A survey of static analysis methods for identifying security vulnerabilities in software systems

M. Pistoia
S. Chandra
S. J. Fink
E. Yahav

In this paper we survey static analysis methods for identifying security vulnerabilities in software systems. We cover three areas that have been associated with sources of security vulnerabilities: access-control, information-flow, and application-programming-interface conformance. Because access control mechanisms fall into two major categories, stack-based access control and role-based access control, we discuss static analysis techniques for these two areas of access control separately. Similarly, security violations pertaining to information flow consist of integrity violations and confidentiality violations, and consequently, our discussion of static analysis techniques for information-flow vulnerabilities includes these two topics. For each type of security vulnerability we present our findings in two parts: in the first part we describe recent research results, and in the second part we illustrate implementation techniques by describing selected static analysis algorithms.

INTRODUCTION

Security of software systems touches on a vast and complex array of issues, making it difficult and expensive to implement a comprehensive security solution. In practice, software development organizations attempt to adhere to a variety of known security principles[1] and security guidelines[2,3] that facilitate the design and implementation of secure software systems.

Ensuring compliance with security guidelines can be especially challenging. These guidelines are often complex, and information technology (IT) professionals are prone to making mistakes, especially when dealing with large programs comprising multiple components, whose security properties may differ. Such is the case, for example, when assembling a Web application consisting of components with differing access-control requirements and levels of trust. Similarly, the use of a third-party component whose application-programming interface (API) is not properly documented presents a security risk. Although organizations could conduct manual code inspections to identify security problems, such reviews are time consuming and expensive, and their coverage uncertain.

To address this problem, the research community has proposed automatic analysis techniques for identifying security vulnerabilities in code. Such techniques may employ *dynamic analysis*, *static analysis*, or both. Dynamic analysis entails executing the program and inferring properties from its observed behavior. Although dynamic analysis may expose bugs, it cannot ensure complete coverage of the target program and thus ensure compliance with security guidelines. Instead, developers have adopted an approach based on *sound* analysis, which can provably identify all possible violations of specific security guidelines.

In this paper we survey static analysis techniques that can be used to automatically detect security vulnerabilities in software systems. These techniques cover three areas that have been associated with sources of security vulnerabilities[4–7]:

1. *Access control*—The mechanisms for access control restrict access to security-sensitive resources based on a user's identity and membership in predefined groups. Ensuring that an access control policy enforces the required level of security can be difficult, especially for systems with myriad components of different trust levels with access to a multitude of restricted resources. If an access control policy does not grant sufficient permissions to users, runtime authorization failures may result. Conversely, if an access control policy grants users unnecessary permissions, the policy may expose a system to security attacks.
2. *Information flow*—A secure *information flow* ensures that information propagates throughout the execution environment without violating two classes of security violations:

   (a) *Integrity violations* arise when untrusted information flows into a trusted execution environment without having been properly validated. A malicious user could compromise a system by exploiting an integrity violation.
   (b) *Confidentiality violations* arise when confidential information flows from a restricted execution environment to a public one without having been properly declassified. For example, a confidentiality violation arises if trusted code exposes a cryptographic private key to untrusted code.

3. *API conformance*—Web applications often rely on libraries and third-party APIs to provide security-sensitive services. As an example, many applications rely heavily on cryptography libraries to protect confidentiality, prevent integrity violations, and distinguish between trusted and untrusted entities. Incorrect usage of cryptographic functions may lead to insecure storage of security-sensitive information and cause violations of integrity and confidentiality policies.

The range of vulnerabilities that can be addressed through the static analysis techniques surveyed here can be illustrated by considering the top 10 security violations in today's Web applications according to the Open Web Application Security Project (OWASP).[4] These top 10 security violations are: (A1) unvalidated input (information passed through Web requests is not validated before being used by a Web application), (A2) broken access control (access control policies are not properly enforced), (A3) broken authentication and session management (account credentials and session tokens are not properly protected), (A4) cross-site scripting (the Web application is used as a vehicle for an attack against a local machine), (A5) buffer overflow, (A6) injection flaws (malicious commands embedded in parameters passed by Web applications), (A7) improper error handling, (A8) insecure storage (such as that caused by improper coding of cryptographic functions), (A9) application denial of service, and (A10) insecure configuration management. Violations of types A2 and A3, and in some cases A9, fall under access control vulnerabilities. Violations of type A1, A4, and A6 can be viewed as exploiting information flow vulnerabilities. Violations of type A8 can be addressed with measures for ensuring API conformance. Security vulnerabilities of type A5, A7, and A10 are not addressed in our survey.

The rest of this paper is structured as follows. In the next section we discuss access control systems by describing the two major approaches to access control: stack-based access control (SBAC) and role-based access control (RBAC). The next four sections

```
import java.io.*;
import java.net.*;
public class Library {
      private static final String logFileName = "log.txt";
      public static Socket createSocket(String host, int port)
                  throws UnknownHostException, IOException {
            Socket socket = new Socket(host, port);
            FileOutputStream fos = new FileOutputStream(logFileName);
            PrintStream ps = new PrintStream(fos, true);
            ps.print("Socket " + host + ":" + port);
            return socket;
      }
}
```

Figure 1
Source code of Library Java class

cover SBAC vulnerabilities, RBAC vulnerabilities, information flow vulnerabilities, and API conformance vulnerabilities. Each one of these sections consists of two parts. In the first part we survey research results associated with the topic, and in the second part we discuss one or more specific static analysis algorithms that illustrate the techniques used to implement the static analysis method. The last section contains a short summary and final comments.

ACCESS-CONTROL SYSTEMS

Software systems must enforce access control policies for security-sensitive resources and operations. Among the many access-control mechanisms available, SBAC and RBAC have recently gained in popularity and have been adopted for use in both the Java** and .NET Common Language Runtime (CLR) platforms. In this section we describe the basics of SBAC and RBAC and the associated security challenges.

SBAC systems

SBAC was introduced in 1997[8] to enforce access control in multi-component systems. In an SBAC system, when a program attempts to access a restricted resource, the runtime system checks that all callers currently on the thread's stack satisfy a set of permission requirements. SBAC was designed to prevent untrusted code from gaining access to restricted resources by invoking (or being invoked by) more trusted code.

Many SBAC systems grant permissions to code components *declaratively* in a policy database. For example, in Java Standard Edition (Java SE)—formerly known as Java 2, Standard Edition (J2SE**)—an administrator can grant a `FilePermission` to a Java archive (JAR) file, and all the classes in that JAR file will hold that `FilePermission` at runtime.[9] Each permission guards access to a particular resource or resource set and may specify a particular access mode. For example, a `FilePermission` may specify the resource as a file or a set of files and an access mode as any combination of `read`, `write`, `delete`, and `execute`.

When a Java library method attempts to access a restricted resource, an underlying `SecurityManager` calls the `AccessController.checkPermission` service, passing a `Permission` parameter `p` representing the access being attempted. The `checkPermission` function traverses the stack of execution backward, verifying that all calling methods currently on the call stack are authorized to access the resource guarded by `p`.

As an example, the `createSocket` method in the `Library` Java class of ***Figure 1*** opens a network connection on behalf of its client, and on doing so, records the operation in a log file. Both the operations of opening a network connection and writing to a file are security sensitive. Therefore, the client invoking `createSocket` must hold the `SocketPermission` to open a network connection and the `FilePermission` to write to the file system.

Configuring a SBAC security policy can be complicated because permission requirements depend on the dynamic program behavior of various components in various contexts. Typically, practitioners

```
import java.io.*;
import java.net.*;
import java.security.*;
public class AssertingLibrary {
      private static final String logFileName = "log.txt";
      public static Socket createSocket(String host, int port)
                  throws UnknownHostException, IOException,
                              PrivilegedActionException {
            Socket socket = new Socket(host, port);
            PrivWriteOp op = new PrivWriteOp(logFileName);
            FileOutputStream fos = (FileOutputStream)
            AccessController.doPrivileged(op);
            PrintStream ps = new PrintStream(fos, true);
            ps.print("Socket " + host + ":" + port);
            return socket;
      }
}
class PrivWriteOp implements PrivilegedExceptionAction {
      private String logFileName;
      PrivWriteOp (String logFileName) {
            this.logFileName = logFileName;
      }
      public Object run() throws IOException {
            return new FileOutputStream(logFileName);
      }
}
```

Figure 2
Source code of AssertingLibrary Java class

determine the policy configuration through testing, iteratively adding required permissions until tests pass without security exceptions. However, without complete coverage from the test suite, some paths of execution may remain undiscovered until runtime, exposing an application to runtime authorization failures.

The code of Figure 1 illustrates an interesting problem that often arises in SBAC systems. A client program invoking `createSocket` may not be aware that the underlying library will log the network operation to a file. Yet, the program of Figure 1 will not work unless client code invoking `createSocket` on `classLibrary` holds the `FilePermission` required to record the network operation to the log file.

Granting client code the `FilePermission` to log the network operation would constitute a violation of the Principle of Least Privilege,[1] which dictates that any program or user should not be granted more permissions than absolutely necessary. In this example, the library should control the log file; malicious clients could misuse a `FilePermission` to log false data or to erase the contents of the log file. Fortunately, SBAC systems allow portions of library code to be marked as *privilege asserting*. During stack inspection, the runtime system must enforce that privilege-asserting library code holds the necessary permission, but callers of privilege-asserting code need not. In Java, library code can be made privilege asserting by implementing either the `PrivilegedAction` or `PrivilegedExceptionAction` interface. Class `AssertingLibrary` in ***Figure 2*** is a modified version of the class `Library` of Figure 1, with the code performing the file-system operation wrapped in an asserting block.

Although privilege-asserting code is necessary to enforce security policies, unintentional misuse of privilege-asserting mechanisms can introduce security holes. Before making a block *b* of library code privilege-asserting, developers should verify that:

1. The privilege is *necessary*; *b* in fact performs a security-sensitive operation.
2. The privilege is not *redundant*; *b* does not (directly or indirectly) invoke another block of privilege-asserting code before performing a security-sensitive operation.
3. All security-insensitive code that can be moved out of privilege-asserting code is moved,

```
public class StudentBean implements SessionBean {
    private String name, address;
    private Map grades = new HashMap();
    public void setGrade(String c, Character g) {
        grades.put(c, g);
    }
    public void setProfile(String n, String a, Map m) {
        this.name = n;
        this.address = a;
        this.grades = m;
    }
}
```

Figure 3
Source code of the StudentBean EJB class

to avoid future complications from additional security requirements for such code.

4. Block *b* does not contain more than one security-sensitive operation; conversely, it could shield client code from intended permission requirements.
5. Block *b* must respect all *integrity* and *confidentiality* requirements concomitant with the security-sensitive operation.

