Technical Report 80

Towards Security Program Comprehension with Design by Contract and Slicing

Karsten Sohr
Tanveer Mustafa
Philipp Hirch
Markus Gulmann

TZI, Universität Bremen

TZI-Bericht Nr. 80
2016

Universität Bremen

TZI-Berichte

Herausgeber:
Technologie-Zentrum Informatik und Informationstechnik
Universität Bremen
Am Fallturm 1
28359 Bremen
Telefon: +49 421 218 94090

E-Mail: hq@tzi.de
http://www.tzi.de

ISSN 1613-3773

Towards Security Program Comprehension with Design by Contract and Slicing

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

***Abstract*—Over the last years, the field of software security has evolved. 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). We reconstruct parts of an application's security architecture by means of program slicing and specification inference utilizing knowledge on software frameworks and Security APIs. With the help of extended static checkers, we then automatically check whether applications use Security APIs in a way to satisfy their security requirements. Our proposed methodology can be seen as a first step towards more systematic security code audits.**

I. Introduction

Security code audits belong to the most difficult and tedious activities within a Security Development Lifecycle (SDL). Although automatic code review tools, such as Fortify SCA or IBM AppScan, let a security analyst identify low-level programming bugs, security code audits still remain important. Bug finding tools allow one to detect mostly isolated bugs rather than revealing relations between different objects involved in implementing a security requirement. On reviewing encryption functionality, for example, a security analyst needs to answer questions like "Which key is used for encryption?", "Which source is used for the symmetric key?" or "Which data have been encrypted with which algorithm and encryption mode?". In a different scenario, an analyst might be interested in understanding the access control policy implemented in a security-critical web application, a task that bug finders do not cover.

Despite the inherently manual nature of security code audits the question arises of whether this process can be better supported by tools. This allows one to better comprehend the *implemented* security architecture of an application and finally leads to more effective security reviews.

Since developing a general tool-supported approach to software security comprehension is too challenging due to the diverse nature of security mechanisms implemented in software, it is of importance to define a methodology for certain classes of applications. One reasonable restriction is to define such a process for applications that employ widely-used software frameworks, such as Java Enterprise Edition (JEE) [49], Spring [53] or the Android Framework [30]. Then an analyst can start code audits from calls of security-relevant APIs of the software framework. Security-relevant APIs can be encryption, authentication, and authorization APIs, but also include APIs that have indirect implications on the implemented security architecture, such as communication APIs. For example, when an HTTP API is employed, an analyst needs to know which kind of data is sent over the channel, whether these data are security-critical and whether TLS is used in case of sensitive data.

An important subtask of code reviews is to check whether security-relevant APIs are used correctly with respect to security. For example, a study has shown that many popular Android applications implemented SSL functionality wrongly, allowing middleperson attacks [25]. The study concluded that SSL libraries were 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 [21].

In this paper, we propose a methodology that replicates security code audits based on known software engineering techniques including program slicing [34], [39], Design by Contract (DBC) [46], [40], extended static checking [28], and annotation inference [24]. We start our analysis from security-critical API calls of software frameworks and automatically determine dependences by means of backward slicing. This step, for example, allows us to identify the origin of parameters of the API calls, e.g., determining the concrete keys and data of encryption API calls. Slicing also makes the step of extended static checking more tractable because code that is irrelevant for security is eliminated.

The use of DBC is twofold. First, it allows us to formulate concise preconditions for security-relevant APIs. An extended static checker can then verify whether these preconditions are satisfied on each API call, addressing the aforementioned problem of correct API usage. Second, DBC lets a security architect specify application-specific security requirements in form of postconditions, e.g., the access control policy. Again, an analyst can check whether these requirements are satisfied by means of an extended static checker.

Since annotating code with specifications has been shown to be tedious [28], it is important to provide a method that automatically inserts annotations into program code. For this purpose, we use the Daikon tool, which can infer likely DBC specifications by code instrumentation [24]. The preceding

slicing step helps the inference tool produce annotations that better focus on the implemented security mechanisms.

In our approach we decided to adopt state-of-the-art analysis tools rather than building tools from scratch because we aim to utilize mature base analyses. Also, other approaches including ComDroid [13], CryptoLint [21] or MalloDroid [25] in case of Android are specific research tools and do not aim at providing a unified and encompassing solution to the problem of security code audits. Our approach can be applied to different Security or security-relevant APIs as we address the general problem of tracing back dependences of Security API calls.

