definition of RBAC, which is mostly based on the ideas collected throughout the years from researchers in both industry and academia. Such a standard model, referred as the RBAC ANSI standard [6], provides well-defined descriptions of the main RBAC features, including components, system and administrative functions, as well as a description of the different types of RBAC systems that have been discussed in literature.

III. Our Approach: Problem Statement and DBC-based Verification Framework

A. Problem Description

As modern software increases in both size and complexity, developers have turned into using pre-fabricated software *modules* that encapsulate some of the functions devised for their target software to reduce the overall costs and the time of the development process as well as to leverage the experience and knowledge invested in developing such modules, thus possibly reducing the existence of software *bugs* in their final products [9]. If such modules are to be *used* correctly, that is, they indeed contribute to the objectives just described above, it must be clear to developers about what a given module *does* (runtime behavior), how the module *communicates* with other modules, and what *security* constraints, if any, exist for it. This is particularly true for modules implementing highly-sensitive security functionality. Recently, lack of proper understanding of the runtime behavior of software modules has been regarded as the main cause for the existence of security *vulnerabilities*, as shown by Georgiev, et.al. [1], who discovered that a series of major software products failed to correctly use their supporting APIs for implementing security-sensitive functionality such as a secure communication channel. The main cause for these pitfalls was identified as both the poor design as well as the lack of proper specifications of the supporting APIs, in such a way that their intended behavior as well as the security constraints attained to them were easily understood by developers.

B. Model RBAC Classes

Figure 1 shows a snapshot of our proposed JML-based implementation of the RBAC ANSI standard. Following such a document, class `JMLRBACAbstractRole` defines both the basic data and functionality devised for roles in a RBAC model, and it is later refined by classes `JMLRBACCoreRole` and `JMLRBACHierarchicalRole`. The former is intended to model RBAC *core* settings, whereas the latter defines the functionality devised for *hierarchical* ones. The relationship between a given protected object (`JMLRBACResource`) and an operation (`JMLRBACOperation`) that can be performed over it is modeled by means of a *first-class* object of type `JMLRBACPermission`. Separation of duty constraints are modeled by class `JMLRBACAbstractSOD`, and proper refinements for both static (`JMLRBACSSD`) and dynamic (`JMLRBACDSD`) constraints are provided as well. Users, e.g. human agents, are modeled by means of class `JMLRBACUser`. For the brevity, formal verification of the adherence of our proposed approach and the RBAC ANSI standard is omitted.

C. Defining Model RBAC Constraints

As described in Section I we strive to present different ways to specify RBAC constraints based on the model classes described in Section III-B, in order to allow for RBAC policy architects to choose the approach that better fits both the RBAC and the *behavioral* requirements of their applications.

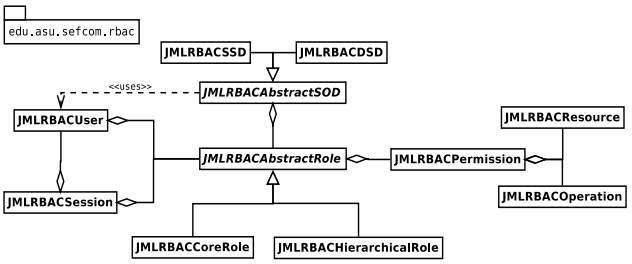

Fig. 1: JML *Model* Classes depicting the RBAC ANSI standard

First, we introduce our *role-based* constraints, which restrict access to a given Java method *M* by specifying a role, or set of roles, the *caller* of *M* must have before *M* is executed. Then, we present our so-called *session-based* constraints, which establish RBAC access restrictions based on the properties of a given *session*, as defined in the RBAC ANSI standard. Examples of these session-based constraints may include requiring a given user session to have several roles *active* at the time method *M* is called. Finally, we present our *permission-based* constraints, which restrict access to *M* by enlisting a set of RBAC *permissions* that must be granted before the method is executed successfully. Figure 2 shows examples of the proposed approach discussed in this section.