Deciding which blocks of library code should be made privilege asserting or verifying that existing privilege-asserting code does not violate the preceding security guidelines may be very difficult without an automated tool.[3]

RBAC systems

RBAC was introduced in 1992[10] as a means to restrict access to the operations performed on a system. RBAC has subsequently been adopted by several popular software platforms, such as Java Platform, Enterprise Edition (Java EE)—formerly known as Java 2, Enterprise Edition (J2EE**)—and CLR.[11,12] In RBAC, a *permission* represents the right to perform a restricted operation. A *role* represents a set of permissions that can be granted to users and groups of computer systems. RBAC permissions represent security-sensitive operations, as opposed to security-sensitive data accessed by the system. When a user attempts a restricted operation, the user must possess a role that includes the necessary permission.

Many systems provide both RBAC and SBAC,[11,12] so that system administrators can restrict access to both operations and system resources at the same time. Security compliance management on such systems must account for both models in order to configure access-control policies faithfully.

Several challenges arise when configuring an RBAC security policy:

1. Identifying access control requirements based on operations may be more difficult than identifying requirements based on resources; for example, it is intuitive to recognize field `socialSecurityNumber` as security sensitive, but there is no intuitive way to recognize what resources are accessed by a method called `perform`.
2. In order to deny a role *q* access to some data, an RBAC policy must deny *q* access to all the operations that directly or indirectly access that data; for example, given a `StudentBean` enterprise bean, it is reasonable to impose that the method `setGrade` be accessible to users who have been granted the role of `Professor` and denied to users with role `Student`. It is also reasonable to allow a user with the `Student` role to invoke method `setProfile`, because a student should have the right to modify his or her own profile. However, without having access to the bean's source code, shown in ***Figure 3***, a system administrator may not realize that granting the role of `Student` access to `setProfile` allows students to modify their own grades. This shows that even though RBAC allows configuring a security policy based on operations, it may still be desirable to map those operations to the data they access and to verify that the intended data-based security policy does not conflict with the operation-based security policy that has been configured.[13,14]

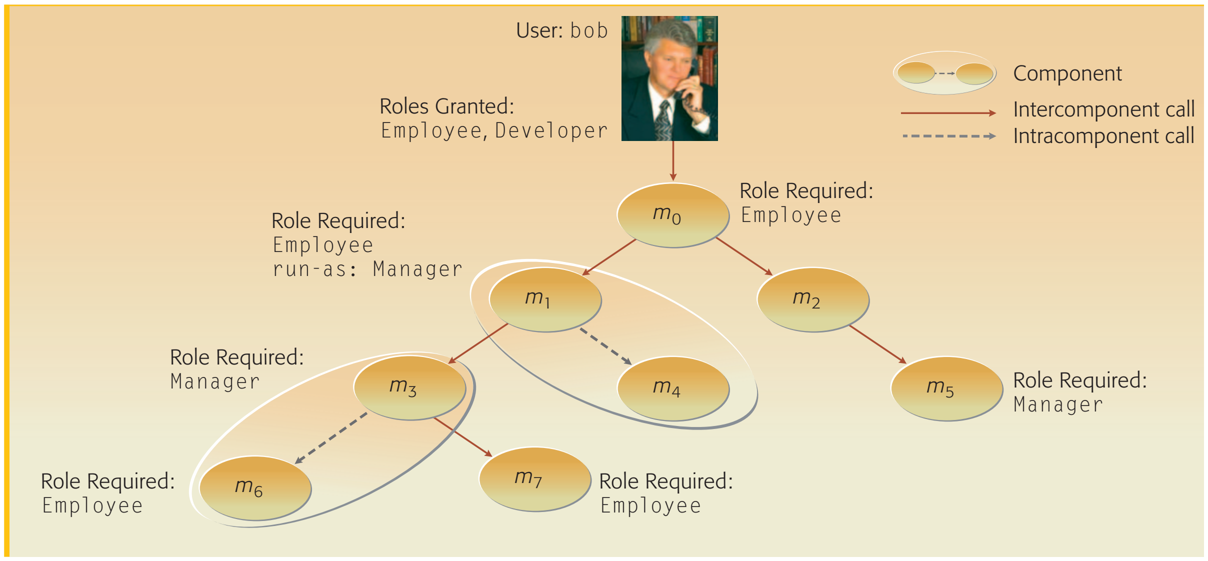

Figure 4
A role-based access control scenario

3. To invoke a particular application entry point *e*, a user must possess all the roles necessary to perform all the operations transitively triggered by *e*; for example, in the scenario of ***Figure 4***, user `bob` has been granted the role of `Employee`, necessary to invoke the application entry point, m_0. In spite of this, the execution of the application will fail because m_0 indirectly triggers the execution of m_5, which requires the role of `Manager`, a role that was not granted to `bob`.
4. No unnecessary roles should be granted to any user; for example, the role of `Developer` is not needed by user `bob` to execute the application of Figure 4. Granting `bob` such a role is a violation of the Principle of Least Privilege.

RBAC configuration is further complicated by *principal-delegation policies*, which dynamically overwrite the roles granted to a user, similar to `setuid` in UNIX**.[15] For example in Figure 4, the component of m_1 has a `run-as` principal-delegation policy that sets the role of the user to `Manager`, and this role is insufficient to execute m_7. A system administrator must also keep in mind that in certain platforms, such as Java EE, RBAC is only enforced when a component is entered; intracomponent access control is not enforced. For example, the invocation of m_7 in the scenario of Figure 4 causes an authorization failure, but that of m_6 does not.

ANALYSIS OF SBAC SYSTEMS

In this section we discuss program-analysis techniques for problem detection and policy validation in SBAC systems, and then focus on a recent algorithm that can be used to infer the SBAC policy of a program, validate an existing policy, and identify candidate code locations for designation as privilege asserting.

Analysis techniques for SBAC systems

Traditionally, software security has been enforced at the operating-system level. Because operating systems have increased in both complexity and size, it has become increasingly difficult to handle security at the operating-system level. Additionally, the operating-system low-level security policies do not lend themselves to application-level access control. The purpose of *language-based security*[16] is to transfer application-level security-policy enforcement to the programming languages used to implement the applications. To achieve this result, language-based security adopts program-rewriting and program-analysis techniques. Program analysis can be used to identify portions of code that do not adhere to the security measures supported by the underlying language. Next, through program rewriting, one can replace those unsecure portions of code with code that complies with the supported security features. A popular language-based security paradigm is to use *type systems*. Rather than requiring the end user to be responsible for security-policy enforcement, type systems transfer this responsibility to the code provider. A program must be written in compliance with the type system, and the end user just needs to type-check the program to ensure that it can be executed safely. Wallach et al.[17] present an approach called Security Architecture Formerly Known as Stack Inspection (SAFKASI). The SAFKASI implementation uses the calculus of Security-Passing Style (SPS) to enforce a form of access control equivalent to traditional stack inspection. They present a formalization of stack introspection, which examines authorization based on the principals currently active in a thread stack at runtime (*security state*). In particular, an SPS is an authorization optimization technique that encodes the security state of an application while the application is executing.[18] Each method is modified so that it passes a security token as part of each invocation. The token represents an encoding of the security state at each stack frame, as well as the result of any authorization test encountered. By running the application and encoding the security state, the SPS explores subgraphs of the comparable invocation graph and discovers the associated security states and authorizations. The goal of this work is to optimize authorization performance. Rather than maintaining the security and code-execution subsystems separately, with distinct semantics and different implementations of push-down stacks, the key intuition behind the SPS calculus is to add the security context as an additional, implicit argument to every method. With SPS, SBAC for a program is achieved by rewriting the program's bytecode before it is loaded, without any need for changing the Java virtual machine or bytecode semantics. Pottier, Skalka, and Smith[19] extend and formalize the SPS calculus via type theory using a λ-calculus, called λ_{sec}.

Jensen et al.[20] focus on proving that code is secure with respect to a global security policy. Their model uses operational semantics to prove the properties, using a two-level temporal logic, and shows how to detect redundant authorization tests.

Bartoletti et al. are interested in optimizing the performance of runtime authorization testing by eliminating redundant tests and relocating others as needed.[21] The reported results apply operational semantics to model the runtime stack. Similarly, Banerjee and Naumann[22] apply denotational semantics to show the equivalence of eager and lazy semantics for stack inspection, provide a static analysis of *safety* (the absence of security errors), and identify transformations that can remove unnecessary authorization tests. A limitation of this approach is that the analyses are restricted to a single thread and require the whole program; incomplete-program analyses are not supported.

Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider[23] describe a system that inlines reference monitors into the code to enforce specific security policies. The objective is to define a security policy and then inject authorization points into the code. This approach can reduce or eliminate redundant authorization tests.

The aforementioned works are specifically designed for Java SE authorization problems and assume that call-graph-construction algorithms are available to translate the theoretical approach into a practical implementation. However, many of the well-known call-graph-construction and data-flow algorithms[24] do not correctly model Java 2, Enterprise Edition (Java EE) cross-component calls and are too conservative to correctly identify authorization requirements.

Both the the Java and .NET CLR platforms support subject-based authentication and authorization. Java Authentication and Authorization Service (JAAS) was introduced as an extension to the Java 2 platform in 1999[25] and became an integral part of the core language starting with Version 1.4. There is little work on static analysis of subject-based authorization, particularly with regard to subject-granted rights analysis.

The MARCO algorithm

In this section, we review static analysis algorithms based on the Mandatory Access Rights Certification of Objects (MARCO) algorithm.[26,27] The MARCO algorithm has been implemented and released as part of the IBM Security Workbench Development for Java (SWORD4J).[28] The Java SBAC architecture mandates that, at the point where `checkPermission` is invoked with a `Permission` parameter `p`, all the code on the execution thread's stack be granted the authorization represented by `p`. MARCO computes permission requirements by

```
public void submitQuery(String userName, String password) {
      String query =
            "SELECT id FROM users WHERE name = '" + userName +
            "' AND password = '" + password + "'";
      execute(query);
}
```

Figure 5
Java method performing an SQL query

modeling the stack inspection mechanism on a computed call graph $G = (N, E)$, representing the execution of the program. For each call graph node n corresponding to a `checkPermission` call, MARCO identifies a set of abstract `Permission` objects that may flow to n as a parameter. The tool solves a data-flow problem, propagating sets of `Permission` abstract objects backward in G, until a fixed point is reached.[29]

In Java, permission requirements propagated on the stack by a call to `doPrivileged` do not propagate beyond the predecessors of the `doPrivileged` node itself. MARCO handles this precisely by killing data-flow propagation of upwardly exposed permissions at `doPrivileged` nodes, and computing permission requirements for `doPrivileged` predecessors as a post-pass to the fixed-point computation.

When the data-flow algorithm just described terminates, each node $n \in N$ is mapped to a set $\Pi(n)$ of `Permission` abstract objects. $\Pi(n)$ over-approximates the permissions required to execute the method modeled by n.

MARCO can help detect which code could be made privilege asserting, while minimizing the risks of violating the Principle of Least Privilege. A static analysis of Java's class-loading behavior can identify intercomponent calls. If $e = (m, n)$ is an intercomponent edge and $\Pi(n) \neq \emptyset$, then the call represented by e can potentially lead to a security-sensitive resource access. Such a call is a candidate for a privilege-asserting block.

