Access Control Contracts for Java Program Modules

Carlos E. Rubio-Medrano and Yoonsik Cheon

TR #10-07
March 2010; revised April 2010

Keywords: access control contracts, pre and postconditions, runtime assertion checking, stack inspection, JML.

1998 CR Categories: D.2.2 [*Software Engineering*] Design Tools and Techniques — Modules and interfaces; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, formal methods, programming by contract; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; D.4.6 [*Operating Systems*] Security and Protection — Access controls; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, pre- and post-conditions, specification techniques.

To appear in the *5th IEEE International Workshop on Security, Trust, and Privacy for Software Applications (STPSA 2010)*, July 19-23, Seoul, Korea.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Access Control Contracts for Java Program Modules

Carlos E. Rubio-Medrano
Department of Computer Science
The University of Arizona
Tucson, Arizona, 85721, USA
cerubiom@cs.arizona.edu

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, 79968, USA
ycheon@utep.edu

***Abstract*—Application-level security has become an issue in recent years; for example, errors, discrepancies and omissions in the specification of access control constraints of security-sensitive software components are recognized as an important source for security vulnerabilities. We propose to formally specify access control assumptions or constraints of a program module and enforce them at run-time. We call such specifications *access control contracts*. To realize access control contracts, we extended the JML language, a formal interface specification language for Java, and developed a prototype support tool that translates access control contracts to runtime checks. The access control contract reduces the vulnerability that a security-sensitive module be exploited to compromise the overall security of a software system. It also facilitates practicing the principle of "security by design" by providing both a practical programming tool and a foundation for formally reasoning about security properties of program modules.**

***Keywords*-access control contracts; pre and postconditions; runtime assertion checking; stack inspection; JML**

I. Introduction

Proper enforcement of security measures at the software application level has become an important concern recently, as many enforcement techniques have traditionally focused on system level concerns, leaving attackers with a chance of compromising the safety of a software system by manipulating the data and resources being accessed by an application. As an example, consider the deployment of a web application, which is widely used and affected by several sources of vulnerabilities because of the data exchange between a remote client and the web application [1]. If the data being exchanged are improperly validated, a malicious third party may be able to trick the web application into executing an improper operation on sensitive data (e.g, SQL injection problem [2]). Over the years, several techniques have been developed, primarily based on high-level, well-typed languages such as Java, to prevent the exploitation of well-known security vulnerabilities such as buffer overflow and control flow hijacking. Support is also provided for application-level security, especially to control access to sensitive resources. In Java, for example, the Java Security Architecture (JSA) allows for developers to enforce access control constraints at runtime by restricting access to sensitive resources according to a precisely specified security policy [3].

It is equally important to also specify and enforce component-level security. When building a security-sensitive application by reusing or assembling existing software components, for example, inaccurate descriptions of access control constraints of the components can potentially lead to a security hole, forcing developers to invest a considerable amount of effort trying to analyze and modify the components to assure their conformance to the security requirements of the application. In fact, this has been recognized as a potential cause of security vulnerabilities for many software systems [4]. The problem is that an improper description of a component's behavior may lead to an application developer to inaccurately establish a series of security measures, which may fail to capture all possible misuses of the interface provided by the component, thus allowing for exploitable vulnerabilities to exist. An attacker can compromise the overall safety of a system by using the application and the interface provided transitively by its components to access security-sensitive resources of the system. It is a challenging task for an application developer to know the real capabilities of an application's components to access security-sensitive resources and their implications. In general, an exhaustive analysis or testing is needed to ensure that a component is used in a safe way by an application and its other components.