1) Role-based Constraints: Figure 2a shows an example of a role-based constraint , which is intended to restrict access to the security-sensitive `transfer()` method to only *callers* who manage to be granted the role *Manager*. We start by declaring the JML model field *role*, of type `JMLHierarchicalRole` (lines 5-6). As described in Section III-B, such a class is intended to model roles that may be organized in a role hierarchy, as defined in the RBAC ANSI standard. Next, we make use of such a JML *model* field to specify the need for such a role to be senior to role *Manager* (lines 11-12), which is in turn defined by means of the custom-made class `JMLRBACBankManagerRole` (not shown), a subclass of `JMLRBACHierarchicalRole`. Based on the semantics of the hierarchical RBAC component defined in the RBAC ANSI standard, any role that happens to be senior to the specified role *Manager* should satisfy this constraint correctly [1]. We outline how to map the model field *role* to an actual implementation construct in Section III-D.

2) Session-based Constraints: Figure 2b shows an example of a session-based constraint . Following the example discussed in the previous section, we require the activation of role *Manager* (`JMLRBACBankManagerRole`) during the current RBAC *session* defined by the JML model field of the same name (lines 5-6), of type `JMLRBACSession`. The RBAC constraint, which is in turn defined in lines 11-13, requires the current *session* to have the *Manager* role activated before a method `transfer()` is executed. The *pure* method `containsActiveRole()` of class `JMLRBACSession` returns true if the provided parameter has been activated as a role in the *session* object. As defined in the RBAC ANSI standard, a RBAC session *S* can have their roles activated dynamically if needed, as soon as the set of currently active roles is a subset of the roles *assigned*. We believe this kind of session-based constraints are useful when the concept of a *session* has been identified as a central feature required for a given software application. For instance, banking applications usually rely on session-based transactions, e.g. requiring a user to be authenticated before a subset of his/her *assigned* roles can be activated. Later on, operations on the system, e.g. transferring money between bank accounts, become available if the activated roles within the session contain proper permissions authorizing them.

3) Permission-based Constraints: Figure 2c shows the definition of a permission-based RBAC constraint requiring a given *role*, defined in lines 5-6, to have been authorized a permission regarded as *TransferPermission* of type `JMLRBACPermission`. Recall that in the RBAC ANSI hierarchical component, a permission *P* is said to be *authorized* to a given role *R* if it was explicitly assigned to *R*, or it has been assigned to another role that happens to be *junior* to *R*. Method `containsAuthorizedPermission()` of class `JMLRBACHierarchicalRole` first calculates the set of all authorized permissions for the receiving role object, by first retrieving all permissions assigned to roles that happen to be junior to it, even directly or as a result of exploring a given role hierarchy. Later, each of the retrieved permissions is compared against the provided permission parameter for equivalence, by using the `equals()` method of class `JMLRBACPermission`, which is overridden to better compare two given instances of such a class.

D. Mapping Model Constraints and Implementation Code

As described in previous sections, we aim to provide an approach for the runtime verification of the RBAC model constraints. As the first step, we provide a way to relate our model approach with the actual source code of a given application. For such a purpose, we leverage the existing JML features for relating model and source code constructs. As demonstrated in [10], JML provides a way to define proper source code values for model constructs by using the `represents` keyword. For Java reference types, e.g. our proposed `JMLRBACHierarchicalRole`, references to a JML *model* field are substituted by a reference to an implementation field of equivalent type. Moreover, in JML, it is also possible to define *model* methods, so the value of a model field gets assigned the result of evaluating a model method at runtime. Commonly, model methods providing an implementation value for a model field are regarded as *abstraction functions*. Figure 2d shows the `mapRole()` model method (lines 12-30) that is used in the `represents` clause for model field *role* (line 10). This model method serves as an abstraction function providing the mapping between our model field *role* and the actual source code implementation intended to enforce

[1] Our implementation of class `JMLRBACHierarchicalRole` guarantees that all roles defined by means of this class are both *junior* and *senior* roles of themselves.

```
1  //@ model import edu.asu.sefcom.rbac.*;
2  public interface BankAccount{
3
4  //@ public instance model int balance;
5  //@ public instance model
6  //@         JMLRBACHierarchicalRole role;
7
8  /*@ public normal_behavior
9    @ requires acc != null && amt > 0 &&
10   @          amt <= acc.balance &&
11   @          role.isSeniorRoleTo(
12   @          new JMLRBACBankManagerRole("Manager"));
13   @ assignable ...
14   @ ensures ...
15   @*/
16 public void transfer(BankAccount acc, int amt);
17 }
```

