Towards Security Program Comprehension with Design by Contract and Slicing

Karsten Sohr, Tanveer Mustafa, Markus Gulmann, Patrick Gerken
Center for Computing Technologies (TZI), Universität Bremen
Bremen, Germany
{sohr|tanveer|gulmann|pgerken}@tzi.de

***Abstract*—Over the last years, the field of software security has made much progress. More and more software vendors employ static code analyzers as well as dynamic application security testing; at the architectural level techniques such as Threat Modeling are used. However, given that deep software security knowledge is still rare in industry, tools are needed that support software vendors in better understanding the implemented security architecture of their applications. In this work, we present an approach to software security comprehension based on principles of Design by Contract (DBC). In particular, we reconstruct parts of an application's security architecture with the help of program slicing and specification inference in form of code annotations by utilizing knowledge on software frameworks and Security APIs. The inferred annotations can then be analyzed by Extended Static Checkers. Our proposed methodology can be seen as a first step towards more systematic security code audits.**

I. Introduction

More and more software frameworks and libraries are made available to developers with quite stable APIs. Typical examples are the Java Enterprise Edition (JEE) [51], Spring [55] or the Android Framework [33]. Often, these frameworks provide security functionality (e.g., encryption, access control, authentication) or APIs that are relevant for security. However, it is not sufficient to only employ Security libraries; it must also be guaranteed that they are *used correctly*. For example, a study has shown that many popular Android applications implemented SSL functionality wrongly, such that middleperson attacks were possible [26]. Moreover, Georgiev et al. showed that many applications (not only Android apps) did not validate SSL certificates correctly or did not check the domain name of the server against the distinguished name in the certificate [32]. They concluded that SSL libraries are too complex and hence wrongly used. This problem applies to cryptographic APIs as well, e.g., developers use hard-coded secrets, generate keys with weak entropy or do not know the security implications of selecting a cryptographic algorithm/mode [22].

In this paper, we address the problem of checking whether applications use Security APIs to satisfy their security requirements. We focus on two aspects, namely (i) checking application-specific security requirements (e.g., access control requirements) and (ii) assuring that Security APIs and software frameworks are not used in a way that introduces security vulnerabilities.

We propose a static analysis approach based on principles of Design by Contract (DBC) [48], [7] and program comprehension techniques [37], [41]. DBC allows one to define code annotations in the form of pre- and postconditions as well as invariants. We employ code annotations to specify the security requirements of an application, e.g., "patient data must be encrypted when they are sent per e-mail". Related tools for Extended Static Checking [7], [29] can then be employed to verify that the code satisfies its annotations (i.e., the security requirements).

Since Extended Static Checking becomes time-consuming or even impossible when applied to large code with many dependences [45], we additionally propose to employ interprocedural program slicing [41]. This technique allows an analyst to automatically determine dependences that influence the calls of Security APIs. This way, one can extract those parts of the *implemented security architecture* from the source code that realize security requirements. Thereafter one can annotate and analyze this extracted security architecture with static checkers.

Since annotating code with specifications has been shown to be a tedious task [29], it is important to have a method at hand that automatically inserts annotations into program code. For this purpose, we propose to use the Daikon tool, which can infer DBC specification by code instrumentation [25]. The preceding slicing step helps then the inference tool produce annotations that better focus on the implemented security mechanisms.

The innovation of our technique lies in the combination of three well-established software engineering techniques (slicing, Extended Static Checking and specification inference) with a comprehensive knowledge base of annotations for common Security APIs. Our approach is not restricted to security functionality—the proposed tool can also encode knowledge of other APIs which are still relevant for security, but do not implement security features (e.g., access to sockets, interprocess communication).

Moreover, the sliced code and the annotations contribute to a better comprehension of the implemented security mechanisms, specifically if specifications can be automatically inferred from code. Although several works discuss the relevance of DBC in the security context [12], [58], they do not consider the important aspects of security program comprehension and correct usage of Security APIs. We believe that more systematic approaches to security code audits will be possible if advanced analysis functionality is incorporated into the next generation of static source code analyzers. The

problem of **program comprehension for security** has not been addressed sufficiently so far as common approaches to static code analysis overlook this aspect.

The remainder of this paper is structured as follows. Section II describes the background of our work focusing on the main concepts of DBC. In Section III, we first motivate and then introduce our analysis technique, whereas Section IV describes experiments that we carried out with a proof-of-principle implementation of our approach. In Section V, we discuss the limitations of our approach and the current tool support. We also elaborate on how future analysis tools should look like. After discussing related work, we conclude the paper in Section VII.

II. Background

We describe the background technologies which are the foundation of our work.

A. Design by Contract

The principle of DBC allows a developer to specify pre- and postconditions, which must be satisfied on function entry and exit, respectively [48], [60], [7]. Invariants apply to the entry and exit of *all public* methods. For most mainstream programming languages, DBC extensions exist [7], [12], [43], most notably, the Java Modeling Language (JML). A comprehensive overview of DBC-based specification languages can be found in a recent survey by Hatcliff et al. [35]. We now describe JML in more detail because we use it for the discussion throughout this paper.

Java Modeling Language: JML is a formal behavioral interface specification language, specifically designed for specifying the functional behavior of Java programs [42], [7]. JML provides a rich set of language constructs that are necessary to precisely specify the functional behavior of Java programs, mostly, in the form of class invariants as well as pre- and postconditions of methods. JML specifications are written in special annotation comments in the form of `/*@...@*/` or simply `//@...` if a single line specification is intended. The JML tools use these annotations to parse the JML specifications out of the Java programs. JML provides `requires` and `ensures` clauses to specify pre- and postconditions of a method. The preconditions enforce the client's obligations, whereas postconditions enforce the implementer's obligations. JML provides a logical variable `\result` that represents the value returned by a method. In addition, JML supports the concept of `pure` methods, which are side-effect free public methods. Only these kinds of methods can be called within a JML specification.

Extended Static Checking: A variety of tools exist that allow one to check the JML constraints at run-time or statically [7]. One such tool is ESC/Java2, which statically detects inconsistencies between the code and the specification using a built-in automatic theorem prover. However, since such conformance checking is in general undecidable, false positives and negatives may be produced. ESC/Java2 employs **modular reasoning**, which is an effective technique when used in combination with static checking. Java methods can be analyzed one at a time and their JML-based specifications can be proven by inspecting only the specification contracts (and not the code) of the methods called within their bodies [29].

B. Program Slicing

Program slicing was first introduced by Weiser who pointed out that developers understand programs according to dependences between statements and not necessarily to the natural order of the code [59]. A backward slicing algorithm starts from a statement, the so-called "slicing criterion", and calculates all the statements that (transitively) influence the slicing criterion. Slicing is often used for program comprehension and debugging tasks in order to focus on those code parts that are relevant for the analysis. Technically, slicing is usually implemented by system dependence graphs (SDGs) [37]. SDGs often contain the statements in static single assignment form (SSA) [3], an intermediate representation well-suited to data and control flow analyses, as well as call graph information. In particular, an SDG represents methods via special nodes. Context-sensitive slicing only allows accessible execution paths, i.e., a method must return to the site where the method has been called and not to other call sites of the method. Krinke gives a detailed overview of slicing techniques [41].

III. The Proposed Analysis Approach

First, we motivate the need for static analysis tools with functionality for security program comprehension, and thereafter, we describe how such an analysis approach works. We conclude this section with a discussion of several Security APIs in the light of the DBC principle.

A. Motivation

Our discussion starts from the task of evaluating the security mechanisms implemented in a clinical information system either by internal quality assurance (QA) or an external evaluator (e.g., w.r.t. the Common Criteria [14]). Among other functionality, this system should provide means for reading and writing electronic health records (EHRs). We also assume that it is a JEE-based client-server application [51].