We claim that formally specifying security concerns of a software component and checking them at runtime alleviate the above-mentioned problem. We also claim that such a specification should be part of the interface description of the component. In order to prove our claim, we extend the Java Modeling Language (JML) [5], a formal interface specification language for Java to describe the behavior of Java program modules such as classes and interfaces. Our extension allows one to formally describe the access control requirements of Java program modules, which we call *access control contracts*. The access control contract of a program module defines a set of access control constraints and relates them to the expected runtime behavior of the module. We first show that JML is ineffective for writing access control contracts. We then propose an extension by introducing a declarative approach for describing the access control contracts in terms of the Java Security Architecture (JSA). JSA is based on a model known as *stack-based access control* [6] and allows one to specify the permissions needed for a program to successfully execute. We also develop a prototype support tool to check at runtime the access control contract of a program module. Our prototype shows that tool support for the access control contract can be easily integrated into the existing JML tools [7].

The main contribution of our work is the introduction of access control contracts to formally specify the access control requirements of a program module. By introducing access control contracts, we related the access control and functional behavior constraints in the same framework of Hoare-style pre and postconditions. By doing so, we also made access control contracts to be part of the interface of a program module. We expect that our work provide a formal basis for reasoning about security properties of Java program modules in a modular way.

The rest of this paper is organized as follows. In Section II, we give a quick overview of both JML and JSA. In Section III, we introduce the concept of access control contracts. We explain in detail how we realize this concept in JML. In Section IV, we describe the experiments that we performed to evaluate the effectiveness of access control contracts. We next describe related work in Section V. We then conclude this paper with a concluding remark in Section VI.

II. Background

A. The Java Modeling Language

The Java Modeling Language (JML) is a behavioral interface specification language tailored for specifying both the runtime behavior and the syntactic interface of Java program modules such as classes and interfaces [5] [7]. It uses Hoare-style pre and postconditions to describe what a module is expected to do at runtime [8]. JML specifications are commonly written in the source code file as special comments. Figure 1 shows an example JML specification. As shown, a JML specification precedes the Java declaration such as a method declaration that it annotates. The **requires** clause specifies the precondition, and the **ensures** clause specifies the postcondition. The precondition of the `connect` method states that the port number should be between 1024 and 4096, inclusive, and the host should be valid. The special syntax `(* *)`, known as an *informal description*, allows one to escape from formality when writing JML assertions. JML treats an informal description as a boolean expression and thus allows to mix formal and informal descriptions in assertions. The pseudo variable `\result` in the postcondition denotes the return value of a method. The postcondition states that the resulting socket should be freshly created and its port number and host name should be same as the given port number and host name.

```
public interface NetworkManager {

 /*@ requires 1024 <= port && port <= 4096 &&
   @   (* host is a valid host name *);
   @ ensures \fresh(\result)
   @   && \result.isConnected()
   @   && \result.getPort() == port
   @   && \result.getHost().equals(host);
   @*/
  public Socket connect(String host, int port);
}
```

Figure 1. Example JML specification

```
public interface NetworkManager {

  /*@ public model boolean isValid(String host) {
    @   if (host == null || host.length() == 0)
    @      return false;
    @   ...
    @ }
    @*/

  //@ requires isValid(host);
  public Socket connect(String host, int port);
}
```

Figure 2. Example model method in JML

An interesting feature of JML is that it supports specification-only declarations [9]. Commonly known as *model elements*, they allows one to write abstract specifications by introducing fields, methods, and classes only to be used in JML specifications but not in Java code. For example, we can formulate the notion of validity of a host name by introducing a model method that checks well-formedness of a host name. Figure 2 shows a model method named `isValid` that checks whether a given host name is valid or not. As shown, a model method can be used only in JML assertions such as the precondition of the `connect` method. The JML compiler can translate assertions written in terms of model elements into runtime checks. In Section III below, we will show how we can utilize model methods to write access control contracts in JML and check them at runtime.