For example, e may be the edge resulting from calling the constructor of `FileOutputStream` from method `createSocket` in Figure 1. Figure 2 shows how to wrap the `FileOutputStream` constructor call into a privilege-asserting block.

As described, MARCO's analysis of permissions on intercomponent edges suffices to determine component code that is a candidate for privilege assertion and to identify the associated permissions. Additionally, the tool can recommend privilege-asserting code locations that lie closest to the authorization checks. This information can help minimize the risk of introducing unnecessary access privileges. Code changes during development or maintenance can create unnecessary or redundant `doPrivileged` calls. For example, after code modifications, a call to `doPrivileged` that was originally necessary may no longer trigger an authorization check. A redundant `doPrivileged` call may result from poor code design or during component integration, where a call to `doPrivileged` becomes redundant because other `doPrivileged` calls now dominate the authorization check. MARCO's algorithm can identify unnecessary or redundant calls to `doPrivileged` by simply detecting any `doPrivileged` node d such that $\Pi(d) = \emptyset$.

If code does not require authorizations, it is poor security practice to include it in privilege-asserting code.[3] For example, the following instruction in the `run` method of ***Figure 5*** appears in a privilege-asserting block even though it does not access restricted resources:

```
System.out.println("User: " + userName
  + "; Host: " + host + "; Port: " + port);
```

Such code should be moved to a non-privilege-asserting block of code even though, in this case, $\Pi(d) \neq \emptyset$. MARCO can statically detect statements that have unnecessarily been inserted into otherwise valid privilege-asserting blocks. Specifically, let d be a `doPrivileged` node and r its `PrivilegedAction` or `PrivilegedExceptionAction` `run` successor. If $\Pi(r) \neq \emptyset$ and r has a successor node n such that $\Pi(n) = \emptyset$, then the method invocation represented by n can be safely moved to non-privilege-asserting code.

The algorithm as described so far does *not* account for *information-flow violations* that can arise from privilege-asserting code. For example, both the `Socket` and the `FileOutputStream` constructor calls in Figure 1 would be reported as potential candidates for becoming privilege asserting, even though wrapping the `Socket` constructor call in a privilege-asserting block would constitute both an integrity and confidentiality violation. Privilege-asserting code should adhere to the integrity and confidentiality requirements.

Clearly, the precision of these static security analyses depends on the precision of the underlying static analyzer. The MARCO tool is based on an a context-sensitive static analyzer that distinguishes invocations of the same method based on their *calling contexts*, consisting of the allocation sites of receivers and parameters. This context-sensitivity policy, unfortunately, does not always suffice to disambiguate different calls to the same method (for example, if that method is static and takes no parameters). This can lead to overly conservative results. Work is underway on a set of open research problems concerning more precise, yet scalable algorithms to increase precision of stack inspection analysis.

In addition to static analysis, permission and privilege-asserting code requirements for a library can also be identified using dynamic analysis. Typically, the developer tests the library code with sample client code, iteratively determining sufficient permissions to execute the test cases. Usually, the client code is granted only a limited number of permissions, while the library is granted sufficient permissions, such as `AllPermission`. Next, it is necessary to take note of all the `SecurityException`s generated when running the test cases, and to distinguish between two categories of `SecurityException`s: (1) those due to the client code's attempting to access some restricted resources through the library without adequate authorizations, and (2) those due to the library's attempting to access some restricted resources on its own without using privilege-asserting code. Eliminating a `SecurityException` of type 2 requires inspecting the library source code, identifying which portion accesses the protected resource, and making that portion of code privilege asserting. A `SecurityException` of type 1 can be eliminated by requiring that client code hold necessary permissions, but this operation must be performed cautiously because granting permissions to the client could hide `SecurityExceptions` of type 2. Manually performing this process is difficult, tedious, and error prone. After modifying the library code or the client security policy, the developer must rerun the test cases. This process must be repeated, possibly many times, until no more authorization failures occur. Additionally, privilege-asserting requirements in the library code may remain undiscovered during testing due to insufficient test coverage, which makes production code potentially unstable.

ANALYSIS OF RBAC SYSTEMS

This section presents a survey of program analysis algorithms for RBAC systems and then focuses on two recent algorithms that have been designed and developed to validate RBAC policies.

Analysis techniques for RBAC systems

The concept of RBAC was introduced for the first time by Ferraiolo and Kuhn,[10] who identified the need for security policies based on the roles that a user has in an organization. Work on building and analyzing models and implementations for RBAC has concentrated on complex architectures.[30] Surprisingly, few approaches for analyzing RBAC mechanisms have been suggested. Schaad and Moffett[31] used the Alloy specification language[32] for modeling the RBAC96 access model and the Alloy Constraint Analyzer (Alcoa)[33] for checking desirable properties of such models, such as separation of duties assigned to roles.

Extensible Markup Language (XML) documents are often used by Web applications. Several mechanisms and frameworks for specification and enforcement of access policies for XML documents have been proposed.[34,35] Such mechanisms are flexible in the sense that they prohibit or allow access to specific individual elements in XML documents. Recently, Murata, Tozawa, Kudo, and Satoshi[36] proposed a static analysis approach based on finite state automata that alleviates the burden of enforcement of such specifications at runtime. Another positive side effect of this work is faster execution of queries over XML documents in some situations.

In the area of Web applications, a number of testing and static analysis techniques have been proposed, but they have concentrated primarily on the problem of control and information flow between static and dynamic HyperText Markup Language (HTML) pages used by Web applications. For example, Ricca and Tonella[37] introduced a Unified Modeling Language** (UML**) model for Web applications that is useful for structural testing. However, this model concentrates on links between Web pages and interactive features of Web applications, such as HTML forms, and does not provide support for distributed object components.

Clarke et al.[38] address the confinement problem of Enterprise JavaBeans** (EJB**) objects. This problem arises in situations where direct references to EJB objects or other server-side distributed objects are returned to clients. Such references allow clients to use EJB objects directly, without going through the indirection of EJB interface objects. As a result, the EJB RBAC model can be circumvented. The work by Clarke et al. identifies the possible ways in which confinement of EJB objects can be breached, and defines simple programming conventions that, if observed, support inexpensive static analysis able to detect confinement breaches or verify that no confinement breach is possible for a given set of enterprise beans.

The ESPE algorithm

RBAC systems present similar challenges to those facing SBAC systems. Specifically, for RBAC, it is desirable to identify an application's role requirements, evaluate an existing RBAC policy to detect if it is too permissive or too restrictive, and detect if the RBAC policy restricts access on data consistently.

RBAC policies restrict access to operations (or methods) rather than data or resources. In order to invoke a particular entry point to an application, a user needs to be granted all the roles required to invoke all the methods transitively invoked starting at that entry point.

The Enterprise Security Policy Evaluation (ESPE) algorithm by Pistoia et al.[26,39] casts the problem of evaluating an RBAC policy to a data-flow problem. The execution of a Java EE application protected with a set of roles R is modeled as a call graph $G = (N, E)$. The goal is to map each node $n \in N$ to $v(n)$, a propositional logic statement in conjunctive normal form, corresponding to the roles necessary to invoke the method m represented by n at runtime. The statement $v(n)$ can be represented as an element of $\mathcal{P}(\mathcal{P}(R))$, where the $\mathcal{P}$ operator maps a set to its powerset. Specifically, if m has been restricted with roles $r_1, r_2, \ldots, r_k \in R$, initially $v(n) := \{\{r_1, r_2, \ldots, r_k\}\}$, modeling the property that a user u initiating any execution traversing m has to show possession of at least one role in the set $\{r_1, r_2, \ldots, r_k\}$, which can also be stated by saying that the roles granted to u must be compatible with the logical expression $r_1 \vee r_2 \vee \ldots \vee r_k$. If m has not been restricted with any role, then $v(n) := \emptyset$. Subsequently, the elements of $\mathcal{P}(\mathcal{P}(R))$ associated with each node are recursively propagated backward in the call graph, performing set unions at each node, until a fixed point is reached.[29] At the end of this process, each node n will be mapped to a set $v(n) = \{R_1, R_2, \ldots, R_h\} : R_i = r_{i1}, r_{i2}, \ldots, r_{ik_i}\} \subseteq R, \forall i = 1, 2, \ldots, h$, meaning that the set of roles granted to u must evaluate to `true` for the proposition $(r_{11} \vee r_{12} \ldots \vee r_{1k_1}) \wedge (r_{21} \vee r_{22} \vee \ldots r_{2k_2}) \wedge \ldots \wedge (r_{h1} \vee r_{h2} \vee \ldots \vee r_{hk_h})$.

ESPE detects a potentially insufficient RBAC policy. For example, in Figure 4, user `bob` has been granted the `Employee` and `Developer` roles, which allow `bob` to invoke the application's entry point m_0, restricted with the `Employee` role. However, the roles granted to `bob` will not allow `bob` to execute method m_5, which will be invoked as part of the execution initiated by m_0. Additionally, the analysis can help detect if the user has been granted unnecessary roles. For example, the `Developer` role is unnecessary for user `bob` in the scenario of Figure 4.

RBAC systems support principal-delegation policies to override the roles granted to a user. Under a principal-delegation policy associated with a component, all the methods subsequently traversed execute with the authority as specified by the principal-delegation policy. For example, consider the Java EE `run-as` policy associated with the component of m_1 and m_4 in Figure 4. This policy overrides the set of roles granted to user `bob` with the singleton $\{$`Manager`$\}$, which will suffice to execute method m_7. To model principal-delegation policies, the data-flow algorithm described above must be augmented by killing,[29] at each component that enforces a principal-delegation policy, any element of $\mathcal{P}(\mathcal{P}(R))$ propagated backward. This augmented algorithm detects whether a principal-delegation policy is insufficient by simply comparing, at each component setting a principal-delegation policy, the roles set by the policy with the roles propagated backward in G.

The SAVES algorithm

Note that by restricting access to methods, an RBAC policy induces an implicit access control policy on the data and resources accessed by those methods. Often, an RBAC system administrator may not be aware of how an application's methods manipulate underlying data and resources. This may lead to inconsistent access control policies, whereby two methods access the same data and resources in the same mode (such as `setGrade` and `setProfile` in Figure 3), but with different access control restrictions according to a misconfigured RBAC.

`StudentBean` in Figure 3 shows two methods, `setGrade` and `setProfile`, both accessing the security-sensitive data accessible through the `grades` field. As we observed, preventing users in the role of `Student` from executing `setGrade` and allowing them to execute `setProfile` leads to an inconsistent policy, since `setProfile` allows replacing the value of the `grades` field.