In Fig. 1, we depict some security requirements for this application, which are part of a hospital's security policy. For example, the requirement exists that a clinician must play the role `Physician` to read or write the patient data of the EHR (`Req 3`); for reading prescriptions, the role `Nurse` suffices (`Req 1`). There are additional access control requirements, usually called "context constraints" [31], which state that the clinician must be on the same ward as the patient. The hospital's security policy also comprises confidentiality and integrity requirements, such as "a doctor's letter must be encrypted and digitally signed with the treating physician's certificate " (`Req 5`). A doctor's letter is usually sent to a general practitioner after a patient's treatment at a hospital has been finished and a follow-up treatment is necessary. In addition, the hospital belongs to a healthcare provider who

Excerpt from a security policy of a clinical information system

Req 1: Data about a patient's prescriptions may only be read by clinicians who assume the roles ''Physician'' or ''Nurse'' and are on the same ward as the patient.

Req 2: Data about a patient's prescriptions may only be written by physicians who are on the same ward as the patient.

Req 3: Patient data may only be read or written by clinicians with the role ''Physician''.

Req 4: Patient data may only be written or read by physicians who are on the same ward as the patient.

Req 5: A doctor's letter must be encrypted and digitally signed.

Req 6: Sending a doctor's letter is only allowed for clinicians with the role ''Physician''.

Req 7: The communication with a Cloud-based server must be authenticated, and confidentiality as well as integrity of the sent/received data must be assured.

Fig. 1. Some security requirements of the clinical information system.

runs several hospitals. In particular, the healthcare provider offers storage capacity as a cloud service to the single hospitals; the corresponding connections must be appropriately authenticated and secured (`Req 7`).

To illustrate our ideas, Fig. 2 depicts an excerpt of the implementation of a fictitious clinical information system. The implementation employs JEE's programmatic authorization `EJBContext.isCallerInRole()` to enforce that the caller of a method plays the appropriate roles. For example, the method `readPrescriptions()` (lines 5-11) checks whether the caller has assumed the roles `Physician` or `Nurse`. In addition, the code uses cryptographic functionality, which is provided by Java security libraries. Also, SSL functionality is implemented using Java security and Apache libraries to enable secure communications with the Cloud. We assume that the developers implemented SSL functionality because they provide their own X.509 root certificate stored in a Java key store.

The job of the software QA now is to evaluate the software w.r.t. the security requirements presented in Fig 1. How can an analyst be sure that the policy and the implementation correspond to each other? For example, does the method `writePrescriptions()` (lines 13-19) satisfy `Req 2`? It may be difficult for an evaluator (at least, if she is external) to understand the details of the code and to identify the relevant code parts which implement an application's security architecture. In practice, the code is much more complex than our example.

Given that Security APIs are increasingly used, knowledge of the security functionality provided by these libraries should be integrated into static code analyzers. Specifically, it would be helpful for an analyst to have a tool at hand that

1) automatically extracts the relevant code locations that implement the security requirements,
2) allows one to specify the security requirements, and
3) verifies the security requirements against the code.

For example, our discussions with product CERTs or QA of large software vendors show that they rarely have the chance to comprehensively validate the code against the security requirements due to the high workload [38]. A tool that helps them focus on the implementation of security requirements would be very valuable for them. Subsequently, we outline the core concepts of such an analysis tool in more detail.

B. The Analysis Approach in Detail

Three central observations based on the aforementioned motivation guide our work: the knowledge of widely-deployed Security APIs should already be available (e.g., as specifications); a technique is required to automatically extract the implemented security policy based on the employed Security API; and a means is needed to statically analyze the implemented security policy against specifications of the application's security requirements.

1) DBC-Based Annotations: We propose to utilize the advantages of DBC-based analysis for this purpose because we consider the application of Security APIs a contract between the client (the application) and the callee (the security library). Following this line of argumentation, the precondition of a Security API method must be satisfied by the client, whereas the postcondition must be fulfilled by the called API method. Preconditions help a developer use Security APIs correctly, e.g., using state-of-the-art cryptographic algorithms or creating secure random numbers for a symmetric key. In addition, the postconditions must be strong enough to enforce an application's security requirements, e.g., the intended access control policy. Consequently, the client is responsible for choosing sufficient Security APIs and use them in a way to guarantee its security requirements.

The aforementioned steps 2) and 3) can be handled naturally by DBC and related tools. For example, the security requirement "Data about a patient's prescriptions may only be written

```
@Resource EJBContext ctx;
Cipher mCipher;
Signature mSignature;

public String readPrescriptions(String ehrID, String userID){
    Clinician clinician = getClinician(userID);
    EHR eHR= getEHR(ehrID);
    if !((ctx.isCallerInRole("Physician") || ctx.isCallerInRole("Nurse")) && eHR.getWard() == clinician.getWard())
        throw new SecurityException("No sufficient access rights.");
    return eHR.getPrescriptions();
}

public void writePrescriptions(String ehrID, String userID, String pres){
    Clinician clinician = getClinician(userID);
    EHR eHR= getEHR(ehrID);
    if !((ctx.isCallerInRole("Physician") || ctx.isCallerInRole("Nurse")) && eHR.getWard() == clinician.getWard())
        throw new SecurityException("No sufficient access rights.");
    eHR.setPrescriptions(pres);
}

public String readPatientData(String userID, String ehrID){
    Clinician clinician = getClinician(userID);
    EHR eHR= getEHR(ehrID);
    if !(ctx.isCallerInRole("Physician")  && eHR.getWard() == clinician.getWard())
        throw new SecurityException("No sufficient access rights.");
    String patientData =
        ehr.getAdministrativeData() + ehr.getPrescriptions() + ehr.getDiagnosis();
    return patientData;
}

public void sendDiagnosis(String userID,  String ehrID, Key key){
    EHR eHR = getEHR(ehrID);
    Clinician clinician = getClinician(userID);
    if !(ctx.isCallerInRole("Physician") && eHR.getWard() == clinician.getWard())
        throw new SecurityException("No sufficient access rights.");
    byte[] signedData = signData(eHR.getDiagnosis().getBytes(), clinician.getKeyName());
    byte[] encryptedSignedDiagnosis=encryptData(signedData, key);
    String mailAddress = getMailAddress(userID);
    sendPatientDataToMailAddress(encryptedSignedDiagnosis,mailAddress);
}

byte[] signData(byte[] data, String keyAlias){
    mSignature = Signature.getInstance("SHA256withDSA", "SUN");
    KeyStore ks = KeyStore.getInstance("Hospital");
    FileInputStream ksfis = new FileInputStream(ksName);
    BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
    ks.load(ksbufin);
    PrivateKey priv = (PrivateKey) ks.getKey(keyAlias, null);
    mSignature.initSign(priv);
    mSignature.update(data)
    return mSignature.sign();
}

byte[] encryptData(byte[] data, Key key){
    mCipher = Cipher.getInstance("AES");
    mCipher.init(Cipher.ENCRYPT_MODE, key);
    return mCipher.doFinal(data);
}

HttpsURLConnection openSSLConnection(URL url) throws IOException, GeneralSecurityException {
    TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
    tmf.init(getKeystore());
    SSLContext context = SSLContext.getInstance("TLS");
    context.init(null, tmf.getTrustManagers(), null);
    tmf.HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();
    urlConnection.setSSLSocketFactory(context.getSocketFactory());
    urlConnection.setHostnameVerifier(new AllowAllHostnameVerifier());
    return urlConnection;
}
```