(a) *Role*-based RBAC constraints

```
1  //@ model import edu.asu.sefcom.rbac.*;
2  public interface BankAccount{
3
4  //@ public instance model int balance;
5  //@ public instance model
6  //@         JMLRBACSession session;
7
8  /*@ public normal_behavior
9    @ requires acc != null && amt > 0 &&
10   @          amt <= acc.balance &&
11   @          session.containsActiveRole(
12   @          new JMLRBACBankManagerRole("Manager"));
13   @ assignable ...
14   @ ensures ...
15   @*/
16 public void transfer(BankAccount acc, int amt);
17 }
```

(b) *Session*-based RBAC constraints

```
1  //@ model import edu.asu.sefcom.rbac.*;
2  public interface BankAccount{
3
4  //@ public instance model int balance;
5  //@ public instance model
6  //@         JMLRBACAbstractRole role;
7
8  /*@ public normal_behavior
9    @ requires acc != null && amt > 0 &&
10   @          amt <= acc.balance &&
11   @          role.
12   @          containsAuthorizedPermission(
13   @          new
       JMLRBACPermission("TransferPermission"));
14   @ assignable ...
15   @ ensures ...
16   @*/
17 public void transfer(BankAccount acc, int amt);
18 }
```

(c) *Permission*-based RBAC constraints

```
1  import org.apache.shiro.*;
2  //@ model import edu.asu.sefcom.rbac.*;
3
4  public class CustomerAccount implements BankAccount{
5
6   //@ private represents role <- mapRole();
7
8  /*@ private model pure
9    @              JMLRBACHierarchicalRole mapRole(){
10   @  JMLRBACHierarchicalRole newRole =
11   @  new JMLRBACHierarchicalRole("DefaultRole");
12   @
13   @  Subject currentUser = SecurityUtils.getSubject();
14   @
15   @  if (currentUser.hasRole("manager")) {
16   @   newRole = new JMLRBACBankManagerRole("Manager");
17   @   newRole.addPermission(
18   @   new JMLRBACPermission("TransferPermission"));
19   @  }
20   @  return newRole;
21   @ }
22   @*/
23 }
```

(d) Mapping *Role*-based constraints of Figure 2a to an implementation

Fig. 2: Approaches for specifying RBAC constraints using DBC

the constraints defined in the JML specifications. Figure 2d presents a case when the Apache Shiro [11] security API is used for implementation purposes. Using such an API, information about the externally-defined RBAC settings for the `CustomerAccount` application can be obtained and used to create proper JML model constructs depicting our approach in the body of the `mapRole()` abstract function. First, an instance of the Apache Shiro class `Subject` (*currentUser*), which contains information about the current executing user at runtime, is obtained (lines 19-20). Later, information contained on this *currentUser* reference is used to populate a freshly created instance of our model class `JMLRBACHierarchicalRole` (lines 22-28), in such a way the object returned by the `mapRole()` abstract function can be used every time a RBAC constraint using the *role* field is encountered as illustrated in Figure 2a (lines 11-13). Even though this running example has been mostly focused on our proposed role-based model constraints, a similar approach can be used to provide a mapping between our session and permission-based constraints and a corresponding source code implementation.

IV. Evaluation

A. Supporting Tool

JET [7] is a dedicated tool tailored for providing automated *unit* testing of JML-specified Java modules. Using *JET*, testers can verify the correctness of a Java module by checking the implementation of each of its methods (either public, protected or private) against their corresponding JML specifications. More details can be found at [7]. We have modified *JET* to allow for both *initialization* and *finalization* routines to be executed before and after a unit test is performed. An *initialization* routine is defined in such a way it retrieves all information regarding the RBAC settings devised for an application and creates proper data structures so that our *abstraction* functions discussed in Section III-D can effectively provide a *mapping* between the implementation constructs and the RBAC constraints. In a complementary way, a *finalization* routine is expected to perform some cleanup work, e.g. disposing data

structures, that better fits the testing process needs. As an example, in Figure 2d, an initialization routine would populate the necessary data structures such that the `Subject` class (lines 19-20) can effectively contain the RBAC information required by the `mapRole()` abstraction function (lines 12-31).