Centonze et al.[14] propose a theoretical foundation for RBAC to identify such problems and implement a static analyzer for RBAC consistency validation. Their implementation, called Static Analysis for Validation of Enterprise Security (SAVES), considers abstract memory locations corresponding to EJB fields. Intuitively, fields represent the granularity by which an RBAC policy allows control of restricted data. SAVES distinguishes when a security-sensitive field f is accessed in read or write mode by a method m. An RBAC policy for a program p can be seen as a function $\mu : R \rightarrow \mathcal{P}(M)$ where R is the set of roles defined for p and M is the set of methods executed by p. An RBAC policy μ is said to be *location inconsistent* if there exist $m \in M$ and $q \in R$ such that q has been denied access to m, but the same fields that q could have accessed through m are accessible through other methods whose access has been granted to q. A location inconsistency indicates that the intent of the security policy is unclear. Centonze et al. have proved that a method-based RBAC policy μ has an equivalent location-based RBAC policy if and only if μ is location consistent.

Given a program p with an RBAC policy μ, SAVES performs a field-sensitive and context-, flow-, and path-insensitive interprocedural mod-ref analysis to determine the sets of fields read and written by each method, and detects potential location inconsistencies for μ. If no location inconsistency is detected, SAVES can report the location-based RBAC policy equivalent to μ. Experimental results reported in Reference 14 show that the analysis is effective for a number of Java EE applications.

INFORMATION FLOW

In this section we discuss techniques that identify information-flow vulnerabilities in software systems and focus on integrity and confidentiality as the two main types of vulnerabilities. We discuss the research work to date in this area and describe in more detail some recent contributions.

The data manipulated by a program can be tagged with security levels,[40] which naturally assume the structure of a partially ordered set. Under certain conditions, this partially ordered set is a lattice.[41,42] In the simplest example, this lattice only contains two elements, indicated with *high* and *low*. Given a program, the principle of *non-interference* dictates that low-security behavior of the program not be affected by any high-security data.[43] Assuming that *high* means *confidential* and *low* means *public*, then verifying that no information ever flows from higher to lower security levels (unless that information has previously been *declassified*) is equivalent to verifying *confidentiality*. Conversely, if *high* means *untrusted* and *low* means *trusted*, then verifying that no information ever flows from higher to lower security levels (unless that information has previously been *endorsed*) is equivalent to verifying *integrity*.

Vulnerabilities such as those caused by nonvalidated input and injection flaws constitute integrity violations. Globally, declassification and endorsement are also known as *downgrading* because they allow high-level security information to be used in low-level security contexts.

Integrity

The data that originate from an untrusted source is referred to as *tainted*.[44] Tainted data and the variables that hold or reference it can be maliciously used for *overwrite attacks*,[44] which may consist, for example, of overwriting the name of a file or jump

```
import java.net.*;
import java.security.*;
public class TaintedLibrary {
      public static Socket createSocket
            (final String host, final int port, final String userName)
                  throws Exception {
            Socket s;
            PrivOp op = new PrivOp(host, port, userName);
            try {
                  s = (Socket) AccessController.doPrivileged(op);
            }
            catch (PrivilegedActionException e) {
                  throw e.getException();
            }
            return s;
      }
}
class PrivOp implements PrivilegedExceptionAction {
      private String host, userName;
      int port;
      PrivOp(String host, int port, String userName) {
            this.host = host;
            this.port = port;
            this.userName = userName;
      }
      public Object run() throws Exception {
            System.out.println("User: " + userName + ";
                  Host: " + host + "; Port: " + port);
            return new Socket(host, port);
      }
}
```

Figure 6
Source code of TaintedLibrary Java class

address. Sometimes, however, it is necessary to use a tainted variable in a trusted environment when restricted resources are accessed. In such cases, the data can be endorsed by performing sanity checks on it before using it in restricted operations.[45] Sanity checks are usually domain or component specific. For example, the SQL query in Figure 5 needs to perform a security-sensitive operation based on input coming from a user. If the program did not validate user inputs, a malicious client could call `submitQuery` passing "'' OR '1'='1'" as the value of the `password` parameter, causing the password check in the SQL query to become `password="'' OR '1'='1'"`, which always succeeds.

In SBAC systems, integrity issues commonly arise in the context of privilege-asserting code. The requirement for integrity establishes that no value defined in untrusted code should ever be used inside privilege-asserting code unless that value has been previously endorsed. In an SBAC system, a tainted variable is not necessarily a security problem. It may constitute a security problem if it is also a *privileged* variable, meaning that it is used inside privilege-asserting code.[3] Even a privileged tainted variable is not necessarily a security problem. In fact, it is appropriate to distinguish two types of privileged tainted variables: if a privileged variable is used to access a restricted resource, that variable is called *malicious*; otherwise, it is called *benign*. Because authorization checks are not performed beyond the stack frame invoking `doPrivileged` in Java or `Assert` in CLR, an untrusted client application could exploit a malicious variable to have the privilege-asserting code access arbitrary restricted resources on its behalf.

Consider, for example, the `TaintedLibrary` class shown in ***Figure 6***. Both `host` and `port` are tainted variables because an untrusted client can arbitrarily set them. The fact that they are used inside privilege-asserting code to open a socket makes them malicious and constitutes a potential security risk. An untrusted client, with no `SocketPermission`, can invoke `createSocket` on the trusted library and have the library open an arbitrary socket connection on its behalf. Conversely, variable `userName`, though tainted and privileged, is benign because its value is not used to access a restricted resource. In Figure 2, variable `logFileName` is not tainted because its value cannot be set by a client application.

In RBAC systems, integrity violations can arise due to incorrectly specified principal-delegation policies. A principal-delegation policy overwrites the roles granted to the executing user with the roles specified by the policy itself. From that point on, the execution of all the cascading calls will be performed as if the user had been granted the roles specified by the principal-delegation policy. A principal-delegation policy is often used to elevate, in special circumstances, the authority of the users executing the program, without making it necessary to grant those users roles that would allow them to execute unintended operations. For example, in the scenario of Figure 4, the principal-delegation policy associated with the component of methods m_1 and m_4 assigns user `bob` the role of `Manager`, which is required to invoke m_3. This principal-delegation policy has made it unnecessary to grant user `bob` the role of `Manager`, which could have been misused. The integrity requirement establishes that no value defined by the user be used after the user's authority has been elevated unless that value has been previously endorsed.

Confidentiality

Confidentiality issues in SBAC systems also arise in the context of privilege-asserting code. The requirement for confidentiality establishes that a value flowing out of a privilege-asserting block of code b should remain confined inside the component of b unless a check has been performed to verify that the value can be safely released. This requirement can also be used when deciding whether it is appropriate to make a block of code privilege asserting. For example, in the `Library`, class shown in Figure 1, it is appropriate to make the `FileOutputStream` constructor call privilege asserting (as done later in the code of Figure 2), not only because there is no integrity break but also because the constructed `FileOutputStream` object remains confined inside the `Library` class itself. Conversely, the call to the `Socket` constructor should not be made part of privilege-asserting code. Figure 6 shows that if the code to the `Socket` constructor were made privilege asserting, there would be not only an integrity violation but also a confidentiality violation, because the constructed `Socket` object is released to the potentially untrusted client that invoked `createSocket`.

In RBAC systems, confidentiality violations can arise due to incorrectly specified principal-delegation policies, as is the case for integrity violations. When a principal-delegation policy elevates the roles of a user, all the data defined inside the code executed under that policy should remain confined inside that code. For example, in the scenario of Figure 4, any value defined or computed in the component of m_3 and m_6 has been originated under the authority of the role `Manager`. A flow of information which makes that value accessible to the component of m_1 and m_4 might violate the confidentiality requirement by allowing users with the role of `Employee` to access information intended only for users in the role of `Manager`.

Analysis techniques for information flow

While the accurate detection of information flow is undecidable,[46] static analysis can be used to over-approximate information flows in a program in order to ensure information-flow security. In this section, we present a survey of algorithms for checking information-flow security.

The basic idea behind using static analysis for detecting information flow is to statically check that flow of information between variables in a program is consistent with the security labeling of variables. Each variable is labeled with a certain security level. If a variable x is used to derive, or influence, the value of another variable y, there is potential information flow from x to y. The flow is permissible under a security policy only if the policy allows the security level of x to flow to the security level of y. Formally, let $(S, \leq)$ be a lattice of security levels. For security levels a and b, $a \leq b$ means that it is allowed for information of level a to flow into level b. If the lattice is modeling confidentiality, then a is no more secret than b; if the lattice is modeling integrity, then a is at least as trusted as b. We denote the security level of a variable x by $dom(x)$. There exists an *explicit* flow from x to y if the value of x is assigned to y in the program, as in $y := x$. There is an *implicit* flow from x to y if the value of x is used to evaluate the outcome of a conditional, which then controls an assignment to y, as in `if` $(x > 0)$ `then`

$y := 1$ `else` $y := 0$ `endif`. We denote an explicit or implicit flow from x to y as $x \Rightarrow y$. Denning and Denning's certification of programs for secure information flow[40] uses the following proposition:

A program respects the security policy implied by the lattice $(S, \leq)$ when, for any flow $x \Rightarrow y$, it must be the case that $dom(x) \leq dom(y)$.

To enforce a program's information-flow security, this property, which defines a sufficient but not necessary condition, must be verified for each flow in the program.

Goguen and Meseguer[43] have given a more general notion of information-flow security based on non-interference. Informally, the non-interference principle says that an observer must not be able to see variation in "low-security" outputs that is derived from variation in "high-security" inputs (so the observer cannot make any inference on high-security information). Suppose a computation undergoes the following transition between input and output states: $(H_1, L_1) \rightarrow (H_2, L_2)$, where $H_1$and H_2 are high-security components and L_1 and L_2 are low-security components of the states. For this computation to be secure, it must be the case that for any other value of high-security component, the low-security output does not change: $(H_1', L_1) \rightarrow (H_2', L_2)$. Remember that the interpretation of the terms "high security" and "low security" depends on the problem being solved: for confidentiality, *higher* and *lower security* mean more and less secret; for integrity, they mean less and more trusted, respectively. For example, let h be a secret variable and l be a public variable. Then the program `input` h; `if` $(h > 0)$ `then` $l := 1$ `else` $l := 0$ `endif`; `output` l violates non-interference, because different initial values of high input h can result in different final values of the low output l.

Formally, let $\rightsquigarrow \subseteq S \times S$ be an *interference* relation on security levels. If $a \rightsquigarrow b$, the security level a is allowed to interfere with the security level b, in the sense that it can impact observable values in level b. Generally, we are interested in the complement of this relation, $a \not\rightsquigarrow b$, which prohibits a from interfering with b. Note that $\rightsquigarrow$ must be reflexive, but in general it need not be transitive.

While Goguen and Meseguer presented non-interference in a more abstract setting of "actions," our presentation here is limited to program statements. We label each input or output statement x by its security level $dom(x)$ (generalizing *dom* to apply to statements). The security level of an input statement is the security level of the input value being provided to the computation. The security level of an output statement is the security level of the variable being made visible external to the computation.