Fig. 2. Excerpt from the source code of a fictitious clinical information system.

```
1  /*@ public normal_behavior
2      requires ctx.isCallerInRole("Physician")  &&
3        getEHR(ehrID).getWard() == getClinician(userID).getWard();
4      also
5      public exceptional_behavior
6      requires !(ctx.isCallerInRole("Physician")  &&
7        getEHR(ehrID).getWard() == getClinician(userID).getWard());
8      signals_only SecurityException;
9    @*/
10 public void writePrescriptions(String ehrID, String userID, String pres){
11     Clinician clinician = getClinician(userID);
12     EHR eHR= getEHR(ehrID);
13     if !((ctx.isCallerInRole("Physician") || ctx.isCallerInRole("Nurse")) && eHR.getWard() == clincian.getWard()))
14        throw new SecurityException("No sufficient access rights.");
15 }
16
17 /*@ ensures mCipher.getInput().equals(mSignature.getOutput()) && mCipher.getKey().equals(key) &&
18      mCipher.getAlgorithm().equals("AES/CBS/PKCS5Padding") &&
19      mSignature.getInput().equals(ehr.getDiagnosis()) && ...;
20   @*/
21 public void sendDiagnosis(String userID,  String ehrID, Key key){
22     EHR eHR = getEHR(ehrID);
23     Clinician clinician = getClinician(userID);
24     if !(ctx.isCallerInRole("Physician") && eHR.getWard() == clinician.getWard())
25         throw new SecurityException("No sufficient access rights.");
26     byte[] signedData = signData(eHR.getDiagnosis().getBytes(), clinician.getKeyName());
27     byte[] encryptedSignedDiagnosis=encryptData(signedData, key);
28 }
29
30 /*@ ensures mCipher.getInput().equals(data) && mCipher.getKey().equals(key) &&
31             mCipher.getAlgorithm().equals("AES/CBS/PKCS5Padding") && mCipher.getOutput().equals(data);
32   @*/
33 byte[] encryptData(byte[] data, Key key){
34     mCipher = Cipher.getInstance("AES");
35     mCipher.init(Cipher.ENCRYPT_MODE, key);
36     return mCipher.doFinal(data);
37 }
```

Fig. 3. Annotated sliced source code.

by physicians who are on the same ward as the patient." (see also `Req 2`) is shown in Fig. 3 as a JML specification (lines 1-9). The method behaves normally if the appropriate roles have been activated and the additional context constraint is satisfied; otherwise, a security exception is thrown.

Other security requirements can be formulated similarly, in particular those that are related to cryptography. For example, consider the JML annotation of the method `encryptData()` in Fig. 3 (see lines 29-31). It states that `encryptData()` ensures that `data` is encrypted with the symmetric key `key` and the algorithm `AES/CBC/PKCS5Padding`.

In addition, we can see that the specifications are directly defined within the context of the corresponding methods. This allows for more concise specifications and hence is an advantage compared to rule languages of current source code analyzers as West and Chess from Fortify concede ([11], page 99). The rule language of the static code analyzer Fortify SCA, for example, must indirectly define the context of the rule to be applied. This may become tedious.

2) Slicing: We mentioned before that ESC/Java2 is built on an automatic theorem prover. This task is costly, in particular, if Java libraries such as Java container classes are heavily used. These libraries must also be annotated, which leads to the so-called "specification creep" problem [29]. Furthermore, verification conditions that must be internally processed by the prover can become extremely large and hence cannot be proven [29]. Furthermore, Lloyd and Jürjens point out in the context of a different case study, a biometric authentication system, that it is was difficult to check JML-annotated methods because the application's methods were too large ([45], page 89). For this reason, we need a way to narrow down the code to locations that implement the security functionality. Here, the slicing step comes into play, which allows us to automatically extract these relevant locations.

Enabling static checking is not the only motivation for slicing. It is also helpful for program comprehension tasks. For example, one can separate out permission-enforcement code or the implemented crypto mechanisms and attempt to understand these specific code views. Moreover, if one attempts to automatically infer JML annotations from code, many unrelated code annotations will be generated. Slicing reduces the number of these annotations.

In our approach we employ context-sensitive interprocedural backward slicing. Interprocedural slicing allows us to follow dependences through called methods. This gives a security analyst a more comprehensive view on the security-relevant code. For example, starting from the slicing criterion in method `encryptData()` (see Fig. 2, line 57)

```
return mCipher.doFinal(data);
```

the slice also includes the `sendDiagnosis()` method and follows `signData()` (see line 26).

High-level algorithm: Static Analysis of an Application against its Security Requirements

Input: The sources of an application and the employed Security APIs.

Step 0: Annotate the Security APIs with DBC specifications.

Step 1: Load the application's source code into an SDG to enable program analysis.

Step 2: Search for calls of the Security API on the SDG
and add them to the slicing criterion (automated step).

Step 3: Do context-sensitive, interprocedural backward slicing w.r.t. the slicing criterion.

Step 4: Create a new source file with the sliced version of the application.

Step 5: Insert DBC annotations for each method by employing an automated inference mechanism, such as Daikon.

Step 6: Call an Extended Static Checker on the annotated code.

Result: A static checker's report on specification violations of the specified security requirements.

Fig. 4. Overall algorithm for abstracting and analyzing the implemented security policy of an application.

3) Specification Inference: One open point of our approach is the burden of annotating code, which has been reported to be tedious in many cases [29]. We address this problem by employing a dynamic approach, which, for example, is implemented in the Daikon tool [25]. Daikon automatically infers likely specifications by instrumenting the program under analysis. The quality of the specifications depends on the test cases that are used for instrumentation.

In order to make this approach work, we must consider the following aspects due to the nature of the annotations we plan to infer:

- We must capture data at exceptional program exits to generate exceptional and non-exceptional cases (heavy-weight JML specifications). Daikon does so and it has separate data for each type of exception thrown. It then does machine learning to determine which behaviors are associated with which thrown exceptions.
- The specifications contain `pure` method calls, such `isCallerInRole()`. One needs to tell Daikon which methods are `pure` such that those methods are considered for inferring JML annotations.
- We must support conditional or disjunctive invariants. Daikon can compute these, although it often requires a bit of assistance from the user in order to produce good disjunctions.

We expect that the inferred specifications contribute to a better understanding of the implemented security mechanisms. As the aforementioned approach can only provide suggestions for specifications, the source code must still be regarded during analyses. In the end, we need a way to evaluate the human effort saved by these automated tools versus the effort that might have been required for traditional unassisted code review. This task, for example, can be done by defining controlled experiments [16] where one group uses the proposed techniques, whereas a second control group uses traditional code review.

4) Steps of the Proposed Approach: Fig. 4 depicts the single steps of our proposed analysis approach. In the first step, we generate an internal representation of the code (IR), suitable for program analysis, such as the SSA form [3] and SDGs [37]. Thereafter, the tool searches for Security API call statements (e.g., `mCipher.doFinal(data)`) on the IR and collects them into *one* common slicing criterion. Since an analyst does not need to enter the slicing criterion on her own, this leads to a higher degree of automation. Here, our technique utilizes knowledge of the applied Security API that is preloaded into the slicer.

From the criterion, we conduct backward slicing, i.e., find all the program statements that influence the slicing criterion. We then obtain all the statements on which the Security API calls depend. We refer to this sliced code as the "implemented security architecture" of the analyzed application. In Fig. 3, one can see that the method `sendPatientData()` has been sliced. The slicing criterion is the `doFinal()` call of the `encryptData()` method (see Fig. 2, line 57). In particular, the lines 38 and 39 from Fig. 2 have been removed. In addition, since slicing respects data and control dependences, we can track which data are really encrypted by the `doFinal(data)` call and find out that they are the signed diagnosis data (see line 36, Fig. 2). Similarly, the method `writePrescriptions()` shown in Fig. 3 has been sliced with the criterion `ctx.isCallerInRole("Physician")`. Furthermore, it is important to note that

slicing w.r.t. Security APIs can also be used as a stand-alone task for security program comprehension.

In the next step, we annotate the sliced application with DBC specifications, which are based on the security requirements of the application. This can be done manually or automatically with tools such as Daikon. As the last step, we check the annotations against the sliced code by means of Extended Static Checking. Please note that we rely on modular reasoning here, a key concept of Extended Static Checking as indicated in Section II-A. In particular, the Security API does not need to be verified; we even do not provide the code of the API's methods, but can use the `//@ assume` statement made available by ESC/Java2. This statement, for example, allows ESC/Java2 to assume postconditions of a method without proving them, e.g., we do not have to prove the correctness of the `isCallerInRole()` method.