None of the proposed basic analysis tools, however, could be used out of the box. Rather, they had to be appropriately adjusted to fit our purpose. For example, as precise context-sensitive interprocedural slicing is computationally expensive, we had to use optimized, but more imprecise slicing options (e.g., neglecting heap dependences). Then a series of related other slices had to be added automatically to compensate for the missing precision—identifying the necessary supplemental slices for the specific analysis problem was one of the challenges to be solved. Also, we had to extend the Daikon tool to support exceptional specifications. This extension was finally integrated into the Daikon itself as this feature was long missing.

In summary, we believe that more systematic approaches to security code audits are 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 presents our current implementation of the approach. Section V describes experiments that we carried out with a proof-of-principle implementation of our approach. In Section VI, 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 VIII.

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 [46], [58], [7]. Invariants apply to the entry and exit of *all public* methods. For most mainstream programming languages, DBC extensions exist [7], [12], [41], 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. [32]. 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 [40]. 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 are 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 [28].

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 [57]. 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) [34]. 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 [39].

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

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: Security requirements of the clinical information system.

conclude this section with a discussion of several Security APIs in the light of the DBC principle.

A. Motivation

We discuss our technique for tool-automated security code audits with the help of a fictitious clinical information system, which is to be evaluated by internal quality assurance (QA) or an external evaluator (e.g., w.r.t. the Common Criteria [14]). Among other functionality, it should provide means for reading and writing electronic health records (EHRs). We further assume that it is a JEE-based client-server application [49].

Fig. 1 depicts some security requirements for this application as part of a hospital's security policy. `Req 3` requires that a clinician must play the role `Physician` to read or write the patient data of the EHR; for reading prescriptions, the role `Nurse` suffices (`Req 1`). There are additional access control requirements, 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. As the hospital belongs to a healthcare provider who runs several hospitals, storage capacity as a cloud service is offered to the 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 software QA now is to evaluate the software w.r.t. the security requirements presented in Fig 1. Typical questions to be answered by an analyst include "Does the `writePrescriptions()` method satisfy `Req 2`?" and "Does the `sendDiagnosis()` method make sure that patient data are signed and encrypted (`Req 5`), and if so, are secure encryption algorithms used?". It is often a laborious task for an evaluator 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 code analyzers. Specifically, a tool would be desirable 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 [36]. A tool that helps them focus on the implementation of security requirements would be very valuable for them.

```
1  @Resource EJBContext ctx;
2  Cipher mCipher;
3  Signature mSignature;
4
5  public String readPrescriptions(String ehrID, String userID){
6      Clinician clinician = getClinician(userID);
7      EHR eHR= getEHR(ehrID);
8      if !((ctx.isCallerInRole("Physician") || ctx.isCallerInRole("Nurse")) && eHR.getWard() == clinician.getWard())
9          throw new SecurityException("No sufficient access rights.");
10     return eHR.getPrescriptions();
11 }
12
13 public void writePrescriptions(String ehrID, String userID, String pres){
14     Clinician clinician = getClinician(userID);
15     EHR eHR= getEHR(ehrID);
16     if !((ctx.isCallerInRole("Physician") || ctx.isCallerInRole("Nurse")) && eHR.getWard() == clinician.getWard())
17         throw new SecurityException("No sufficient access rights.");
18     eHR.setPrescriptions(pres);
19 }
20
21 public String readPatientData(String userID, String ehrID){
22     Clinician clinician = getClinician(userID);
23     EHR eHR= getEHR(ehrID);
24     if !(ctx.isCallerInRole("Physician")  && eHR.getWard() == clinician.getWard())
25         throw new SecurityException("No sufficient access rights.");
26     String patientData =
27         ehr.getAdministrativeData() + ehr.getPrescriptions() + ehr.getDiagnosis();
28     return patientData;
29 }
30
31 public void sendDiagnosis(String userID,  String ehrID, Key key){
32     EHR eHR = getEHR(ehrID);
33     Clinician clinician = getClinician(userID);
34     if !(ctx.isCallerInRole("Physician") && eHR.getWard() == clinician.getWard())
35         throw new SecurityException("No sufficient access rights.");
36     byte[] signedData = signData(eHR.getDiagnosis().getBytes(), clinician.getKeyName());
37     byte[] encryptedSignedDiagnosis=encryptData(signedData, key);
38     String externalPractice = getExternalPractice();
39     String mailAddress = getMailAddress(externalPractice);
40     sendPatientData(encryptedSignedDiagnosis,mailAddress);
41 }
42
43 byte[] signData(byte[] data, String ksName, String keyAlias){
44     mSignature = Signature.getInstance("SHA256withDSA", "SUN");
45     KeyStore ks = KeyStore.getInstance("Hospital");
46     FileInputStream ksfis = new FileInputStream(ksName);
47     BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
48     ks.load(ksbufin);
49     PrivateKey priv = (PrivateKey) ks.getKey(keyAlias, null);
50     mSignature.initSign(priv);
51     mSignature.update(data);
52     return mSignature.sign();
53 }
54
55 byte[] encryptData(byte[] data, Key key){
56     mCipher = Cipher.getInstance("AES");
57     mCipher.init(Cipher.ENCRYPT_MODE, key);
58     return mCipher.doFinal(data);
59 }
60
61 void conncectToHealthcareProviderCloud(URL url) throws IOException, GeneralSecurityException {
62     TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
63     tmf.init(getKeystore());
64     SSLContext context = SSLContext.getInstance("TLS");
65     context.init(null, tmf.getTrustManagers(), null);
66     HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();
67     urlConnection.setSSLSocketFactory(context.getSocketFactory());
68     urlConnection.setHostnameVerifier(new AllowAllHostnameVerifier());
69     urlConnection.connect();
70 }
```