Let *run* be the state update function $State \times Statement \rightarrow State$. Let *purge* be a function that, given a trace of statements α, removes from it all such statements x whose security level must not interfere with a given security level υ; it is defined as follows: $purge(x \cdot \alpha, \upsilon) := x \cdot purge(\alpha, \upsilon)$ if $dom(x) \rightsquigarrow \upsilon$, and $purge(x \cdot \alpha, \upsilon) := purge(\alpha, \upsilon)$ otherwise. Let s_0 be the initial state of the program. Then, the security criterion can be stated as follows:

A program respects the security policy implied by a non-interference relation $\not\rightsquigarrow$ if for any sequence of statements α in an execution ending in a statement z:

$$output(run(s_0, \alpha), z) \\ = output(run(s_0, purge(\alpha, dom(z))), z)$$

That is, the output produced at statement z must be identical even if all such previous statements have been purged whose security level must not interfere with the security level of z. In the previous example, on any run in which `input` h is purged (and the default value of h, assumed 0, is in effect), the output is 0 for l; whereas a nonpurged run with input value of 1 or higher would produce the output of 1 for l. For sake of contrast, consider a slightly modified example: `input` h; `if` $(h > 0)$ `then` $l := 1$ `else` $l := 0$ `endif`; $l := 2$; `output` l. In this example, even if the statement `input` h is purged from any run, the output for l is 2, which is the same as for any nonpurged run, and so the conditions for non-interference are satisfied.

Non-interference is a more general criterion than our first criterion of secure information flow, in that it only constrains the projection of outputs produced from actual statement sequences; it does not constrain implicit or explicit flow at each statement. Note that Denning and Denning's criterion would have rejected the modified example above because an implicit flow violating the security-lattice rule does exist. Volpano, et al.[47] have shown a type-based algorithm that certifies implicit and explicit flows similarly to the first criterion and also ensures non-interference. Non-interference is traditionally the technical criterion used for proving correctness of security analysis algorithms or type systems. However, it is also harder to check non-interference directly.

Secure information flow is important in the context of Web applications. A number of approaches for reasoning about flow of information in systems with mutual distrust have been proposed. For example, Myers and Liskov[48] use static analysis for certifying information control flow and avoiding costly run-time checks.

In Java and CLR, information flow issues are particularly relevant with privilege-asserting code. Privilege-asserting code has historic roots in the 1970s. The Digital Equipment Corporation (DEC) Virtual Address eXtension/Virtual Memory System (VAX/VMS) operating system had a feature similar to the `doPrivileged` method in Java 2 and the `Assert` method in CLR. The VAX/VMS feature was called *privileged images*. Privileged images were similar to UNIX `setuid` programs,[15] except that privileged images ran in the same process as all the user's other unprivileged programs. Thus, they were considerably easier to attack than UNIX `setuid` programs because they lacked the usual separate process and separate address-space protections. One example of an attack on privileged images is demonstrated in a paper by Koegel, Koegel, Li, and Miruke.[49]

The notion of tainted variables as vehicles for integrity violations became known with the Perl language. In Perl, using the `–T` option allows detecting tainted variables.[50] Shankar, Talwar, Foster, and Wagner present a tainted-variable analysis for CQual using constraint graphs.[51] To find format string bugs, CQual uses a type-qualifier system with two qualifiers: *tainted* and *untainted*.[52] The types of values that can be controlled by an untrusted adversary are qualified as being tainted, and the rest of the variables are qualified as untainted. A constraint graph is constructed for a CQual program. If there is a path from a tainted node to an untainted node in the graph, an error is flagged.

Newsome and Song propose a dynamic tainted-variable analysis that catches errors by monitoring tainted variables at runtime.[44] Data originating or arithmetically derived from untrusted sources, such as the network, are marked as tainted. Tainted variables are tracked at runtime, and when they are used in a dangerous way, an attack is detected.

Ashcraft and Engler[45] also use tainted-variable analysis to detect software attacks due to tainted variables. Their approach provides user-defined sanity checks to untaint potentially tainted variables.

Pistoia[26] proposes an algorithm based on program slicing to automatically discover malicious tainted variables in a library. His approach can be used to decide whether a portion of library code should be made privileged or not.

Hammer, Krinke, and Snelting's algorithm

Snelting et al.[53] make the observation that program dependence graphs (PDGs) and non-interference are related in the following manner. Consider two statements s_1 and s_2. If $dom(s_1) \not\rightsquigarrow dom(s_2)$, then, in a security-correct program, it must be the case that $s_1 \notin backslice(s_2)$. Here, *backslice* is the function that maps each statement s to its *static backward slice*, consisting of all the (transitive) predecessors of s along control- and data-dependence edges in the PDG. Based on this observation, Hammer et al.[54] have presented an algorithm that checks for non-interference: for any output statement s, it must be the case that *backslice*(s) contains only statements that have a lower security label than s. Hammer et al. also refine slices with path conditions to get higher accuracy, but we elide the details here. Note that even PDG-based computation, as in the above technique, is only an approximation to the ideal of non-interference. We assume that the reader is familiar with PDGs and slicing, as these are standard concepts in program analysis. Here, we present just an example to illustrate the idea in the context of information-flow security. Consider the program shown in ***Figure 7***. (***Figure 8*** shows the PDG for this program.) Edges that are in the backward slice from the output statement are shown in red. It is clear that the backward slice of the output statement includes the higher-security input statements, which must not interfere with the output statement (assume that $\rightsquigarrow$ coincides with $\leq$ in the security lattice shown in Figure 7). Note also, that a more sophisticated program verifier may be able to reason that the outcome of the branch at line 10 is always false. An ideal checker for non-interference would not report a

```
Security labeling:                                              Security lattice

h1, topSecretFile:  TOPSEC;                               PUB ≤ CONF ≤ TOPSEC
h2, confidentialFile:  CONF;                                    LO1 ∨ LO2 ≤ PUB
l1:  LO1;
l2:  LO2;
m, publicFile;  PUB;

Program Statements                             Statement-wise Certification Checks

1:   input h1 from topSecretFile;                    dom(topSecretFile) ≤ dom(h1)
2:   input h2 from confidentialFile;              dom(confidentialFile) ≤ dom(h2)
3:   if (h1 op h2) {
4:     l1 = 1;
5:     l2 = 0;
6:   } else {
7:     l1 = 0;
8:     l2 = 1;
9:   }                                         dom(h1) ∨ dom(h2) ≤ dom(l1) ∧ dom(l2)
10:  if (l1 == l2) {
11:    m = true;
12:  } else {
13:    m = false;
14:  }                                                     dom(l1) ∨ dom(l2) ≤ dom(m)
15:  output m to publicFile;                                 dom(m) ≤ dom(publicFile)
```

Figure 7
Example illustrating information-flow algorithms

security violation. It should be noted that PDG-based algorithms, such as the one above, have not been shown to scale to large applications, of the size of several hundred-thousand lines of code. Flow-sensitive approaches, such as Denning and Denning's algorithm and several type-system-based algorithms, can scale better. The latter also enjoy the advantage of compositional analysis, which means that parts of programs can be analyzed in isolation, which is generally hard to do in PDG-based analysis.

Dealing with heap data

In this section, we describe an algorithm for performing non-interference analysis in the presence of heap-allocated data structures, which are very common in Java and other object-oriented languages. The analysis of heap-based objects is an entire area of research in itself, and even a brief survey is beyond the scope of this article; a recent research paper[55] gives a good overview of the current state of the art. In the present paper, we focus on a recent algorithm by Livshits and Lam[56] that is engineered to work well for tainted-variable analysis of large Java applications.

Livshits and Lam's analysis requires prior computation of a specific heap analysis called flow-insensitive *points-to* analysis. This analysis computes a "may point to" relation over a program, where $pointsTo(o_1.f, o_2)$ means that the field f of the object named o_1 might refer to the object named o_2 in some execution of the program. A points-to relation is also computed for local variables: $pointsTo(v, o)$ means that the local variable v might refer to the object named o. The relation $pointsTo(v.f, o)$ holds if there exists an o' such that $pointsTo(v, o')$ and $pointsTo(o'.f, o)$. The *pointsTo* relation is the same for the entire program, ignoring the control flow of the program. (By contrast, the PDG-based algorithm of Hammer et al. handles heap objects in a flow-sensitive manner, albeit at much higher cost.) We refer the reader to a paper by Whaley and Lam[57] that describes the details of the heap analysis used by Livshits and Lam.

Tainted-variable analysis is an integrity problem in which we are interested as to whether less-trusted data obtained from the user might influence other data that the system trusts. Clearly, to do this analysis, one needs to identify sources and sinks of possibly tainted data. For Java, this amounts to identifying methods that originate a tainted value and methods that use a possibly tainted value. The Livshits and Lam algorithm gets this information

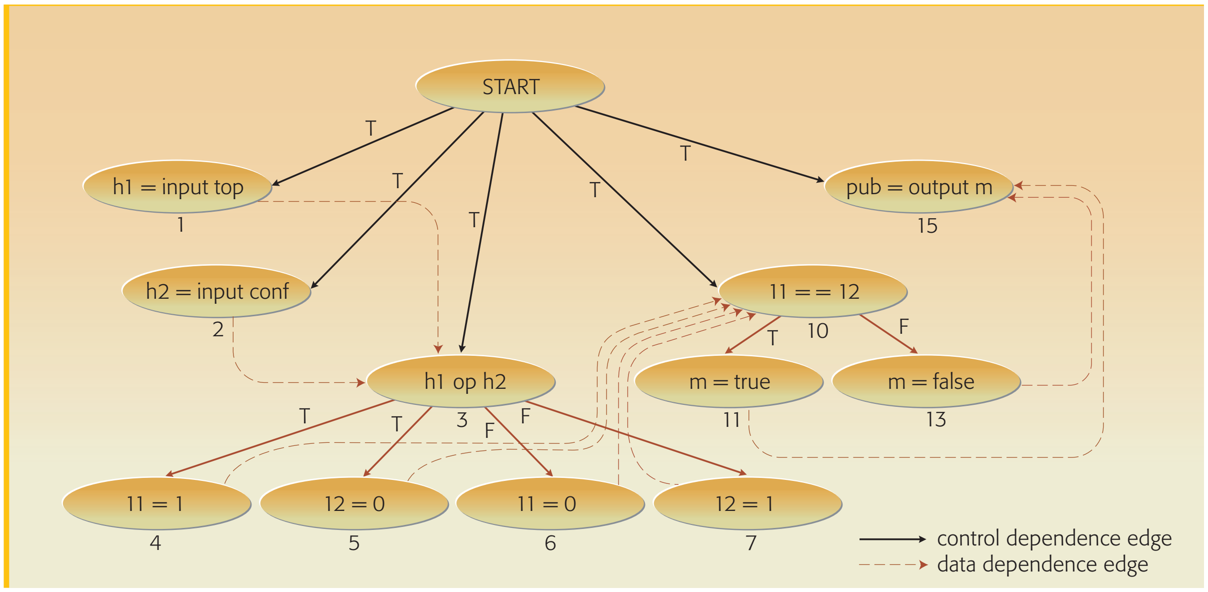

Figure 8
Program dependence graph for example in Figure 7

from programmer-supplied descriptors. A *source descriptor* is of the form ⟨*m*, *p*, *a*⟩, where *m* is a method, *p* is the position of a certain argument, and *a* is a possibly empty field-access path from *m*'s *p*-th parameter. A non-empty access path is needed when a field of a parameter points to the object of interest. The position of the return value is –1. For example, a method `getParameter (String username)` has the following descriptor: ⟨`getParameter`, –1, ϵ⟩, which means that the return value from `getParameter` refers to the tainted source object. In the same way, *sink descriptors* can be given for methods that need a trusted value. For example, a method `executeQuery (String query)` has the descriptor ⟨`executeQuery`, 1, ϵ⟩, meaning that the `query` parameter points to an object that the method trusts.