B. The Java Security Architecture

The Java Security Architecture (JSA) was originally designed to enforce security measures when executing Java applets downloaded from a network [3] [10]. Applets are executed by placing them in a so-called sandbox environment, which restricts access to any sensitive resources. This approach was too restrictive in practice because many trusted applets couldn't run properly, thus limiting their potential benefits for distributed applications. Later, a more flexible yet secure architecture was developed to allow for the mediation of access to protected resources. As before, an application is placed in a sandbox environment and by default denied access to any sensitive resources such as system files and network ports. In order for an application to access a protected resource, it must be granted a proper permission over it; a permission is represented as a subclass of `java.security.Permission`. JVM uses a security policy file to determine if access to a resource requested by an application should be granted or denied. An entry in the policy file commonly includes, among other things, (1) the name of the class or jar file the permission is granted to, (2) the code base (i.e., location) from which the code was downloaded, and (3) the creator of the code identified by a digital signature. At runtime, when an application, A, requests access to a protected resource, R, all classes composing A that are located in the current execution call stack must have been granted a permission, say P, to access R. This checking process is known as *stack inspection* [11] [12]. If any class in the current execution stack is found not to be granted the required permission, P, then the access to R is denied; otherwise, R can be accessed by A in the way allowed by P, e.g., reading or writing a system file. Note that for an application running outside the sandbox environment the stack inspection procedure never takes place, and thus all permissions requested by the components of such an application are granted.

III. Access Control Contracts

We propose to document the access control requirements of a program module. Such documents, that we call *access control contracts*, should be part of the module's interface along with the description of the module's functional behavior. The access control contract of a module define precisely the access control constraints of the module, e.g, by listing all the permissions needed to invoke the module successfully. We show that both the access control constraints and the functional behavior of a program module can be specified in the framework of Hoare-style pre and postconditions. If an access control constraint appears in the precondition, it is a client's obligation in that the client has to invoke the module in a state where the constraint is satisfied; otherwise, nothing is guaranteed. On the other hand, if an access control contract appears in the postcondition, it is an implementer's obligation in that the implementer has to guarantee for the constraint to be satisfied when the module completes its execution. Below we explain how one can write access control contracts in JML.

A. Writing Access Control Contracts in JML

As explained in Section II-A, model methods can be used to introduce an additional vocabulary for writing JML assertions. Moreover, when an implementation of a model method is provided, JML assertions written in terms of model methods can be translated to runtime checks and thus executed at runtime [9]. We can utilize model methods to introduce a new vocabulary for writing access control contracts. For example, Figure 3 shows a refinement of the JML specification shown in Figure 1 to include

```
public interface NetworkManager {
 /*@ public model boolean hasPermission(Permission p) {
   @   try {
   @     AccessController.checkPermission(p);
   @   } catch (AccessControlException) {
   @     return false;
   @   }
   @   return true;
   @ }

 /*@ requires hasPermission(
   @   new SocketPermission(host + ":" + port,
   @     "connect"))
   @ requires 1024 <= port && port <= 4096 &&
   @   (* host is valid *);
   @ ensures \fresh(\result)
   @   && \result.isConnected()
   @   && \result.getPort() == port
   @   && \result.getHost().equals(host);
   @*/
  public Socket connect(String host, int port);
}
```

Figure 3. Access control contract written in JML using a model method

an access control constraint for the `connect` method. It defines a model method named `hasPermission` that takes a permission object and checks if the permission can be granted. The model method was written in terms of the `checkPermission` method of the `AccessController` class. The `checkPermission` method performs the stack inspection procedure to determines whether the access request indicated by the specified permission should be allowed or denied, based on the current access control context and security policy. It quietly returns if the access request is permitted; otherwise, it throws an access control exception. The `connect` method now has an additional **requires** clause to state its access control constraint. It states that a "connect" socket permission on the given host and port number is required to execute the method successfully; if a client calls the method without the specified permission, nothing is guaranteed. By writing explicitly the access control constraints of a program module and making it part of the module's interface, both the client and the implementer are now better informed of potential security concerns or consequence of using or implementing the module.