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(\result);
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.

B. The Analysis Approach in Detail

Subsequently, we describe the core concepts of our approach to tool-assisted security audits in more detail.

1) DBC-Based Annotations: We propose to utilize the advantages of DBC-based analysis because we consider the application of Security APIs a contract between the client (the application) and the callee (the security library). 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. The postconditions must then 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 using them in a way to guarantee its security requirements.

The aforementioned steps 2) and 3) can naturally be handled by DBC and related tools. For example, the security requirement "Data about a patient's prescriptions may only be written 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 returns 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 30-31). It states that `encryptData()` ensures that `data` is encrypted with the symmetric key `key` and the algorithm `AES/CBC/PKCS5Padding`.

2) Slicing: We have mentioned 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 [28]. Furthermore, verification conditions that must be internally processed by the prover can become extremely large and hence cannot be proven [28]. Lloyd and Jürjens, for example, carried out a case study, a biometric authentication system, and pointed out that it was difficult to check JML-annotated methods because they were too large ([43], 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 helps, which allows one 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,

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.
        If the annotations are automatically generated, possibly revise them.

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.

many unrelated code annotations will be generated. Slicing reduces the number of these annotations.

The underlined code in Fig. 2 represents the slice that results when using the `doFinal(data)` call as the slicing criterion (see line 58). In particular, the lines 38 to 40 from Fig. 2 do not belong to the slice. Since slicig respects data and control dependences, we can track which data are encrypted by the `doFinal(data)` call and find out that they are the signed diagnosis data (see line 36, Fig. 2).

Furthermore, the example shows the usage of interprocedural backward slicing; the backward slice starts within the `encryptData()` method and runs through the `signData()` and `sendDiagnosis()` methods. Interprocedural backward slicing gives a security analyst a more comprehensive view on the security-relevant code, which is distributed over several methods. In particular, a security analyst can understand the relationships between the different Java objects (and their configurations) that are involved in encrypting the `data` array, e.g., the `Cipher` and the corresponding `Key` object as well as the `Signature`, its `PrivateKey`, and the `KeyStore` object.

Fig. 3 then shows the result of slicing the `sendDiagnosis()` method (see line 21-28) w.r.t. the `doFinal(data)` and `isCallerInRole(``Physician'')` calls. Hence, combined slices are also possible.

Table I shows typical APIs calls that can be used as slicing criteria and let an analyst construct specific security views for analysis tasks. Starting from these calls other relevant statements are automatically added to the slice including calls of other API methods of the surrounding API class, e.g., `update()` or `init()` in case of the `doFinal()` API method. Furthermore, new analysis problems can then be defined by using other API method calls as slicing criteria, i.e., the approach can be generalized.