In a prerequisite step ("Step 0"), DBC annotations for the Security APIs must be provided. This task must be done manually and hence requires some effort. However, it needs to be carried out less often than annotating applications. Usually, this is the case when the initial version of a Security API is made available or when the Security API changes.

5) Discussion of the Approach with the Help of Code Examples: To give an example of predefined annotations for Security APIs, please consider the method `doFinal()` of the class `javax.crypto.Cipher`. We can define a postcondition for this method as follows:

```
/*@ ensures
     this.getInput().equals(input)
     && this.getKey().equals(key)
     && this.getAlgorithm().equals(algorithm)
     && this.getOutput().equals(\result);
  @*/
```

It states that `getOutput()` returns the result of encrypting `input` with the parameters `key` and `algorithm`. Methods such as `getInput()` and `getKey()` are specification-only JML model methods, which can also be used by clients of the `doFinal()` method to make specification more readable. An extended static checker can also use this specification to prove the postcondition of the `encryptData()` method (see Fig. 3, line 29). This implies that `Cipher.init()` and `Cipher.getInstance()` must have been called before with the appropriate parameters for the key and the algorithm.

In particular, the code in Fig. 2 contains several vulnerabilities. The method `writePrescriptions()` does not implement `Req 2` correctly, i.e., nurses can also write prescriptions.

Furthermore, the hostname verifier in the SSL code is set to an instance of `AllowAllHostnameVerifier`. This class essentially turns hostname verification off. Even software vendors with a well-defined Security Development Lifecyle (SDL) follow such practices, e.g., SAP, who built this code into a mobile communication library such that more than ten (partly security-critical) apps were vulnerable.

To detect such situations, the `setHostnameVerifier()` method of the `HttpsURLConnection` defines a precondition as follows:

```
/*@ requires
      v instanceof StrictHostnameVerifier ||
      v instanceof BrowserCompatHostnameVerifier;
 @*/
```

This precondition assures that the full implementation for hostname verification is used.

A further vulnerability can be found in the statement (see Fig. 3, line 33)

```
mCipher = Cipher.getInstance("AES");
```

In this code, neither the encryption mode nor a padding scheme is defined. Depending on the installed Java crypto provider, the electronic code book mode (ECB) could be the default, which is known to be insecure [2].

If the `Cipher.getInstance()` method contains the annotation

```
//@ requires this.getAlgorithm().equals("AES/CBC/
     PKCS5Padding");
```

then an Extended Static Checker automatically identifies the aforementioned issue.

As a further observation, JML preconditions tend to correlate to rules that guarantee the secure usage of an API method. For example, the `getInstance()` method may have a precondition stating that the CBC mode should be used with AES encryption. The same remark applies to the JML precondition for the `setHostnameVerifier()` API method. Since preconditions are required to be satisfied by the caller, they can be conveniently provided by the called library (e.g., in a knowledge base).

JML postconditions, such as JML heavyweight specifications and `ensures` statements, let one express application-specific security requirements. Application-specific requirements include the role-based policy or requirements stating which data are to be encrypted or signed by which key.

C. Example Security Libraries

We subsequently give three examples of Security APIs beyond Java and JEE security that underline the relevance and generality of our idea. These examples reflect different aspects of application security.

Web-based authorization APIs: The Spring software framework, for instance, makes available certain authorization API methods such as `hasRole()` [53] as shown in the following annotated code fragment:

```
/*@ public normal_behavior
    requires securityExpr.hasRole("Manager") ||
      securityExpr.hasRole("Financial Officer");
    also
    public exceptional_behavior
    requires !(securityExpr.hasRole("Manager") ||
      currentUser.hasRole("Financial Officer"));
    signals_only SecurityException; @*/
public int getBalance(){
   if(!(securityExpr.hasRole("Manager") ||
                securityExpr.hasRole("Teller")))
```

```
        throw new SecurityException("Access Denied");
    return balance;
}
```

The access control check allows the method to be successfully completed only if the caller has activated the appropriate roles. The `hasRole()` calls correspond to `isCallerInRole()` calls and can be automatically extracted from the code by slicing. The annotations are similar to those given in Fig. 3. Other Security APIs with similar security features for JEE-based web applications are Apache Shiro [57] and ESAPI [52].

One note should be made on declarative access control, which is widely-used in applications that employ Java-based software frameworks such as JEE or Spring. In a preprocessing step, the configuration files containing the role-method assignments (e.g., deployment descriptors) can be parsed. Then `//@ assume` statements with appropriate role checks can be inserted at the beginning of the corresponding methods referred to in the configuration files. For example, if the deployment descriptor requires the role "Teller" for executing method `getBalance()`, we can place the following statement at method entry:

```
//@ assume currentUser.hasRole("Teller");
```

This step gives an analyst a unified view on the implemented access control mechanisms in the analyzed application. So, we can also cover declarative access control.

Java Trusted Software Stack (jTSS): A different kind of Security API are libraries for accessing security hardware, such as Trusted Platform Modules (TPMs) [9]. One such library is jTSS, which is an implementation of the TCG Software Stack for Java [39]. Specifically, jTSS provides security functionality to measure the hardware and software status of IT systems trustworthily, using secure storage and different signing keys. Based on these security features, the implementation of many security-critical applications is conceivable. For instance, we can build systems which provide digital evidence (e.g., to be used at court) based on a TPM [54]. This system must then ensure non-repudiation requirements. In case of a dispute, it must not be possible to mistrust this "digital evidence". Due to the fact that APIs for TPMs tend to be quite complex, flaws in the application are conceivable which stem from the incorrect usage of the API and may lead to a violation of non-repudiation requirements.

Our approach can support an analyst in asserting that the application actually implements a digital-evidence system correctly. To implement requirements, such as "evidence data must be signed with a non-migratable 2048-bit RSA platform key, and this key must be bound to the current software state of the platform", jTSS makes available a series of API methods. E.g., `TcIRsaKey.createKey()` enforces the binding of the signing key to the system configuration of the platform. Slicing can extract these calls automatically and Extended Static Checking lets one verify that the appropriate API methods have been called in the right order with the correct parameter values.

Android SDK: Android has become one of the most prominent smartphone platforms today. This is one reason why it has attracted much attention in the security research community. The Android Framework provides a rich set of APIs, which allows a developer to implement small Java-based applications called "apps", which can be downloaded from application repositories. Android apps usually consist of components, such as activities (which implement the user interface of an application) or services (which carry out background jobs).

Researchers found out that many apps showed weaknesses [13], [23], [34], [26], [46]; in some cases, an attacker could even execute system permissions [34]. Some vulnerabilities were caused by the erroneous usage of interprocess communication (IPC). Android uses IPC for the communication between apps, which are otherwise separated through different Linux user IDs. Typically, a data structure called "intent" is used on performing an IPC. Intents are responsible for exchanging data and defining target addresses of the IPC.