The main advantage of writing access control contracts in this way is that it doesn't require additional support from JML or its tools. Access control contracts are just regular JML assertions written using model methods, and thus can be processed with existing JML tools such as the JML compiler to produce runtime checks. However, there are also several shortcomings. The assertions tend to be long and verbose, as one has to introduce a model method. It also has an imperative flavor because one has to write the body of the model method for runtime checks. For this, one has to be familiar with JSA and its particular implementation, e.g., the `AccessController` class and its methods. Another problem is that there is no standard way of writing model

```
public interface NetworkManager {
 /*@ requires \has_permission(SocketPermission,
   @   host + ":" + port, "connect")
   @ requires 1024 <= port && port <= 4096 &&
   @   (* host is a valid host name *);
   @ ensures \fresh(\result)
   @   && \result.isConnected()
   @   && \result.getPort() == port
   @   && \result.getHost().equals(host);
   @*/
  public Socket connect(String host, int port);
}
```

Figure 4. Access control contract written using permission expression

methods for access control contracts, and thus different programmers or specifiers can introduce model methods of different styles, e.g., with different names, signatures, logics, and bodies. The resulting assertions may be hard to read and understand, especially when they are tangled with regular assertions stating functional behaviors. The absence of a standard way to define access control contracts also increases the complexity of creating and integrating support tools. Thus, it would be difficult to develop tools specifically for access control contracts.

B. Extending JML for Access Control Contracts

We extended JML to provide built-in support for writing access control contracts. One such a construct is a new JML expression, `\has_permission` , that tells if the current execution has the specified permission. A permission is denoted by specifying the name of a permission class along with the arguments for creating an instance of the permission class; a permission class is a subclass of the `java.security.Permission` class. By using this expression, one can write access control contracts declaratively. For example, Figure 4 shows the specification of the `connect` method rewritten using the permission expression. As before, the first precondition states that a client has to have a "connect" socket permission on the given host and port number to execute the method successfully. However, the specification is now more succinct and doesn't involve Java code.

The syntax of permission expression is given below.

```
<perm_expr> ::= \has_permission(<class_name> [, <args>])
<args> ::= <expr> | <expr> , <args>
```

The first argument should be the name of a permission class, any subclass of the `java.lang.Permission` class, and the rest should be arguments appropriate for invoking a constructor of the permission class. For example, a socket permission denoting access to a network via sockets can be denoted by the class `SocketPermission` along with a host specification (e.g, "www.cs.utep.edu:80") and a set of actions (e.g., "connect, accept") specifying ways to connect to that host.

Informally, the meaning of the permission expression is defined as follows. Each permission expression denotes a permission object, p, and the current execution must be granted the required security clearance with respect to p; i.e., the execution must be granted p or another permission q that implies p. In addition, the security clearance restriction is extended to include all the callers that are present in the execution call stack. Thus, operationally, the permission expression denotes the stack inspection procedure explained in Section II-B.

In addition to the permission expression, we introduced another new expression to restrict the set of callers based on their static declarations. This expression, called `\called_by` expression, takes a class name or a method signature and evaluates to true if the current execution is originated from the given class or method. We found this expression is useful for specifying the behavior of a module that provides different services to different clients.

C. The `jmlacc` tool

In order to support the extension described in the previous section, we developed a prototype tool called `jmlacc` using Polyglot [13], an extensible compiler framework. The tool supports a small subset of JML along with our extension and is tailored to produce runtime assertion checking code by translating JML specifications into Java bytecode in a similar fashion as done by the JML compiler [14]. We also used the wrapper method approach to wrap the original method with assertion checking methods such as pre and postcondition checking methods (see Figure 5 for sample instrumented code). What is new is the treatment of the permission expression. A permission expression is translated to a call to a separate checking method that, if a security manager is installed, triggers the stack inspection procedure (refer to the `checkPerm` method in Figure 5).

IV. Evaluation

In order to evaluate the effectiveness of our approach we inspected JML-annotated classes found in the `java.io` and `java.security` packages. The JML specifications for the I/O classes were obtained from the JML distribution available at the JML website (`http://www.jmlspecs.org`), and the specifications for the security classes were from our earlier effort on formalizing the behavior of Java security classes in JML [15]. Our findings were surprising. Only few classes had specifications of access control constraints, and the access control-related properties were written informally. We believe that this was because JML doesn't provide built-in support for writing such specifications.