API	API class	Seed methods
JEE authorization API	EJBContext	isCallerInRole()
Java encryption	Cipher	doFinal()
Java signature	Signature	sign()
Java keystore	KeyStore	getKey()
Https-based communication	HttpsURLConnection	connect()

TABLE I: API calls to be used as slicing seeds.

3) Specification Inference: One open point of our approach is the burden of code annotation [28]. We address this problem by employing a dynamic approach, which, for example, is implemented in the Daikon tool [24]. 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.

Due to the nature of the annotations to be inferred in our case, we must consider the following aspects:

- We must capture data at exceptional program exit points to generate exceptional and non-exceptional cases to produce heavyweight JML specifications.
- We must support `pure` method calls within the JML specifications. Since our overall approach is based on properties of Security(-relevant) APIs, we expect an increased use of pure method calls within specifications, such as `isCallerInRole()` or `getKey()`.

- We must support conjunctive invariants as shown in the annotations from Fig. 3.

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.

4) Summary of the 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 [34]. 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. This slicing step is crucial because it enables us to carry out extended static checking as well as annotation inference. Otherwise, these tasks would be prohibitively expensive w.r.t. space and time.

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. If Daikon is used, the annotations must be manually revised because Daikon only infers likely rather than precise specifications.

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 need to provide the code of the API's methods, but can use the `//@ assume` statement made available by extended static checkers. This statement allows the checker to assume conditions without proving them.

For example, we do not have to prove the correctness of the APIs provided by Java's `Cipher` class and hence only need to provide implementation stubs, which use `//@ assume` statements. Fig. 5 shows this concept. The annotation for the `doFinal()` method states that `getOutput()` returns the result of encrypting `data`; the annotation of `init()` says the parameter `key` is used with this `Cipher` instance. Methods such as `getOutput()` and `getKey()` are specification-only JML methods, which can also be used by clients of the `doFinal()` method to make specification more readable.

In a prerequisite step ("Step 0"), DBC annotations for the Security APIs including the implementation stubs must be provided. This task must be done manually and hence requires some effort. However, it needs to be carried out less often

```
 1 public class Cipher{
 2
 3   /*@
 4       requires algorithm.equals("AES/CBS/PKCS5Padding");
 5       ensures \result.getAlgorithm().equals(algorithm)
            && algorithm.equals("AES/CBS/PKCS5Padding");
 6     @*/
 7   public static Cipher getInstance(String algorithm}{
 8     Cipher res = new Cipher();
 9     //@ assume res.getAlgorithm().equals(algorithm);
10     return res;
11   }
12
13   //@ ensures this.getKey().equals(key);
14   public void init(int mode, Key key){
15     //@ assume this.getKey().equals(key);
16   }
17
18   /*@
19       ensures this.getInput().equals(data) && this.
            getOutput().equals(\result);
20     @*/
21   public byte doFinal(byte[] data){
22     byte[] res = new byte[42];
23     //@ assume this.getOutput().equals(res);
24     return res;
25   }
```

Fig. 5: Exemplary implementation stubs for the `Cipher` class.

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: An extended static checker can use specifications of the Security APIs to verify that an application satisfies its security requirements. The postcondition of the `encryptData()` method (see Fig. 3, line 30) can be proven by using the annotations of the `Cipher` class depicted in Fig. 5. In particular, this implies that `Cipher.init()` and `Cipher.getInstance()` must have been called before with the appropriate parameters for the key and the encryption algorithm.

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 SDL follow such practices, e.g., SAP, who built this code into a mobile communication library such that several (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 34)

```
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 precondition