Suppose there is a program variable v_1, such that $v_1.a$ refers to a source object corresponding to some source descriptor ⟨*m*, *p*, *a*⟩. Suppose another variable v_2 is such that $v_2.a'$ refers to a sink object corresponding to some sink descriptor ⟨*m*′, *p*′, *a*′⟩. If there exists an object *o*, such that *pointsTo*($v_1.a$, *o*) and also *pointsTo*($v_2.a'$, *o*), then the algorithm conservatively assumes that there is a possible flow from the source to the sink. Note that there is no indication from the *pointsTo* relation that the two specific *pointsTo* facts used in the above deduction hold during the same program execution.

Sometimes, there is a need to pass on taintedness between objects handled by a library call. For example, the `StringBuffer.append (String instring)` function returns an output string derived from an input string. Livshits and Lam require "derivation descriptors" for such methods. A *derivation descriptor* is of the form $\langle m, p_s, a_s, p_d, a_d \rangle$, with the meaning that in method *m*, the object refereed by its p_d-th parameter (along access path a_d) is derived from its p_s-th parameter (along access path a_s). For example, the descriptor for `append` is ⟨`append`, 1, ϵ, –1, ϵ⟩. In the presence of such methods, the value-flow computation has to account for (transitive) flow from a tainted source to a derivation method's input, and from a derivation method's output to a sink.

API CONFORMANCE

In this section we describe the security vulnerabilities related to violations of API specifications, present a brief survey of work in the area of automatic identification of API violations, and discuss in detail some recent program-analysis approaches.

Enterprise software platforms provide a number of APIs for system security services. These services often present nontrivial interfaces with complex and unenforced usage constraints. Misuse of such

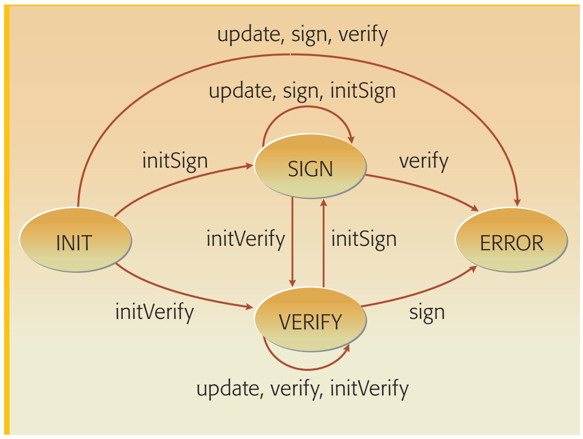

Figure 9
Partial typestate automaton for java.security.Signature

interfaces are not detected by the compiler and can lead to unintended security holes. Some API conformance rules can be checked with trivial syntactic code scanning, commonly provided by tools such as FindBugs,[58] PMD,[59] and Rational CodeReview.[60] For example, Java EE practices for security may include rules such as "Don't call `java.security.Policy.setPolicy`," or "Classes that extend `java.security.Permission` should be final."

Other APIs depend on more complex temporal safety properties. Consider, for example, the public-key cryptography services from Java's `java.security` package. In particular, class `java.security.Signature` provides digital signature functionality. A client must invoke the methods of a `Signature` object according to API conformance rules described in the class documentation. If the client fails to follow these conformance rules, the application will receive a security-related exception. Note that the conformance rules are not expressed in the programming language, and are not checked by the compiler.

These API conformance rules can be expressed as temporal safety properties in a deterministic finite state automaton, or a *typestate specification.*[61] Typestate specifications can encode correct usage rules for many security-related libraries and interfaces.[62,63] For example, ***Figure 9*** presents a partial typestate specification of a `Signature` object. This partial specification represents a subset of the permitted behaviors for a `Signature` object (we omit some behaviors for simplicity). For this typestate specification, if a `Signature` object moves to the `ERROR` state, then a runtime exception may be thrown at program runtime. Typestate verification represents a classic problem in program verification.

Analysis techniques for API conformance

Strom and Yemini introduced the typestate model to describe temporal safety properties.[61] In subsequent years, a large number of efforts have focused on verifying or checking typestate properties for imperative programming languages such as C or Java. Previous work on typestate verification has included approaches that use special type systems or program annotations[61,64–70] (these systems often restrict aliasing by various means, although Reference 68 does not) and techniques that are purely analytical.[71–79] Note that a number of these approaches address verification problems not expressible in terms of finite state machines, in addition to those that are. The ESP system[74] uses context-sensitive interprocedural data-flow analysis to check typestate properties for C programs. ESP introduces *property simulation,* a heuristic to provide partial path sensitivity according to how data-flow facts correlate with particular typestates. For object-oriented languages, typestate specifications commonly associate a typestate with a particular object instance. In order to reason about how a program manipulates an object, *alias analysis* presents a key challenge. Many existing verification frameworks use a two-phase approach, performing points-to analysis as a preceding phase, followed by typestate checking.[71,74,80] The current version of ESP[75] uses an integrated approach, recording typestate and alias information in a flow-sensitive manner. Field et al. present algorithms based on abstractions that integrate alias and typestate information but restricted to shallow programs, with only single-level pointers to typestate objects.[77]

The parametric shape analysis presented in Reference 81 has served as the basis for very precise verification algorithms, in which the verification is integrated with heap analysis.[78,79,82] These algorithms, however, do not scale well. Approaches based on counterexample-guided refinement have had impressive results in certain domains,[83,84] but so far they have been less successful in dealing with complex heap manipulation, partly because these approaches attempt to *automatically* derive appropriate heap analyses. Flow analyses can apply to

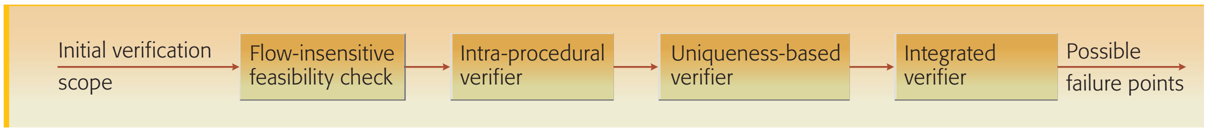
Initial verification scope
Flow-insensitive feasibility check
Intra-procedural verifier
Uniqueness-based verifier
Integrated verifier
Possible failure points

Figure 10
A state-of-the-art typestate verifier for Java

extant programs in general programming languages, but generally require expensive interprocedural analysis. In contrast, type systems can provide a modular approach to typestate checking, but generally can accept a smaller class of correct programs. DeLine and Fähndrich[85] present a type system for typestate properties for objects. Their system ensures that a program which typechecks has no typestate violations and provides a modular, sound checker for object-oriented programs. To handle aliasing, they employ the *adoption* and *focus* operations to a linear type system, as described in Reference 86. With these operations, the type checker can assume *must-alias* properties for a limited program scope and thus, apply strong updates allowing typestate transitions. Aiken et al.[70] present an inference algorithm for inferring *restricted* and *confined* pointers, which they use to enable strong updates.

A state-of-the-art typestate-verifier algorithm

This section presents an overview of a state-of-the-art typestate verifier algorithm for Java.[76] The verification system is a *composite* verifier built out of several *composable* verifiers of increasing precision and cost. Each verifier can run independently, but the composite verifier stages analyses in order to improve efficiency without compromising precision. The early stages use the faster verifiers to reduce the workload for later, more precise stages, as shown in ***Figure 10***.

The system is comprised of four stages: a flow-insensitive analysis, an intraprocedural analyzer, an inexpensive analysis based on *uniqueness* analysis, and an integrated verifier combining alias information and typestates. This subsection covers a short overview of an abstraction technique used by the integrated verifier, the last and most precise analysis stage. The integrated verifier performs flow- and context-sensitive verification with an abstraction that combines aliasing information with typestate information. The use of a combined domain is more precise than separately performing typestate checking and flow-sensitive alias analysis, as is common with abstract interpretation over combined domains.[87]

The abstract domain captures information about the typestate of the given abstract object, as well as information regarding potential aliases of an abstract object. In this presentation, we represent an abstract program state as a set of tuples, where each tuple has the following elements:

- `AO`, a representation of an abstract object from the preliminary pointer analysis
- `T`, the typestate occupied by the abstract object
- `Must`, a set of symbolic access paths (for example, `x.f.g`) that must point to a particular object
- `May`, a bit indicating whether the `Must` set is `incomplete`; may there exist other access paths pointing to the abstract object, which do not appear in the `Must` set?
- `MustNot`, a set of symbolic access paths that must not point to a particular object

Notice that this paper presents a simplified abstraction to illustrate the combined tracking of aliasing and typestate. The full verifier[76] also incorporates a *uniqueness* abstraction and several optimizations, not described here.

By way of example, consider the code in ***Figure 11***. We wish to verify that the program uses `Signature` objects correctly according to the typestate specification of Figure 9.

The example includes one allocation site for `Signature` objects, the allocation at line 5, which we will denote by S. When processing this statement for the first time, the solver generates an abstract state representing an abstract object S in the `INIT` state, pointed-to by `s`. Immediately after line 5, there can be no other pointers to this object, so `May` is set to `false`, resulting in an abstract program state of $\{\langle S,$ `INIT`, $\{$`s`$\}$, `false`, $\emptyset\rangle\}$.

```
1: public static void foo() throws Exception {
2:     Collection<Signature> signatures = new LinkedList<Signature>();
3:
4:     for (int i = 0; i < 5; i++) {
5:             Signature s = new Signature();
6:             s.initSign();
7:             s.update();
8:             s.sign();
9:             signatures.add(s);
10:    }
11:
12:    for(Iterator<Signature> it2=signatures.iterator();it2.hasNext(); ){
13:             Signature s2 = it2.next();
14:             s2.initVerify();
15:             s2.update();
16:             s2.verify();
17:    }
18: }
```

Figure 11
Example program using `java.security.Signature`

Based on the typestate automaton, we see that statement 6 forces a typestate transition to the `SIGN` state. Because the abstract state includes `Must` points-to information for `s`, the effect of statement 6 can be precisely reflected by a *strong update,* resulting in an abstract state of {⟨*S*, `SIGN`, {`s`}, `false`, Ø⟩}.

The next interesting operation occurs at statement 9, which stores the abstract object into a set. Because the abstraction has not been tracking this set, statement 9 induces new access paths which may point to the abstract object. The abstraction models this by setting `May` to `true`, as follows: {⟨*S*, `SIGN`, {`s`}, `true`, Ø⟩}.