If, for example, a mobile application does not appropriately protect its components, other apps might have undesirable access. In addition, intents must be secured; otherwise, attacks such as Activity and Service hijacking are possible. Similarly, if broadcast messages are not adequately protected, eavesdropping or denial of service attacks are conceivable [13], [23]. A typical security rule to be respected by Android apps is "always specify an access permission on intent broadcasts if the target component has not explicitly been defined and the intent contains extra information". A JML precondition can then be defined as follows:

```
//@ requires (intent.getExtra()!= null &&
              intent.getComponent()== null &&
              intent.getClass()== null &&
              intent.getPackage()== null) ==>
                              broadcastPerm!=null;
```

Please note that we demand that the intent has an `extra` field because this is additional information that may be sensitive. The caller of the `sendBroadcast()` API method must then assure that this precondition is satisfied. The Online Manager app from the Deutsche Telekom[1], for example, shows behavior that violates this rule:

```
Intent localIntent = new Intent("de.telekom.hotspot.intent.
    action.SMS_STATUS");
localIntent.putExtra("status", CredentialSmsStatusType.
    SMS_STATUS_CREDENTIALS_RECEIVED);
localIntent.putExtra("username", paramString1);
localIntent.putExtra("password", paramString2);
paramContext.sendBroadcast(localIntent);
```

It sends the hotspot password of their clients per broadcast message. Although meant as a private message for the internal app components, it erroneously published the intent to all other installed apps because no broadcast permission has been specified. We reported this flaw to the developers and found out that they had not understood the security ramifications of

[1] https://play.google.com/store/apps/details?id=de.telekom.hotspotlogin.de, more than 1 Mio. downloads

the Android Framework. This underlines the need of tools that encode security knowledge on software frameworks. Tools like ComDroid [13] and CryptoLint [22] could help here, but they are very specific research tools and hence do not provide a unified and encompassing solution to the problem of misusing APIs.

Android also supports delegation concepts in the form of "pending intents" [24]. If the recipient of a pending intent has not explicitly been set, it can be further delegated to an unanticipated destination, leading to undesirable access to the component [23]. The corresponding rule can be specified as a JML annotation as follows:

```
//@ requires intent.getComponent()!= null ||
             intent.getClass()!= null ||
             intent.getPackage()!= null;
```

In the app SAP CRM Service Manager[2] code can be found where both an intent (for starting a service component) and a pending intent (broadcast intent) are not secured:

```
1  String str = Values.getInstance(this).getGcmProductId();
2  Intent localIntent = new Intent("com.google.android.c2dm
       .intent.REGISTER");
3  localIntent.putExtra("app", PendingIntent.getBroadcast(
       this, 0, new Intent(), 0));
4  localIntent.putExtra("sender", str);
5  startService(localIntent);
```

The intent defined in line 2 is insecure and hence can be accessed by another app. The attacker then obtains a pending intent, which is the payload of the insecure intent (see line 3) and which is a broadcast intent. This pending intent is also insecure such that an attacker can fill in any data of interest and send the broadcast. The SAP CRM Service Manager app must assume that this broadcast has come from one of its own components and calls its corresponding broadcast receiver. This receiver, which handles access to a cloud service, is protected by a signature permission. However, as the attacker controls the pending intent, she has inadvertently access to the receiver.

In summary, we have discussed different Security or security-relevant APIs in the context of the proposed technique. It is important for an application to use these APIs correctly to meet its security requirements. An analysis tool that supports the advanced analyses should foster knowledge of these APIs in a knowledge base (in the form of code annotations). This aspect will be discussed in Section V.

IV. Implementation and Application of the Analysis Approach

We first describe the proof-of-concept implementation of our analysis approach. Thereafter, we report on our experience with the implemented tool while carrying out two case studies. We selected two different application areas, JEE-based web applications and the Android Framework, to demonstrate the applicability and generality of our technique.

[2]https://play.google.com/store/apps/details?id=com.syclo.sap.SAPServiceMgr.client.android

A. Implementation Aspects

1) Slicing with Wala: We used the slicer provided by the byte code analysis framework Wala [20] as the basis for our implementation. We also made available a tool that lets an analyst enter slicing criteria via a graphical user interface. Depending on the use case, an analyst can select predefined security-critical APIs, e.g., `doFinal()`, `sign()` or `isCallerInRole()` as slicing criteria. Thereafter, our tool internally searches the call graph part of the SDG to find all seed statements for slicing (see also Section III-B2). Having automatically collected all these statements, the tool calls the Wala slicer.

We had to perform some optimizations to make slicing feasible. In particular, we excluded about 380 classes/packages from the analysis space, e.g., Swing classes. Otherwise, the slicer would descend into the Java API implementation, which is prohibitively expensive. Furthermore, we had to set Wala's `NO_HEAP` option, i.e., data flows through the heap are not followed. This analysis is in general too expensive as also pointed out by Sridharan et al. [56].

2) Annotation Inference with Daikon: We used the Daikon tool for automatically inferring JML annotations for Java code. As Daikon needs concrete test cases that the instrumentation process runs, this task is specific to the software under analysis. For example, the test cases must contain the names of the roles if the `isCallerInRole()` API call is used. Other parts of the test cases, however, are only dependent on the applied software framework or Security API, e.g., setting parameters for defining the encryption and signature algorithms. As mentioned in Section III-B3, one important challenge was to detect the different normal and exceptional exit paths of methods to generate heavyweight JML specifications.

B. Case Studies

JEE-based Web Application: We have implemented a demonstration web application, which is based on the code depicted Fig. 2. The aim of this evaluation step is to demonstrate that our tool can deal with different Java-based Security APIs, such as Java cryptography, SSL/TLS functionality, and JEE-based authorization.

One important aspect of the slicing task is that it can be used for security program comprehension. For example, an analyst can follow the slice from the `sign()` statement to the code location where the private key is obtained. In case of the JEE application the key has been retrieved from the Java keystore:

```
PrivateKey priv = (PrivateKey) ks.getKey(keyAlias, null);
```

In particular, one can see that the keystore is not secured by a password (`null` parameter), although a sensitive private key is stored there. Furthermore, a JML precondition can be specified for the `getKey()` API as follows:

```
\\@ requires \typeof(\result)==PrivateKey
                                  ==> password != null;
```

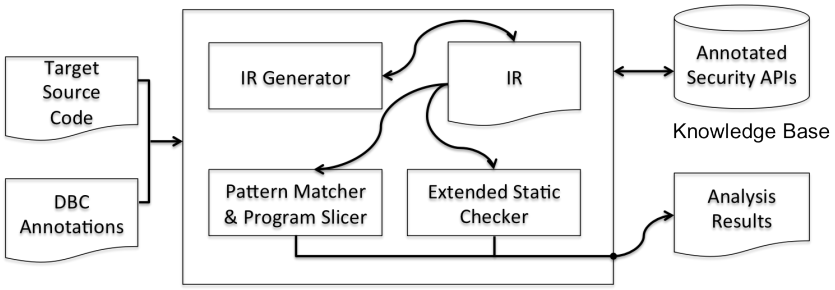

Fig. 5. Analysis infrastructure.

```
1  /*@ public normal_behavior
2    @ requires checkCallingOrSelfPermission(GRANT_REVOKE_PERMISSIONS)== PERMISSION_GRANTED;
3    @ also
4    @ public exceptional_behavior
5    @ requires !(checkCallingOrSelfPermission(GRANT_REVOKE_PERMISSIONS) == PERMISSION_GRANTED);
6    @ signals_only SecurityException;
7    @ */
8  public void revokePermission(String packageName, String permissionName) {
9    final PackageParser.Package pkg = mPackages.get(packageName);
10   if (pkg.applicationInfo.uid != Binder.getCallingUid())
11     mContext.enforceCallingOrSelfPermission(GRANT_REVOKE_PERMISSIONS, null);
12 }
```

Fig. 6. Annotated and sliced source code of Android's `PackageManagerService`.

Again this shows that rules for Security APIs can be naturally specified as JML preconditions.

Beyond the aforementioned slicing experiments, we also produced variants of the code. The purpose of this mutants-based testing approach [40] is to check for false positives and negatives. False positives are statements that occur in the slice, but are not related to the analyzed security mechanism, whereas we speak of false negatives when security-relevant code does not occur in the slice. False positives only make the slice larger, whereas false negatives may lead to misunderstanding of the code. If Extended Static Checking is used after slicing, then even flaws may be missed.