```
requires \result.getAlgorithm().equals("AES/CBC/
    PKCS5Padding");
```

then an extended static checker automatically can identify the aforementioned issue.

As a further observation, preconditions tend to correlate to rules that guarantee the secure usage of an API method. For example, the `getInstance()` method has a precondition stating that the CBC mode should be used with AES encryption. The same remark applies to the 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).

Postconditions, such as exceptional 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.

In summary, our approach allows one to encode security knowledge on APIs in form of JML annotations. Through the automated extraction of security-relevant parts from the code, the process of security reviews can be better automated. This leads to more comprehensive and hence more effective code reviews. Consequently, common situations can be avoided where one security aspect has been considered deeply, whereas others have been neglected. For example, SAP's Android apps showed quite good cryptographic implementation (reasonable key management, usage of secure algorithms, secure random number generation), but weaknesses in SSL/TLS encryption. Most importantly, our approach replicates and automates the procedure security analysts follow when auditing code. They start their analysis from security-relevant API calls, such as `Cipher.doFinal()` or `UrlConnection.connect()`. Then they trace back parameters, return values and further related objects to their origins while carrying out manual security analyses (quasi in their mind based on experience).

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()` [51] 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 [55] and ESAPI [50].

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 [37]. 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 [52]. 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], [22], [31], [25], [44]; in some cases, an attacker could even execute system permissions [31]. 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], [22]. A typical security rule for 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, showed behavior that violated 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 the Android Framework. This underlines the need of tools that encode security knowledge on software frameworks.

To implement a slicing tool for Android applications (available in Java source or byte code), additional edges must be inserted into the call graph. For example, edges must be added between `sendBroadcast()` calls and corresponding `onReceiver()` calls of broadcast receiver components. On selecting API calls, such as `sendBroadcast()`, `startActivity()` or `startService()`, as further slicing criteria, parts of the security architecture of the app can be reconstructed and used for program understanding tasks.

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 VI.

Algorithm 1: Adding automatically additional slices.

Input : A program *p* to be sliced, a *slicer* and the slicing criterion *crit*.
Output: A combination of slices *s* w.r.t. criterion *crit*

```
1 Initialize slicer with the options NO_HEAP and
  NO_EXCEPTIONAL_EDGES;
2 Build SDG sdg;
3 s = Slice p w.r.t. criterion crit;
4 for each method call statement mc in s do
5 |   for each parameter call statement pcall of mc do
6 |   |   if pcall not in slice s then
7 |   |   |   s'= Slice p w.r.t. criterion pcall;
8 |   |   |   s = Merge s and s';
9 Return s;
```

IV. Implementation Aspects

We now describe the proof-of-concept implementation of our analysis approach based on WALA and Daikon. The focus of the description lies on extensions of the current tools to better fit our purposes, e.g., by adding new features (in case of Daikon) or building a new tool based on the analysis infrastructure (in case of WALA).

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

A. Slicing with WALA

We used the slicer provided by the byte code analysis framework WALA [19] as the basis for our implementation. We also implemented 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 (see I) as slicing criteria. Our tool then internally searches the call graph part of the SDG via a depth-first search to find all seed statements for slicing (see also Section III-B2) and finally calls the WALA slicer.

We had to perform some optimizations to make slicing feasible given that interprocedural slicing in general is of quartic complexity [35]. In particular, we excluded about 380 classes/packages from the analysis space, e.g., Swing classes or basic Java classes, such as `java.lang.String`. Otherwise, the slicer would descend into the Java API implementation, which is prohibitively expensive.

We enabled WALA's `NO_HEAP` option, i.e., data flows through the heap are not followed. This analysis is in general too costly as pointed out by Sridharan et al. [54]. Since the slicing algorithms must also descend into library/API methods, dependences may be lost on using `NO_HEAP` then. This leads to false negatives, which are problematic because security-critical code may not appear in the slice.

Two situations may occur when dependencies cannot be correctly traced back through the body of an API method: (1) a dependency to another API call of the same library class is lost (e.g., Fig. 2, lines 57-58, lines 67-69); (2) a dependency that influences one of the API call's actual parameters is lost (e.g., Fig. 2, lines 49-50, lines 64 and 67). The former dependency may stem from an internally used *common* member variable of the API method's surrounding class. The latter dependency may originate from cutting off data flows going through the method to the actual parameters. Both cases can be resolved similarly by automatically adding further slices. For example, the latter case can be addressed by the algorithm depicted in Fig. 1. All actual parameter statements are used as additional slicing seeds if they are not contained in the current slice. The second case is slightly more complicated. The rough idea is to add API method calls of the surrounding class as slicing criteria if they are not contained in the slice yet and if they occur earlier in the call graph or are located in the same method.

Another optimization that we used is not to consider exceptional edges in the control flow. Otherwise all method call statements that could potentially throw an exception would be added to the slice which would lead to many false positives.

B. 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 analyzed software. The process of test-suite generation can be automated by (1) determining the entry point methods of backward slices and (2) generating test cases for these methods (e.g., looping through their parameters and the member variables of the surrounding class).

To make the inference approach work for our purposes, we extended Daikon. As Daikon did not support heavyweight JML specifications including exceptional cases (see Fig. 3), we adjusted Daikon's bytecode instrumenter Chicory [24]. Chicory inserts hooks into the bytecode to print information about program runs into a trace file that is later analyzed by the Daikon engine. We extended Chicory to also protocol exceptional method exits including the type of thrown exception. Our main contribution here lies in identifying code locations where exceptions are thrown and writing out these abnormal program exits into the trace file. Two situations must be considered here: (1) a throw instruction is directly encountered and (2) an exception is propagated up the call stack. Case (1) is straightforward, whereas case (2) is more difficult. We solved this problem as follows: insert try-catch statements that cover the whole method, add code to protocol an abnormal exit and finally rethrow the exception. With some further optimizations, this approach works well and has been finally integrated into Daikon's current version.

Furthermore, Daikon was configured to produce specifications that contain pure method calls, such as `isCallerInRole()` or `Cipher.getKey()`. To utilize this feature, we had to tell Daikon which methods are pure such that they are considered while inferring JML annotations. Daikon had some restrictions concerning this feature, e.g., pure method calls must either have member variables of the surrounding class as parameters or an empty parameter list. We adjusted the software under analysis accordingly.

Another requirement was to support conjunctions in specifications (see Fig. 3). Daikon produces implications, which can equivalently be translated with negations to conjunctions. For producing meaningful implications, we used Daikon's splitter feature. This allows one to define certain "conditional program points", which Daikon then uses to infer implications/conjunctions.

Furthermore, Daikon had to be adjusted to print the heavyweight specifications into Java classes. Fig. 6 depicts annotations inferred and thereafter automatically inserted JML annotations. Lines 15 and 16 represent the fact that a security exception is thrown in the exceptional case—both lines together are then translated by our annotator into the clause