A key element of the integrated verifier's abstraction is the use of a *focus* operation,[81] which dynamically (during analysis) makes distinctions between objects that the underlying basic points-to analysis does not distinguish. In the example code, the key focus operation occurs at statement 14. When first propagating at statement 14, the solver propagates an abstract state that indicates an abstract object in the `SIGN` state, but does not indicate any useful `Must` pointer information: {⟨*S*, `SIGN`, Ø, `true`, Ø⟩}.

Statement 14 causes a typestate transition to the `VERIFY` state. In order to apply strong updates downstream, the solver applies a focus operation to *split* the abstract state into two cases, each of which holds more precise pointer information. There are two possibilities for each concrete object at this program point: either `s2` points to the object, or it does not. In the first case, statement 14 causes a transition to the `VERIFY` state, and `s2` *must* point to the object; in the second case, statement 14 does not cause a typestate transition, and `s2` *must not* point to the abstract object; therefore, after applying this focus logic to statement 14, the abstraction produces two tuples: {⟨*S*, `VERIFY`, {`s2`}, `true`, Ø⟩, ⟨*S*, `SIGN`, Ø, `true`, {`s2`}⟩}.

Downstream, when processing statement 16, the solver uses the alias information to avoid a spurious transition (false positive) to the `ERROR` state. In particular, the second tuple indicates that `s2` *must not* point to the object manipulated by statement 16; thus, the call to `verify` will not occur on an object in the `SIGN` state.

The solver thus iterates to a fixed point, updating the abstract state at each program point based on simple flow functions derived from the language semantics. ***Table 1*** shows the final result of the fixed-point iteration at each program point in the example. The abstraction suffices to demonstrate that this example uses `Signature` objects correctly, although those objects flow through complex collection classes.

More generally, this verifier can verify similar patterns that flow across procedures, using flow-

Table 1 Abstract states for typestate verification of Program in Figure 11 and Property in Figure 9

Statements	Abstract States
4	{⟨S, INIT, {s}, false, Ø⟩, ⟨S, SIGN, Ø, true, Ø⟩}
5	{⟨S, SIGN, {s}, false, Ø⟩, ⟨S, SIGN, Ø, true, {s}⟩}
6	{⟨S, SIGN, {s}, false, Ø⟩, ⟨S, SIGN, Ø, true, {s}⟩}
7	{⟨S, SIGN, {s}, false, Ø⟩, ⟨S, SIGN, Ø, true, {s}⟩}
8	{⟨S, SIGN, {s}, false, Ø⟩, ⟨S, SIGN, Ø, true, {s}⟩}
9	{⟨S, SIGN, {s}, true, Ø), ⟨S, SIGN, Ø, true, {s}⟩}
10	{⟨S, SIGN, Ø, true, Ø⟩}
11	{⟨S, SIGN, Ø, true, Ø⟩}
12	{⟨S, SIGN, Ø, true, Ø⟩, ⟨S, VERIFY, Ø, true, Ø⟩}
13	{⟨S, SIGN, Ø, true, Ø⟩, ⟨S, VERIFY, Ø, true, Ø⟩}
14	{⟨S, SIGN, Ø, true, {s2}⟩, ⟨S, VERIFY, {s2}, true, Ø⟩}
15	{⟨S, SIGN, Ø, true, {s2}⟩, ⟨S, VERIFY, {s2}, true, Ø⟩}
16	{⟨S, SIGN, Ø, true, {s2}⟩, ⟨S, VERIFY, {s2}, true, Ø⟩}

and context-sensitive interprocedural data-flow analysis. The current implementation, as described in Reference 76, can analyze roughly 100,000 LOC (lines of code) in about 10 minutes. The results in Reference 76 report that the analyzer verifies correctness for 93 percent of the eligible statements, throughout a suite of moderate-sized benchmarks, for a set of 11 typestate properties from the Java standard libraries.

CONCLUSION

In this paper, we describe various static-analysis techniques for identifying security vulnerabilities in software systems. We present a number of security analyses in detail and provide a broad overview of related work in several areas. In particular, we focus on analyses of component-based systems, such as the Java and .NET CLR platforms, which have adopted both SBAC and RBAC as access-control mechanisms. For such systems, we discuss how static analysis can facilitate automatic determination of permission requirements and automatic placement of privilege-asserting operations. Additionally, we show how to detect violations of policies for integrity and confidentiality. Finally, we discuss how static analysis can be used to verify the correct usage of security libraries and interfaces. For each of these areas, we present a brief survey of research results and then discuss a few algorithms in depth in order to illustrate fundamental algorithmic techniques.

ACKNOWLEDGMENTS

The authors thank Paolina Centonze, Julian Dolby, Nurit Dor, Robert J. Flynn, Emmanuel Geay, Aaron Kershenbaum, Alexey Loginov, and Ganesan Ramalingam for their invaluable contributions to this paper.

**Trademark, service mark, or registered trademark of Sun Microsystems, Inc. or The Open Group, in the United States, other countries, or both.

CITED REFERENCES

1. J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE* **63**, No. 9, 1278–1308 (September 1975).
2. Common Criteria, http://www.commoncriteriaportal.org.
3. Sun Microsystems, Security Code Guidelines, http://java.sun.com/security/seccodeguide.html.
4. Open Web Application Security Project, http://www.owasp.org.
5. U.S. Department of Homeland Security, http://www.dhs.gov.
6. National Vulnerability Database, National Institute of Standards and Technology, U.S. Commerce Department, http://nvd.nist.gov.
7. SecurityFocus, Symantec Corporation, http://www.securityfocus.com.
8. 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," *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA (December 1997).
9. M. Pistoia, D. Reller, D. Gupta, M. Nagnur, and A. K. Ramani, *Java 2 Network Security*, 2nd ed., Upper Saddle River, NJ, USA, Prentice Hall PTR (August 1999).
10. D. F. Ferraiolo and D. R. Kuhn, "Role-Based Access Controls," *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, Baltimore, MD, USA (October 1992), pp. 554–563.

11. M. Pistoia, N. Nagaratnam, L. Koved, and A. Nadalin, *Enterprise Java Security*, Reading, MA, USA, Addison-Wesley (February 2004).

12. A. Freeman and A. Jones, *Programming .NET Security*, Sebastopol, CA, USA, O'Reilly & Associates, Inc. (June 2003).

13. G. Naumovich and P. Centonze, "Static Analysis of Role-Based Access Control in J2EE Applications," *SIGSOFT Software Engineering Notes* **29**, No. 5, 1–10 (September 2004).

14. P. Centonze, G. Naumovich, S. J. Fink, and M. Pistoia, "Role-Based Access Control Consistency Validation," *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '06)*, Portland, Maine, USA (July 2006), pp. 121–132.

15. H. Chen, D. Wagner, and D. Dean, "Setuid Demystified," *Proceedings of the 11th USENIX Security Symposium*, Berkeley, CA, USA, USENIX Association (August 2002), pp. 171–190.

16. F. Schneider, G. Morrisett, and R. Harper, "A Language-Based Approach to Security," Cornell University, Ithaca, NY, USA, Technical Report TR2000-1825 (November 2000).

17. D. S. Wallach, A. W. Appel, and E. W. Felten, "SAFKASI: A Security Mechanism for Language-Based Systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)* **9**, No. 4, 341–378 (2000).

18. D. S. Wallach and E. W. Felten, "Understanding Java Stack Inspection," *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, IEEE Computer Society Press (May 1998), pp. 52–63.

19. F. Pottier, C. Skalka, and S. F. Smith, "A Systematic Approach to Static Access Control," *Proceedings of the 10th European Symposium on Programming Languages and Systems*, Springer-Verlag (2001), pp. 30–45.

20. T. P. Jensen, D. L. Métayer, and T. Thorn, "Verification of Control Flow Based Security Properties," *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, USA (May 1999) pp. 89–103.

21. M. Bartoletti, P. Degano, and G. L. Ferrari, "Static Analysis for Stack Inspection," *Proceedings of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science* **54**, Amsterdam, The Netherlands, Elsevier (2001), pp. 69–80.

22. A. Banerjee and D. A. Naumann, "A Simple Semantics and Static Analysis for Java Security," Stevens Institute of Technology, Hoboken, NJ, USA, Technical Report CS2001-1 (July 2001).

23. U. Erlingsson and F. B. Schneider, "IRM Enforcement of Java Stack Inspection," *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, IEEE Computer Society (May 2000), pp. 246–255.

24. B. G. Ryder, "Dimensions of Precision in Reference Analysis of Object-Oriented Languages," *Proceedings of the 12th International Conference on Compiler Construction*, Warsaw, Poland (April 2003), pp. 126–137, invited paper.

25. C. Lai, L. Gong, L. Koved, A. J. Nadalin, and R. Schemers, "User Authentication and Authorization in the Java™ Platform," *Proceedings of the 15th Annual Computer Security Applications Conference*, Scottsdale, AZ, USA, IEEE Computer Security (December 1999), pp. 285–290.

26. M. Pistoia, "A Unified Mathematical Model for Stack- and Role-Based Authorization Systems," Ph.D. dissertation, Polytechnic University, Brooklyn, NY, USA (May 2005).

27. M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar, "Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection," *Proceedings of the 9th European Conference on Object-Oriented Programming*, Glasgow, Scotland, UK, Springer-Verlag (July 2005).

28. IBM Corporation, Security Workbench Development Environment for Java (SWORD4J), http://www.alphaworks.ibm.com/tech/sword4j/.

29. G. A. Kildall, "A Unified Approach to Global Program Optimization," *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Boston, MA, USA, ACM Press (1973), pp. 194–206.

30. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-Based Access Control Models," *IEEE Computer* **29**, No. 2, 38–47 (February 1996).

31. A. Schaad and J. D. Moffett, "A Lightweight Approach to Specification and Analysis of Role-Based Access Control Extensions," *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, Monterey, CA, USA, ACM Press (2002), pp. 13–22.

32. D. Jackson, "Alloy: A Lightweight Object Modelling Notation," *ACM Transactions of Software Engineering Methodologies* **11**, No. 2, 256–290 (2002).

33. D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: The Alloy Constraint Analyzer," *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, ACM Press (2000), pp. 730–733.

34. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati, "A Fine-Grained Access Control System for XML Documents," *ACM Transactions on Information Systems Security* **5**, No. 2, 169–202 (2002).

35. M. Kudo and S. Hada, "XML Document Security Based on Provisional Authorization," *Proceedings of the 7th ACM Conference on Computer and Communications Security*, Athens, Greece, ACM Press (November 2000), pp. 87–96.

36. M. Murata, A. Tozawa, M. Kudo, and S. Hada, "XML Access Control Using Static Analysis," *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington, DC, USA, ACM Press (October 2003), pp. 73–84.

37. F. Ricca and P. Tonella, "Analysis and Testing of Web Applications," *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, ON, Canada, IEEE Computer Society (2001), pp. 25–34.