B. Experimental Process

We conducted a series of experiments tailored to measure the runtime performance as well as the effectiveness of both our approach and our supporting tool, which we describe in previous sections. For such a purpose, we provide a sample *banking* application based on the running example we have described throughout this paper. Such an application is distributed within 20 Java classes containing 833 lines of code and 861 lines of JML specifications, and depicts an RBAC model for restricting access to security-sensitive operations, e.g. *transfer*, *withdraw*, and *deposit*, which are implemented as Java methods. In addition, it makes use of the Apache Shiro API [11] for implementing security-related functionality. We performed our experiments 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. The Java JRE used was Java SE 1.7.0_06 as provided by Oracle Inc. Our first experiment was intended to measure the impact of our approach in the runtime performance of our sample application. As described in [7], our supporting tool *JET* translates our proposed model classes into RAC code, which is used to provide runtime verification for RBAC constraints. We executed a sample trace of the Java methods exposed by our sample application and calculated the average execution time over 1,000 repetitions. Figure 3 presents our experimental results. The notation R-*X* refers to an execution of our sample trace by using a RBAC setting with a *X* number of roles. The term *W/RAC* refers to the sample application compiled with a standard Java compiler, whereas *RAC* denotes the same application compiled with our supporting tool. All running times are measured in milliseconds. As expected, the introduction of RAC code has a noticeable impact on performance. This is mostly due to the RAC code generated to process both the JML contracts as well as the *abstraction* functions (Section III-D) mapping our model classes with implementation constructs. In addition, increasing the number of roles in the sample RBAC setting also increases execution time, as more processing time is needed for the abstraction functions to process a larger number of roles.

A second experiment was designed to measure the effectiveness of our supporting tool to detect faulty implementations of RBAC constraints. For such a purpose, we augmented our banking application with an auxiliary method, called `checkRoles()` (not shown), intended to check at runtime if a list of roles, received as a parameter, includes any role allowed to execute a given banking method under test. If so, `checkRoles()` returns quietly, otherwise, a security exception is thrown. Information about the roles allowed to execute a given method is provided by our RBAC constraints,

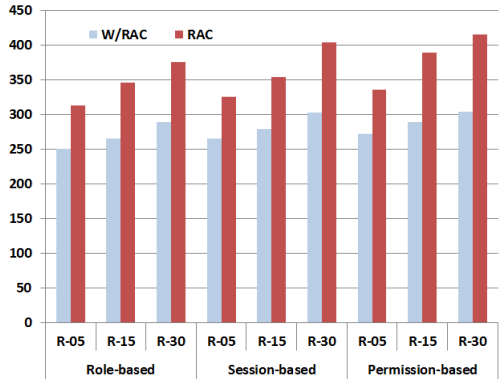

Fig. 3: Performance measurements for a sample banking application

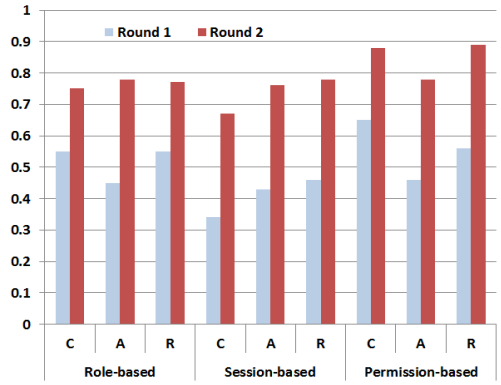

Fig. 4: Effectiveness measurements for a sample banking application

whereas information on the executing roles at runtime is provided by the Apache Shiro API, in an approach similar to the one shown in Figure 2d. Based on this, we followed an approach based on mutation testing [12], by deliberately inserting changes, also known as *mutants*, to the list or roles taken by `checkRoles()` as a parameter: first, we introduced a technique calling *adding* mutants, which would insert additional role names to the forementioned parameter list. In a similar fashion, we introduced our *changing* and *removing* techniques, which would change and remove role names for the parameter list, respectively. We applied these three techniques to different methods of our sample application and recorded the results obtained by our tool: In an initial *first* round, we observed the performance of our tool to detect our manually-inserted *mutants* when provided with our original set of RBAC constraints as JML specifications. We run each method separately, activating a single *mutation* at a time. Later, in a *second* round, we introduced some changes in our JML specifications in an effort to increase the amount of faulty implementation cases detected by the tool, and executed the testing process once again. Changes in the JML specifications included both the *refinement* of the RBAC constraints, e.g. adding specification cases, and the implementation of the abstraction functions introduced in Section III-D. We repeated

the experimental process several times and collected the results shown in Figure 4 [2]. Our results show the *effectiveness* rate, that is, the number of successfully-detected user cases divided by the total number of use cases produced by the tool. As it can be observed in Figure 4, initial effectiveness rates detected during round 1 are significantly improved after round 2, due to the fact the JML specifications are strengthened to better handle previously undetected cases.