```
signals_only SecurityException;
```

A further point to be regarded is that Daikon produces many specifications that are not relevant in our context (e.g., see lines 3 to 6 in Fig. 6). We had to remove these constraints manually to obtain more readable specifications. Furthermore, Daikon was able to conclude conditional statements (implications) within a heavyweight JML specification. This, for example, can be seen in line 7 in Fig. 6. Finally, Fig. 7 depicts the revised specifications.

V. Case Studies

JEE-based Web Application: We have implemented a test 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. In particular, we generated code variants. The purpose of this mutants-based testing approach [38] is to check for false positives and negatives of the slicing process. 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:

- 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)—this case tests for false negatives w.r.t. interprocedural slicing,
- adding pointer assignment statements that influence the slicing criterion, e.g., line 2 in the following code:

```
1 PrivateKey priv = (PrivateKey) ks.getKey(keyAlias,
      null);
2 PrivateKey priv1 = priv;
3 mSignature.initSign(priv1);
4 mSignature.update(data)
5 return mSignature.sign();
```

- insertion of additional if-statements that influence the slicing criterion,
- 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.

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

In total, we generated 30 test cases for evaluating the slicing step. Of all test cases, 12 false positives occurred, whereas two false negatives were found (0.6 precision, 0.93 recall). The false positives occurred when unrelated objects are accessed; access to unrelated variables of primitive Java types did not contribute to this rate. One reason for these false positives is the fact that WALA also follows exceptional control flow edges. In many cases, this is the desired behavior, but it led to the false positives.

We found false negatives in situations similar to the code example in the enumeration above. In that specific case, WALA did not include the statement `priv1 = priv;` because WALA does copy propagation optimizations during SSA generation for simple assignments. In a more complex case, WALA missed statements due to the `NO_HEAP` option, i.e., data dependences are not followed through the heap. We finally addressed this problem by adding two further slices with `initSign()` and `update()` calls, respectively, as seed statements. In Section VI, we discuss a different kind of false negative. This situation is more fundamental as it is inherent in our general approach and not in the slicing implementation.

After configuring the slicer as described before, we perform program understanding tasks. For example, a security analyst can follow the slice from the `sign()` statement back to the code location where the private key is obtained. In the JEE application, the private key has been retrieved from a Java keystore (see Fig. 2, line 49):

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

One can see that the keystore is not secured by a password (`null` parameter), although a sensitive private key is stored there. To automatically detect such situations, we specify the following JML precondition for the `getKey()` API:

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

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

A. 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. The Android Framework makes available authorization APIs, which mobile applications can use to enforce more fine-grained access control policies [23]. 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.

Here, we follow a case study, which has been discussed elsewhere [47], but extend it in several ways. First, we can analyze different Android versions and have carried out analyses of Android 1.5 until Android 5.0—the approach followed in [47] only considered Android 4.0.3. Furthermore, we are able to analyze all Java-based system services, which are about 35 per Android version[2]. The source code of the system services is (mostly) automatically collected from the code base. For each Android version, we analyzed about 120,000 lines of code in total (considering all Android system services).

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 II and Table III one can conclude

[2]Only in a few cases, the code could not be sliced because WALA could not generate a call graph, e.g., see the entries in Table III denoted by "−−".