The test scenarios include the following cases:

- insertion of unrelated statements concerning local variables of primitives Java types,
- insertion of unrelated statements concerning local variables of class types,
- accessing unrelated member fields,
- introduction of further private helper methods (e.g., we introduced an additional helper method `getPrivateKey()` to retrieve the private signature key from the Java keystore).
- adding pointer assignment statements that depend on the slicing criterion, e.g., line 2 in the following code:

```
1 localCipher= Cipher.getInstance("AES");
2 mCipher = localCipher;
3 mCipher.init(Cipher.ENCRYPT_MODE, key);
4 return mCipher.doFinal(data);
```

We considered the `sign()`, `doFinal()`, and the `isCallerInRole()` statements as slicing seeds for the test cases.

In total, we generated 27 test cases for evaluating the slicing step. Of all test cases 12 false positives occur, whereas no false negatives have been found (0.56 recall, 1.0 precision). The false positives occur when objects are accessed; access to variables of primitive Java types does not contribute to this rate.

Although we did not find any false negatives during evaluation, we discuss a situation where a false negative is possible in Section V. This situation is more fundamental as it is inherent in our general approach and not in the slicing implementation—thus it is a different kind of false negative.

C. Android Framework

The second case study shall demonstrate that our approach can be applied to real-world code. Target of our analyses is the Android Framework, specifically, Android system services. In this respect, we follow a case study, which has been discussed elsewhere [49], but extend it in several ways. First, we can now analyze different Android versions and have carried out analyses of Android 1.5 until Android 5.0—the approach followed in [49] only considered Android 4.0.3. Furthermore, we are able to analyze all Java-based system services, which are about 35.

The Android Framework makes available authorization APIs, which mobile applications can use to enforce more fine-grained access control policies [24]. Examples are the `checkCallingPermission()` and `checkCallingOrSelfPermission()` methods in the `android.Context` namespace. These methods serve a similar purpose as the `isCallerInRole()` method from JEE. The implementation of the Android Framework heavily uses these authorization APIs, with more than 400 calls within system

services and content providers. Fig. 6 displays annotated code from the `PackageManagerService`, a security-critical system service of the Android platform that administers all the packages of Android apps including their permissions. In general, these authorization APIs can be used as slicing criteria to better understand the implemented permission model of Android.

In the following, we discuss some experiments that we carried out with our analysis infrastructure regarding the analysis of the implemented permission model in Android system services. From Table I and Table II one can conclude for selected system services and Android versions that slicing reduces the code size to less than 10% when considering the access control policy. Moreover, Table III shows the development of the permissions along different Android versions. One can see that new permissions have been introduced that correspond to new security-relevant Android functionality. For example, the `revokePermision()` API was only a *local* method before Android 4.1.2. It is now exported to allow for the revocation of permissions at runtime and requires the permission `GRANT_REVOKE_PERMISSIONS`. It still is an undocumented feature that is exported as a hidden API and can only be accessed via Java reflection. Other permissions that are enforced in the `PackageManagerService` and that cannot be granted to third-party apps are `MANAGE_USERS` and `INTERACT_ACROSS_USERS_FULL`. The former allows the management of multiple users on the smartphone, whereas the latter allows communications between different smartphone users. Our approach contributes to a better understanding of such "hidden features" and the corresponding access control policy. In particular, the inference of JML annotations makes explicit the access control policy for such undocumented mechanisms. Daikon helped us infer specifications as given in Fig. 6. Other approaches, such as PScout [5], that construct a permission map for Android do not cover hidden features, which are only available to the system or dedicated system apps.

The focus of this case study lies more on the comprehension of the implemented security mechanisms complex software system rather than analyzing software w.r.t. secure usage of the software frameworks. Since Google developers use their own framework to implement critical system services, we do not expect that relevant security holes can be found here with the help of tools. This is more to be expected in third-party Android apps or JEE-based web applications where adequate security knowledge is often sparse. In that case, the JML-based annotations come into play, which codify security knowledge of Security APIs.

V. Discussion

We now discuss limitations as well as prospects of our analysis approach including tool support, educational aspects, and related software initiatives.

Tool Support and its Current Limitations: One reason why we concentrate on JML in this paper is the rich tool set available (see Section II-A). Although ESC/Java2 in particular is quite mature and supports most of the JML features, such as model methods, it has limitations, which make it difficult to apply in industrial contexts. First, only Java versions up to Java 1.4 are supported, i.e., Java generics cannot be dealt with. Second, the problem of Extended Static Checking is undecidable, i.e., the tool will produce false positive and negatives, but with a moderate rate [7].

To improve Extended Static Checking, there are currently ongoing efforts for building a new extended static checker for Java within the OpenJML initiative[3]. At the time of this writing, however, this tool does not completely implement heavyweight JML specifications [7], which are needed to express exceptional behavior and which we use for the analysis of access control checks. When this problem has been addressed, we hope that our approach can be applied to larger case studies in industrial contexts in the near future. In particular, advanced JML concepts, such as *model* features [7] as well as complex Java data structures (e.g., Java container classes using generics), are then better supported. This newer extended static checker is expected to leverage more powerful backend SMT solvers such as Yices [21] and Z3 [18].

Fig. 5 displays a possible architecture of our proposed analysis infrastructure. Input is the code under analysis. Moreover, the tool infrastructure contains a knowledge base which stores information about the interface of different Security APIs and software frameworks as well as their annotations. We further assume that we have a common IR for the different tasks. Program slicers, such as Wala (Java) [20] and CodeSurfer (C) [1], and Extended Static Checkers work on similar IRs. Developing such a common analysis infrastructure would require a substantial engineering effort, but in the end, it would lead to a better tool integration.

Limitations of the Approach: Our approach itself has some limitations. For example, program slicing possibly eliminates statements that are relevant for the security of the application. Consider the method `sendDiagnosis()` in Fig. 2. If we use the cryptographic and programmatic access control API calls as slicing criteria, then the statement

```
sendPatientDataToMailAddress(
          encryptedSignedDiagnosis, mailAddress);
```

will be ignored. If the parameter `encryptedSignedDiagnosis` contained only unencrypted data, then we would not detect this flaw because the statement would not be in the slice. An analyst could add this statement manually, but this would increase her workload.

Similarly, missing access control checks are also ignored by our technique as the following code shows:

```
1 public String readPrescriptionsWOCheck(String ehrID,
      String uID){
2   Clinician clinician = getClinician(uID);
3   EHR eHR = getEHR(ehrID);
4   if(eHR!=null)
```

[3]http://jmlspecs.sourceforge.net/

System Service (Source Code)	2.2.2	4.0.3	4.1.2	4.2.2	4.3.1	4.4.2
ActivityManagerService	14529	14609	15193	14567	14890	16415
BackupManagerService	2519	5642	5715	5756	5901	6056
DevicePolicyManagerService	944	2032	2042	2313	2481	2825
LocationManagerService	1885	2216	2459	1986	2156	2319
PackageManagerService	9839	8525	9383	10047	10805	11402
WindowManagerService	11417	9760	9999	11031	10399	10790
Total	41133	42784	44791	45720	46632	49807

TABLE I
DEVELOPMENT OF SELECTED SYSTEM SERVICES OVER DIFFERENT ANDROID VERSIONS (ORIGINAL SOURCE CODE).

Systemservice (Slice)	2.2.2	4.0.3	4.1.2	4.2.2	4.3.1	4.4.2
ActivityManagerService	2716	2064	2155	2427	3109	1449
BackupManagerService	91	98	98	92	98	98
DevicePolicyManagerService	35	35	35	40	43	72
LocationManagerService	25	25	25	11	11	11
PackageManagerService	112	148	189	215	237	199
WindowManagerService	274	437	310	370	−−	−−
Total	3253	2807	2812	3155	3498	1829

TABLE II
DEVELOPMENT OF SELECTED SYSTEM SERVICES OVER DIFFERENT ANDROID VERSIONS (SLICES).