We were able to complete and refine existing specifications using our extension to JML. In some cases, we decomposed the expected behavior of a method based on the access control requirements. For example, Figure 6(a) shows the original JML specification for the `canRead` method of class `java.io.File`. The **signals_only** clause states

```
public class MyManager implements NetworkManager {

  public Socket connect(String host, int port){
   checkPre$$connect(host, port);
   Socket result = connect$$orig(host, port);
   checkPost$$connect(host, port, result);
   return result;
  }

  private Socket connect$$orig(String host, int port) {
    // original code for connect
  }

  private void checkPre$$connect(String host, int port) {
    if (!(checkPerm(new SocketPermission(host + ":" + port,
                                        "connect"))
          && 1024 <= port && port <= 4096))
       throw new JMLPreconditionError();
  }

  private void checkPost$$connect(String host, int port,
    Socket result) {
    if (!(result.isConnected()
          && result.getPort() == port
          && result.getHost().equals(host)))
       throw new JMLPostconditionError();
  }

  private boolean checkPerm(Permission p) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
       try { sm.checkPermission(p); }
       catch (AccessControlException e) {
         return false;
       }
    }
    return true;
  }
}
```

Figure 5. Skeletal runtime checking code

that the method may terminate abruptly by throwing only a security exception; however, the specification doesn't say when it may throw such an exception. According to its API documentation, the `canRead` method requires its clients to be granted a file permission to access the denoted file. Figure 6(b) shows a revised specification written using the new permission expression. The **old** clause introduces a name for an expression. The revised specification now clearly says that the method may throw a security exception only if the caller doesn't have a "read" file permission on the file; otherwise, the method should return normally.

V. Related Work

The use of formal languages for specifying security constraints of software modules has received considerable attention lately, and there were several attempts to extend JML to specify security constraints. Groslambert et al. studied the specification and verification of liveness and safety properties for Java classes using JML [16]. Huisman and Tamalet recently proposed to specify security properties in JML by making use of security automata [17]. We believe that our approach for using JML to write security properties and check them at runtime provides a complementary technique to these approaches.

```
  /*@ ensures (* true iff the file has read access *);
    @ signals_only SecurityException;
    @*/
  public boolean canRead();
```

(a) Original specification

```
/*@  old boolean perm = \has_permission(FilePermission,
  @   getPath(), "read");
  @  {|
  @    requires perm;
  @    ensures \result == this.exists()
  @      && (* the file can be read *);
  @   also
  @    requires !perm;
  @    signals_only SecurityException;
  @  |}
  @*/
public /*@ pure @*/ boolean canRead();
```

(b) Revised specification

Figure 6. JML specification for the File.canRead method

Smans et al. discussed specification of access control constraints in .NET CLR framework [18]. They proposed to use the Spec# specification language to define access control constraints for C# modules. As in JSA, the CLR framework makes use of the stack inspection mechanism to enforce the constraints at runtime. In principle, their approach is similar to ours in that special language constructs were introduced for specifying the runtime permissions needed to use a component. However, our approach makes a stronger connection between access control contracts and functional behavior specifications of a module and allows the behavior of the module be defined in terms of access control contracts. We also showed how to leverage existing facilities such as model methods to write access control contracts.

Hussein and Zulkernine proposed an approach for specifying intrusion detection scenarios by using an extension to UML [19]. Their approach focuses on describing possible misuse cases of software components—i.e., cases that malicious parties could implement to use software components in ways not intended by their original developers—to identify possible security breaches. On the other hand, our approach focuses on specifying the correct or expected behavior.

Pistoia et al. provided a comprehensive description of potential vulnerabilities that can lead to security holes in Java applications [4]. Interestingly they pointed out that the lack of correct and complete specifications of access control constraints is an important problem that can indeed compromise the overall security of a software system being built from heterogeneous components. They also described current techniques for static enforcement of access control constraints.