38. D. Clarke, M. Richmond, and J. Noble, "Saving the World from Bad Beans: Deployment-Time Confinement Checking," *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, Anaheim, CA, USA, ACM Press (2003), pp. 374–387.

39. M. Pistoia, S. J. Fink, R. J. Flynn, and E. Yahav, "When Role Models Have Flaws: Static Validation of Enterprise Security Policies," *Proceedings of the International Conference on Software Engineering (ICSE 2007)*, Minneapolis, MN, USA (forthcoming, May 2007).

40. D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow," *Communications of the ACM* **20**, No. 7, 504–513 (July 1977).

41. D. E. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM* **19**, No. 5, 236–243 (May 1976).

42. G. Grätzer, *General Lattice Theory*, 2nd ed., Boston, MA, USA, Birkhäuser (January 2003).

43. J. A. Goguen and J. Meseguer, "Security Policies and Security Models," *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, IEEE Computer Society Press (May 1982), pp. 11–20.

44. J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, IEEE Computer Society (February 2005), http://jimnewsome.net/papers/taintcheck.pdf.

45. K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, IEEE Computer Society (May 2002), pp. 143–159.

46. A. Sabelfeld and A. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas in Communications* **21**, No. 1, 5–19 (January 2003).

47. D. Volpano, C. Irvine, and G. Smith, "A Sound Type System for Secure Flow Analysis," *Journal of Computer Security* **4**, Nos. 2-3, 167–187 (January 1996).

48. A. C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control," *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, ACM Press (October 1997), pp. 129–142.

49. J. F. Koegel, R. M. Koegel, Z. Li, and D. T. Miruke, "A Security Analysis of VAX VMS," *ACM '85: Proceedings of the 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, ACM Press (1985), pp. 381–386.

50. L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*, 3rd ed., Sebastopol, CA, USA, O'Reilly and Associates, Inc. (July 2000).

51. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers," *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, USA (August 2001), pp. 201–220.

52. J. S. Foster, T. Terauchi, and A. Aiken, "Flow-Sensitive Type Qualifiers," *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany (June 2002), pp. 1–12.

53. G. Snelting, T. Robschink, and J. Krinke, "Efficient Path Conditions in Dependence Graphs for Software Safety Analysis," *ACM Transactions on Software Engineering and Methodology* **15**, No. 4, 410–457 (2006).

54. C. Hammer, J. Krinke, and G. Snelting, "Information Flow Control for Java Based on Path Conditions in Dependence Graphs," *Proceedings of IEEE International Symposium on Secure Software Engineering*, Arlington, Virginia, USA (2006), pp. 87–96.

55. M. Sridharan and R. Bodik, "Refinement-Based Context-Sensitive Points-To Analysis for Java," *Proceedings of ACM Conference on Programming Language Design and Implementation* (2006), pp. 387–400.

56. V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," *Usenix Security Symposium* (2005).

57. J. Whaley and M. S. Lam, "Cloning Based Context-Sensitive Pointer Alias Analysis Using BDDs," *Proceedings of ACM Conference on Programming Language Design and Implementation* (2004), pp. 131–144.

58. D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, ACM Press (2004), pp. 132–136.

59. PMD, http://sourceforge.net/projects/pmd/.

60. G. Begic and B. Higgins, "Overview of Static Analysis in IBM Rational Application Developer 6.0," http://www-128.ibm.com/developerworks/rational/library/05/higgins.

61. R. E. Strom and S. Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *IEEE Transactions on Software Engineering* **12**, No. 1, pp. 157–171 (1986).

62. J. Whaley, M. C. Martin, and M. S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces," *Proceedings of the International Symposium on Software Testing and Analysis* (July 2002), pp. 218–228, http://citeseer.ist.psu.edu/525755.html.

63. R. Alur, P. Cerny, P. Madhusudan, and W. Nam, "Synthesis of Interface Specifications for Java Classes," *SIGPLAN Notices* **40**, No. 1, pp. 98–109 (2005).

64. R. E. Strom and D. M. Yellin, "Extending Typestate Checking Using Conditional Liveness Analysis," *IEEE Transactions on Software Engineering* **19**, No. 5, 478–485 (May 1993).

65. R. DeLine and M. Fähndrich, "Enforcing High-Level Protocols in Low-Level Software," *Proceedings of ACM Conference on Programming Language Design and Implementation* (June 2001), pp. 59–69.

66. R. DeLine and M. Fähndrich, "Adoption and Focus: Practical Linear Types for Imperative Programming," *Proceedings of ACM Conference on Programming Language Design and Implementation*, Berlin (June 2002), pp. 13–24.

67. J. S. Foster, T. Terauchi, and A. Aiken, "Flow-Sensitive Type Qualifiers," *Proceedings of ACM Conference on Programming Language Design and Implementation*, Berlin (June 2002), pp. 1–12.

68. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," *Proceedings of ACM Conference on Programming Language Design and Implementation*, Berlin (June 2002), pp. 234–245.

69. V. Kuncak, P. Lam, and M. Rinard, "Role Analysis," *Proceedings of ACM Symposium on Principles of Programming Languages*, Portland, Maine (January 2002), pp. 17–32.

70. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi, "Checking and Inferring Local Non-aliasing," *ACM SIGPLAN Notices* **38**, No. 5, 129–140 (May 2003), in *Conference on Programming Language Design and Implementation (PLDI)*.

71. J. C. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng, "Bandera: Extracting Finite-State Models from Java Source Code," *Proceedings of International Conference on Software Engineering* (June 2000), pp. 439–448.

72. T. Ball and S. K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Proceedings of 8th International SPIN Workshop of Model Checking of Software (SPIN 2001)*, in *Lecture Notes in Computer Science* **2057**, 103–122 (2001).

73. K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA (May 2002).

74. M. Das, S. Lerner, and M. Seigle, "ESP: Path-Sensitive Program Verification in Polynomial Time," *ACM SIG-*

PLAN Notices **37**, No. 5, 57–68 (May 2002), in *Conference on Programming Language Design and Implementation (PLDI)*.

75. N. Dor, S. Adams, M. Das, and Z. Yang, "Software Validation via Scalable Path-Sensitive Value Flow Analysis," *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), pp. 12–22, http://doi.acm.org/10.1145/1007515.
76. S. Fink, E. Yahav, G. Ramalingam, N. Dor, and E. Geay, "Effective Typestate Verification in the Presence of Aliasing," *Proceedings of the International Symposium on Software Testing and Analysis* (2006), pp. 133–144.
77. J. Field, D. Goyal, G. Ramalingam, and E. Yahav, "Typestate Verification: Abstraction Techniques and Complexity Results," *Proceedings of Static Analysis Symposium (SAS'03)*, in *Lecture Notes in Computer Science* **2694**, 439–462 (June 2003).
78. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv, "Deriving Specialized Program Analyses for Certifying Component-Client Conformance," *Proceedings of ACM Conference on Programming Language Design and Implementation*, *ACM SIGPLAN Notices* **37**, No. 5, 83–94, New York: ACM Press (June 17–19 2002).
79. R. Shaham, E. Yahav, E. Kolodner, and M. Sagiv, "Establishing Local Temporal Heap Safety Properties with Applications to Compile-Time Memory Management," *Proceedings of Static Analysis Symposium* (2003), pp. 483–503, http://link.springer.de/link/service/series/0558/bibs/2694/26940483.htm.
80. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani, "Automatic Predicate Abstraction of C Programs," *Proceedings of ACM Conference on Programming Language Design and Implementation* (June 2001), pp. 203–213.
81. M. Sagiv, T. Reps, and R. Wilhelm, "Parametric Shape Analysis via 3-Valued Logic," *Transactions on Programming Languages and Systems (TOPLAS)* **24**, No. 3, 217–298 (May 2002).
82. E. Yahav and G. Ramalingam, "Verifying Safety Properties Using Separation and Heterogeneous Abstractions," *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ACM Press (2004), pp. 25–34.
83. T. Ball and S. K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," *ACM SIGPLAN Notices* **37**, No. 1, pp. 1–3 (Jan. 2002).
84. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," *Proceedings of ACM Symposium on Principles of Programming Languages* (2002), pp. 58–70.
85. R. DeLine and M. Fähndrich, "Typestates for Objects," *18th European Conference on Object-Oriented Programming (ECOOP)*, in *Lecture Notes in Computer Science* **3086**, (June 2004), 465–490.
86. M. Fähndrich and R. DeLine, "Adoption and Focus: Practical Linear Types for Imperative Programming," *ACM SIGPLAN Notices* **37**, No. 5, 13–24 (May 2002), in *Conference on Programming Language Design and Implementation (PLDI)*.
87. P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks," *Proceedings of ACM Symposium on Principles of Programming Languages*, New York, NY, ACM Press (1979), pp. 269–282.

Accepted for publication January 15, 2007.
Published online April 23, 2007.

Marco Pistoia
IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (pistoia@us.ibm.com). Dr. Pistoia received a Ph.D. in mathematics from Polytechnic University, Brooklyn, New York in 2005, and B.S. and M.S. degrees in mathematics *summa cum laude* from the University of Rome, Italy in 1995. From 1996 to 1998, he worked at the IBM Network Computing Research Center in Cagliari, Italy. In the years 1998 and 1999, he was a project manager at the IBM International Technical Support Organization in Raleigh, North Carolina. Since 1999, he has worked at the Watson Research Center, where he is currently a research staff member in the Programming Languages and Software Engineering department. He is the author of 10 books. His research interests include program analysis and software security.

Satish Chandra
IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (satishchandra@us.ibm.com). Dr. Chandra obtained a Ph.D. from the University of Wisconsin-Madison in 1997, and a B.Tech degree from the Indian Institute of Technology-Kanpur in 1991, both in computer science. From 1997 to 2002, he was a member of the technical staff at Bell Laboratories, where his research focused on program analysis, domain-specific languages, and data-communication protocols. In September 2002, he joined IBM Research in New Delhi, India, where he managed a small research group in software engineering. He is currently a member of the Programming Languages and Software Engineering department at the Watson Research Center.

Stephen J. Fink
IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (sjfink@us.ibm.com). Dr. Fink received a B.S. degree from Duke University in 1992 and M.S. and Ph.D. degrees from the University of California, San Diego in 1994 and 1998. Since 1998, he has been a research staff member in the Software Technology Department at the Watson Research Center. He was a member of the team that produced the Jikes Research Virtual Machine. He serves as lead architect for the IBM Common Architecture for Program Analysis. His research interests include static and dynamic program analysis, programming language implementation techniques, and parallel and scientific computation. He is a member of ACM.

Eran Yahav
IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (eyahav@us.ibm.com). Dr. Yahav received a B.S. degree from the Technion-Israel Institute of Technology in 1996, and a Ph.D. degree from Tel Aviv University in 2004. Since late 2004, he has been a research staff member in the Advanced Programming Tools Department at the Watson Research Center. His research interests include static and dynamic program analysis, programming language design and implementation, and program verification. ■