Permissions	2.2.2	4.1.2	4.4.2
CLEAR_APP_CACHE	✓	✓	✓
DELETE_PACKAGES	✓	✓	✓
CLEAR_APP_USER_DATA	✓	✓	-
DELETE_CACHE_FILES	✓	✓	✓
GET_PACKAGE_SIZE	✓	✓	✓
GET_PREFERRED_APPLICATIONS	✓	-	-
SET_PREFERRED_APPLICATIONS	✓	✓	✓
MOVE_PACKAGE	✓	✓	✓
WRITE_SECURE_SETTINGS	✓	✓	✓
GRANT_REVOKE_PERMISSIONS	-	✓	✓
INSTALL_PACKAGES	-	✓	✓
CHANGE_COMPONENT_ENABLED_STATE	-	✓	✓
PACKAGE_VERIFICATION_AGENT	-	✓	✓
MANAGE_USERS	-	-	✓
INTERACT_ACROSS_USERS_FULL	-	-	✓

TABLE III
PERMISSIONS OF THE PACKAGEMANAGERSERVICE IN SELECTED ANDROID VERSIONS.

```
5     return eHR.getPrescriptions();
6   else return "";
7 }
```

Slicing would remove this method from the analysis space because there is no security check. To handle this case, we can first identify all security-critical data and methods, such as `EHR.getPrescriptions()`. Thereafter, we can introduce a Boolean variable for each security-critical asset, e.g., `accessGetPrescriptions`. Also, set this flag at the beginning of the respective method to `false`. When the asset is accessed, then replace this access by the assignment

```
accessGetPrescriptions=true;
```

After employing this approach, we obtain:

```
1 public String readPrescriptionsWOCheck(String ehrID,
      String uID){
2   accessGetPrescriptions=false;
3   EHR eHR= getEHR(ehrID);
4   if(eHR!=null){
5     accessGetPrescriptions=true;
6     return "";
7   }
8   else return "";
9 }
```

Thereafter, we can add this assignment statement to the slicing criterion and need to check an invariant of the form:

```
//@ invariant accessGetPrescriptions == true
                        => security_check == true;
```

This invariant globally[4] ensures that the corresponding security check is true if the security-critical resource is accessed. The approach presumes that a software architect with knowledge of critical data is involved in this analysis/annotation process.

In summary, our technique not necessarily detects all security-relevant code locations. However, our approach is meant to alleviate the work of QA such that they can conduct more effective security code reviews than today. The tools shall not replace the security expert.

Educational Aspects: Our proposed technique can also be viewed from the educational perspective. First, developers tend to roll out their own security features rather than using well-tested security functionality [47]. As software vendors adopt static code analyzers (hopefully, not only for reasons of due diligence of the management) and the topic of software security is widely taught at universities, Security APIs will finally be employed to a larger extent than today. Second, our tool should cover educational aspects, e.g., it could give explanations when the API is used insecurely. Similarly, tools as Fortify SCA currently explain the kind and nature of the security problem in case a possible vulnerability is flagged. As Chess and West point out, didactic aspects have contributed to the success of static code analyzers [11]. Didactic support leads to a better acceptance of the tools.

Security Views: Security APIs and software frameworks often cover quite different security aspects, such as crypto, SSL functionality, access control, authentication, or security for IPC functionality in case of Android. For this reason, it would be desirable to let an analyst select each specific aspect she wants to analyze. Slicing would then extract a specific security view on the software, e.g., an access control view or a view on IPC. These views could be selected either via a graphical user interface or specific configuration files.

In this context, it should be clarified which stakeholder of an SDL can use this tool. Certainly, a developer mostly does not have the security knowledge to apply such a tool. Large vendors, however, often have security-aware members in the development teams as Fichtinger et al. report on the SDL of Siemens [28]. This position seems to be well-suited for such a task. Furthermore, support from the central product CERTs could also help here.

Benefits for Common Criteria Projects: Our approach can be useful for Common Criteria evaluation projects. The Common Criteria demand for medium to high assurance levels (EAL 4 upwards) evidence that the implementation corresponds to the specification of the security functionality. This assurance requirement is known as `ADV_IMP` according to part three of the Common Criteria documents [15]. A completely manual code review is difficult to carry out both for QA and Common Criteria evaluators. Employing COTS static analyzers does not solve this problem as they focus on common implementation bugs. Applying our approach to Common Criteria projects, we can extract the implemented architecture automatically and pinpoint critical code regions. Extended static checking can then be employed for conformance checking. Since the more widely-used levels EAL 4 and 5 only require one to show conformance for *parts of the code* rather than the complete software, slicing is well-suited for such a project. Equipped with a knowledge base of specifications (see Fig. 5), the evaluator can cover more security aspects than relying solely on her own knowledge.

Recent Software Security Initiatives: Our approach also relates to recent software security initiatives like the Building Security In Maturity Model (BSIMM). BSIMM is supported by large software vendors, among them, SAP, Microsoft, and Adobe. It defines best practices, which organizations can follow to secure their applications. One BSIMM activity is to provide secure components, which correspond to our term "Security APIs/libraries" (see also [6], SFD 2.1 "Build secure-by-design middleware frameworks and common libraries").

BSIMM suggests to define code review rules, which support QA representatives in checking whether the secure components are used correctly. In particular, the BSIMM documentation says *Eventually the SSG can tailor code review rules specifically for the components it offers.* [6]. It further concludes *Generic open source software security architectures, including OWASP ESAPI, should not be considered secure out of the box.* Our approach can help here by providing DBC specifications for the Security APIs, administering these annotations in a knowledge base and providing adequate tool support for checking these rules.

VI. Related Work

Static security analysis of software has evolved into an active research area over the years. There are several works on static checking for software security [44], [4], [12], [10], [27], [22]. Important research prototypes from static analysis are e.g. MOPS [10], Eau Claire [12], and LAPSE [44]. MOPS uses temporal logics as formalism and model checking to discover issues such as race conditions in C programs. The tool xg++ by Ashcraft and Engler was used to detect vulnerabilities in the Linux Kernel [4]. Moreover, there is a work by Livshits and Lam who present a tool to detect common low-level vulnerabilities, such as SQL injection vulnerabilities, in Java applications based on points-to analyses [44]. Felmetsger et al. employ the Daikon tool [25] to dynamically infer security specifications for web applications. Thereafter, they use a model checker to detect application logic vulnerabilities violating the specifications [27].

Similarly to our approach, the CryptoLint tool uses program slicing [22]. This tool aims to detect the misuse of cryptographic APIs in Android applications, but does not focus on the more general aspect of security program comprehension.

Some of the research prototypes evolved into commercial tools such as Fortify SCA [30] and Coverity Prevent [17]. Most of the aforementioned approaches and tools focus on finding common kinds of low-level security bugs. Our approach is complementary to them because we extract the implemented security architecture from the code and check it against DBC

[4]i.e., it holds at the end of all public methods

specifications. In particular, we focus on detecting vulnerabilities and weaknesses that are caused by the wrong usage of Security APIs. In addition, we provide an infrastructure for program comprehension w.r.t. security aspects. Last but not least, the task of automatically inferring security-relevant JML annotations from Java code is very valuable for understanding undocumented security features. COTS static analyzers like HP-Fortify SCA do not have incorporated such advanced functionality.

In contrast, threat modeling helps an analyst assess the security architecture of an application [36]. Consequently, the analyses are related to architectural documents rather than considering the source code as we do.

Other approaches deal with the topic of detecting covert channels in applications, e.g., based on non-interference properties [50]. Myers et al. introduced the JFlow language, an annotation-based extension of Java, which allows a developer to define security labels on variables. Proceeding this way, hidden information flows, e.g., induced by the control flow of the application, can be detected. Again, our proposed approach differs from this work by verifying whether Security APIs have been used correctly to satisfy the security requirements of an application; we do not consider covert channel analysis as we still face many basic security problems in software which are prevalent and demand our immediate attention.

Several related works employ DBC concepts for security analysis. Eau Claire allows the formulation of pre- and post-conditions as annotations for C code. Similarly to ESC/Java it is based on an automatic theorem prover [19]. Eau Claire detects common security problems, such as buffer overflows and race conditions. Although Eau Claire only focuses on common security bugs in C applications, it shows the benefit gained by employing static checking for security analysis.