VI. Conclusion

We introduced the notion of *access control contracts* to document the access control constraints of Java program modules. We showed how to write access control contracts formally in JML, a formal interface specification language for Java. Our technique is to introduce a specification-only method, called a *model method*, that performs the stack inspection procedure and to refer to it in method pre and postconditions. We also extended the JML language to provide built-in language support for writing access control contracts. In particular, we introduced a new permission expression that tests whether the current execution can be granted the required security clearance with respect to a given permission. To prove the feasibility of our approach, we developed a prototype tool that translates the permission expression to a runtime check. Our case study, though limited in scope, showed a very promising result, and we next plan to incorporate our extension into the JML language by extending its support tools.

Acknowledgment

Rubio's work was supported in part by NSF grant CNS-0509229 and a Graduate Fellowship awarded by the Mexican Chihuahua State Government. Cheon's work was supported in part by NSF grants CNS-0509299 and CNS-0707874.

References

[1] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, "Bridging the gap between web application firewalls and web applications," in *FMSE '06: Proceedings of the fourth ACM workshop on formal methods in security*. New York, NY, USA: ACM, 2006, pp. 67–77.

[2] W. G. J. Halfond and A. Orso, "Amnesia: analysis and monitoring for neutralizing sql-injection attacks," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 174–183.

[3] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: an overview of the new security architecture in the java development kit 1.2," in *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, December 1997, pp. 10–10.

[4] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM Systems Journal*, vol. 46, no. 2, pp. 265–288, 2007.

[5] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[6] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *Journal of Functional Programming*, vol. 15, no. 2, pp. 131–177, 2005.

[7] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, Jun. 2005.

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

[9] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, "Model variables: Cleanly supporting abstraction in design by contract," *Software—Practice & Experience*, vol. 35, no. 6, pp. 583–599, May 2005.

[10] L. Gong, "Java security: Present and near future," *IEEE Micro*, vol. 17, no. 3, pp. 14–19, 1997.

[11] D. S. Wallach and E. W. Felten, "Understanding Java stack inspection," *Security and Privacy, IEEE Symposium on*, vol. 0, pp. 52–63, 1998.

[12] C. Fournet and A. D. Gordon, "Stack inspection: Theory and variants," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 3, pp. 360–399, 2003.

[13] N. Nystrom, M. Clarkson, and A. Myers, "Polyglot: An extensible compiler framework for Java," in *12th International Conference on Compiler Construction*. Springer-Verlag, 2003.

[14] Y. Cheon and G. T. Leavens, "A runtime assertion checker for the Java Modeling Language (JML)," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, H. R. Arabnia and Y. Mun, Eds. CSREA Press, Jun. 2002, pp. 322–328.

[15] P. Agarwal, C. E. Rubio-Medrano, Y. Cheon, and P. J. Teller, "A formal specification in JML of the Java security package," in *Advances and Innovations in Systems, Computing Science, and Software Engineering*, K. Elleithy, Ed. Springer, 2007, pp. 363–368.

[16] J. Groslambert, J. Julliand, and K. Olga, "JML-based verification of liveness properties on a class," in *Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006)*, November 2006, pp. 41–48.

[17] M. Huisman and A. Tamalet, "A formal connection between security automata and JML annotations," *Fundamental Approaches to Sofware Engineering (FASE)*, vol. 5503 of Lecture Notes in Computer Science, pp. 340–354, 2009.

[18] J. Smans, B. Jacobs, and F. Piessens, "Static verification of code access security policy compliance of .NET applications," *Journal of Object Technology*, vol. 5, no. 3, pp. 35–58, May-June 2006.

[19] M. Hussein and M. Zulkernine, "Intrusion detection aware component-based systems: A specification-based framework," *Journal of Systems and Software*, vol. 80, no. 5, pp. 700–710, 2007.