V. Related and Future Work

Formal verification of RBAC properties has been already discussed in literature [13] [14] [15]. 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 works closely related to ours involve the use of DBC for security-related purposes, which was explored by Dragoni, et.al. [16] and Nico, et.a. [17]. In addition, Belhaouari, et.al. [18] introduce an approach for the verification of RBAC properties using a DBC-like approach. Approaches similar to ours are presented by Mustafa and Sohr [19] and Rubio and Cheon [20]. A summary of our future work comes as follows: first, we plan to work on a *refinement* of the JML model classes introduced in Section III-B, in an effort to better accommodate for a new kind of RBAC constraints using JML specifications, as well as for a better implementation of the RBAC ANSI standard. Second, we plan to extend the capabilities of our proposed extension to the *JET* tool in such a way the efficiency of the tool for discovering faulting implementations can be increased. Finally, we also plan to explore the suitability of our model classes to accommodate for other JML-tools, e.g. tools based on *static* analysis, as it was discussed in [19], so we can leverage the benefits offered by using both *dynamic* and *static* approaches for source code verification.

VI. Conclusions

In this paper, we have introduced a solution for the problem of misusing reusable software modules, e.g. APIs, which has been found to cause serious vulnerabilities in mission-critical software applications. In order to cope with this problem, we have introduced a framework for the abstract specification of security constraints based on ANSI RBAC, a formal model for providing access control guarantees in software systems. The approach described in this paper is mostly based on the Java Modeling Language, a DBC-based specification language for Java, and includes both a set of *model* classes based on the RBAC ANSI standard as well as a supporting tool to carry out the runtime verification of the implementing source code against a set of JML-based specifications. Based on the results presented throughout this work, we believe DBC and JML are effective tools to allow for software designers, policy architects, and developers to better communicate, implement, and verify the correct enforcement of the RBAC constraints devised for software applications.

[2]The terms *C*, *A* and *R* stand for the *changing*, *adding* and *removing* techniques described above.

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 *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 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 *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 50–61.

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

[4] 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.

[5] 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.

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

[7] Y. Cheon, "Automated random testing to detect specification-code inconsistencies," in *Proceedings of the 2007 International Conference on Software Engineering Theory and Practice*, 2007.

[8] American National Standards Institute, "ANSI Website," 2013, http://www.ansi.org.

[9] F. Foukalas, Y. Ntarladimas, A. Glentis, and Z. Boufidis, "Protocol reconfiguration using component-based design," in *Proceedings of the 5th IFIP WG 6.1 international conference on Distributed Applications and Interoperable Systems*, ser. DAIS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 148–156.

[10] 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.

[11] T. A. S. Foundation, "Apache shiro 1.2.1," 2013, http://shiro.apache.org/.

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

[13] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[14] M. Drouineaud, M. Bortin, P. Torrini, and K. Sohr, "A first step towards the formal verification of security policy properties of rbac," in *Proceedings of the 4th International Conference on Quality Software (QSIC)*, H.-D. Ehrich and K.-D. Schewe, Eds., 2004.

[15] H. Hu and G. Ahn, "Enabling verification and conformance testing for access control model," in *Proceedings of the 13th ACM symposium on Access control models and technologies*, ser. SACMAT '08. New York, NY, USA: ACM, 2008, pp. 195–204.

[16] 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. Lecture Notes in Computer Science, J. Lopez, P. Samarati, and J. Ferrer, Eds. Springer Berlin Heidelberg, 2007, vol. 4582, pp. 297–312.

[17] P. L. Nico, C. S. Turner, and K. K. Nico, "Insecurity by contract," in *Proceedings of the IASTED Conference on Software Engineering and Applications, November 9-11, 2004, MIT, Cambridge, MA, USA*, M. H. Hamza, Ed. IASTED/ACTA Press, 2004, pp. 269–274.

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

[19] T. Mustafa, M. Drouineaud, and K. Sohr, "Towards Formal Specification and Verification of a Role-Based Authorization Engine using JML (Position Paper)," in *5th ACM ICSE Workshop on Software Engineering for Secure Systems (SESS10), South Africa*, May 2010.

[20] C. Rubio and Y. Cheon, "Access control contracts for java program modules," in *Proceedings of the 5th IEEE International Workshop on Security, Trust, and Privacy for Software Applications*, ser. STPSA 2010, July 19-23 2010.