Other case studies that use JML in the security context are presented by Lloyd et al. (a biometric authentication system) [45] and Cataño et al. (a JavaCard-based electronic purse) [8]. Both works use JML in conjunction with the static checker ESC/Java for an already implemented application. They faced problems like specification creep, annotation burden, and difficulty in generating and checking verification conditions for the underlying theorem prover. The electronic purse case study did not consider cryptographic operations. JML patterns for security have been introduced by Warnier [58]. Contrary to our proposed technique, all these approaches neither consider the relationship between DBC and Security APIs nor the aspect of security program comprehension.

VII. Conclusion and Outlook

In this paper, we pleaded for integrating the concepts of program slicing, DBC, and Extended Static Checking into future static code analyzers. Specifically, we argued that this approach is well-suited to checking whether Security APIs are used correctly by applications to implement their security requirements. We motivated this thesis with the help of several examples. We also showed that the topic of applying Security APIs correctly is more and more relevant since Security APIs of many software frameworks are quite complex.

However, to achieve a real impact on the Security Development Lifecycle, the current tool support for program slicing and Extended Static Checking must be substantially improved to obtain a seamless tool chain. As a result, software developers will have more powerful static code analyzers that complement currently available tools and lead to more systematic approaches to security code audits.

References

[1] Anderson, P., Zarins, M.: The CodeSurfer software understanding platform. In: Proc. of the 13th International Workshop on Program Comprehension. pp. 147 – 148 (May 2005)

[2] Anderson, R.: Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley, 2nd edn. (2008)

[3] Appel, A.W.: Modern Compiler Implementation in Java. Cambridge University Press (1998)

[4] Ashcraft, K., Engler, D.: Using programmer-written compiler extensions to catch security holes. In: Proceedings of the IEEE Symposium on Security and Privacy. p. 143. IEEE Computer Society (2002)

[5] Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: Analyzing the Android Permission Specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 217–228. CCS '12, ACM, New York, NY, USA (2012)

[6] Building Security In Maturity Model: Intelligence: Security Features and Design (SFD) (2012), http://bsimm.com/online/intelligence/sfd/

[7] Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int'l Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)

[8] Cataño, N., Huisman, M.: Formal specification of Gemplus's electronic purse case study. In: FME 2002. vol. LNCS 2391, pp. 272–289. Springer-Verlag (2002)

[9] Challener, D., Yoder, K., Catherman, R., Safford, D., Van Doorn, L.: A practical guide to trusted computing. IBM Press, first edn. (2007)

[10] Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: Proc. of the ACM Conf. on Computer and Communications Security. pp. 235–244 (2002)

[11] Chess, B., West, J.: Secure Programming with Static Analysis. Addison-Wesley (2007)

[12] Chess, B.: Improving computer security using extended static checking. In: IEEE Symposium on Security and Privacy. pp. 118–130 (2002)

[13] Chin, E., Porter Felt, A., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proc. of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys), Bethesda, USA. pp. 239–252. ACM (2011)

[14] Common Criteria: Common Criteria for Information Technology Security Evaluation—Part 1: Introduction and general model (2009), http://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R3.pdf

[15] Common Criteria: Common Criteria for Information Technology Security Evaluation—Part 3: Security assurance components (2009), http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R3.pdf

[16] Cornelissen, B., Zaidman, A., van Deursen, A.: A controlled experiment for program comprehension through trace visualization. IEEE Trans. Softw. Eng. 37(3), 341–355 (May 2011)

[17] Coverity: Coverity Prevent (2015), http://www.coverity.com

[18] De Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08, Springer, Berlin (2008)

[19] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)

[20] Dolby, J., Sridharan, M.: Static and Dynamic Program Analysis Using WALA, PLDI Tutorial (2010), http://wala.sourceforge.net/files/PLDI_WALA_Tutorial.pdf

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

[22] Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. pp. 73–84. CCS '13, ACM, New York, NY, USA (2013)

[23] Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proc. of the 14th USENIX Security Symposium (Aug 2011)

[24] Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. IEEE Security & Privacy 7, 50–57 (2009)

[25] Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69, 35–45 (December 2007)

[26] Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., Freisleben, B.: Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In: Proc. of the 2012 ACM Conference on Computer and Communications Security. pp. 50–61 (2012)

[27] Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, G.: Toward automated detection of logic vulnerabilities in web applications. In: USENIX Security Symposium. pp. 143–160. USENIX Association (2010)

[28] Fichtinger, B., Paulisch, F., Panholzer, P.: Driving secure software development experience in a diverse product environment. IEEE Security & Privacy 10(2), 97–101 (2012)

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

[30] Fortify Software: Fortify Source Code Analyser (2015), http://www.fortify.com/products

[31] Georgiadis, C., Mavridis, I., Pangalos, G., Thomas, R.: Flexible team-based access control using contexts. In: Proc. of the ACM Symposium on Access Control Models and Technologies. pp. 21–27 (2001)

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

[33] Google Inc.: Android Development - Requirements (2015), http://developer.android.com/sdk/requirements.html

[34] Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic Detection of Capability Leaks in Stock Android Smartphones. In: Proceedings of the 19th Network and Distributed System Security Symposium (2012)

[35] Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. 44(3), 16:1–16:58 (Jun 2012)

[36] Hernan, S., Lambert, S., Ostwald, T., Shostack, A.: Uncover security design flaws using the STRIDE approach. MSDN Magazine (Nov 2006), http://msdn.microsoft.com/en-us/magazine/cc163519.aspx

[37] Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12(1), 26–60 (Jan 1990)

[38] Huebner., G.: Personal Communication (2013)

[39] Institute for Applied Information Processing and Communications, TU Graz: IAIK jTSS - TCG Software Stack for the Java (tm) Platform (2012), http://trustedjava.sourceforge.net/index.php?item$=$jtss/readme

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

[41] Krinke, J.: Advanced Slicing of Sequential and Concurrent Programs. Ph.D. thesis, Universität Passau (2003)

[42] Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer Academic Publishers (1999)

[43] Leino, K.R.M., Müller, P.: Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs (2009)

[44] Livshits, B., Lam, M.: Finding Security Vulnerabilities in Java Applications Using Static Analysis. In: Proc. of the 14th USENIX Security Symposium (Aug 2005)

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

[46] Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: statically vetting Android apps for component hijacking vulnerabilities. In: Proc. of the 2012 ACM conference on Computer and communications security. pp. 229–240. CCS '12 (2012)

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

[48] Meyer, B.: From structured programming to object-oriented design: The road to Eiffel. Structured Programming (1), 19–39 (1989)

[49] Mustafa, T., Sohr, K.: Understanding the implemented access control policy of Android system services with slicing and extended static checking. International Journal of Information Security 14(4), 347–366 (2015), http://dx.doi.org/10.1007/s10207-014-0260-y

[50] Myers, A.C.: JFlow: practical mostly-static information flow control. In: Proc. of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 228–241 (1999)

[51] Oracle Inc.: The Java EE 5 Tutorial (2013), http://docs.oracle.com/javaee/5/tutorial/doc/bnbyk.html

[52] OWASP: OWASP Enterprise Security API (2012), https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

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

[54] Richter, J., Kuntze, N., Rudolph, C.: Security digital evidence. In: 5th IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering, Oakland, USA. pp. 119–130 (2010)

[55] springsource community: Documentation (2013), http://www.springsource.org/documentation

[56] Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. pp. 112–122. PLDI '07 (2007)

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

[58] Warnier, M.: Language Based Security for Java and JML. Ph.D. thesis, Radboud University, Nijmegen, Netherlands (2006)

[59] Weiser, M.: Program slicing. In: Proceedings of the International Conference on Software Engineering. pp. 439–449. IEEE Press, Piscataway, NJ, USA (1981)

[60] Zeller, A.: Why programs fail - a guide to systematic debugging. Elsevier (2006)