```
1  /*@
2      @ public normal_behavior // Generated by Daikon
3      @ requires this.mHasFeature == refreshing;
4      @ requires this != null;
5      @ requires this.mContext != null;
6      @ ensures this.mHasFeature == \old(this.mHasFeature);
7      @ ensures (this.mHasFeature == false)  ==>  (this.mContext.checkCallingPermission(MANAGE_DEVICE_ADMINS) ==
           PERMISSION_DENIED || this.mContext.checkCallingPermission(MANAGE_DEVICE_ADMINS) == PERMISSION_GRANTED);
8      @ ... // further invariants
9      @ also
10     @ public exceptional_behavior // Generated by Daikon
11     @ requires this == \old(this);
12     @ requires this.mHasFeature == \old(this.mHasFeature);
13     @  ... // further invariants
14     @ requires PERMISSION_DENIED == this.mContext.checkCallingPermission(MANAGE_DEVICE_ADMINS);
15     @ requires Exception != null;
16     @ requires Exception.getClass().getName() == java.lang.SecurityException.class.getName();
17     @*/
18     public void setActiveAdmin(ComponentName adminReceiver, boolean refreshing, int userHandle) {
19         if (!mHasFeature) return;
20         mContext.enforceCallingOrSelfPermission(MANAGE_DEVICE_ADMINS, null);
21     }
```

Fig. 6: Sliced and with Daikon annotated source code of the `DevicePolicyManagerService`.

```
1  /*@ public normal_behavior
2    @ requires checkCallingOrSelfPermission(MANAGE_DEVICE_ADMINS)== PERMISSION_GRANTED;
3    @ ensures (this.mHasFeature == false)  ==>
4    @ (this.mContext.checkCallingPermission(MANAGE_DEVICE_ADMINS) == PERMISSION_DENIED ||this.mContext.
        checkCallingPermission(MANAGE_DEVICE_ADMINS) == PERMISSION_GRANTED);
5    @ also
6    @ public exceptional_behavior
7    @ requires checkCallingOrSelfPermission(MANAGE_DEVICE_ADMINS) == PERMISSION_DENIED;
8    @ signals_only SecurityException;
9    @ */
10 public void setActiveAdmin(ComponentName adminReceiver, boolean refreshing, int userHandle) {
11         if (!mHasFeature) return;
12         mContext.enforceCallingOrSelfPermission(MANAGE_DEVICE_ADMINS, null);
13 }
```

Fig. 7: Slightly revised specification from Fig. 6.

for selected system services and Android versions that slicing reduces the code size to less than 10% when considering the access control policy. Since we used WALA's `NO_HEAP` option as handling a complete heap model appeared to be too costly, we also looked for false negatives w.r.t. slicing. In the `PackageManagerService` of Android 4.0.3 we compared the results obtained by slicing with information that we gained by a manual code review of the original code and identified no false negatives. This behavior can be explained because the permission enforcement checks and possible influencing statements did not involve any access to variables stored on the heap.

Moreover, Table IV 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. The API `revokePermision()`, for example, was only a *local* method before Android 4.1.2. Thereafter, it was 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 undocumented mechanisms. Daikon helped us infer specifications as given in Fig. 7—the sliced code was taken from the `DevicePolicyManagerService`, a service used to implement MDM apps. Interestingly, Daikon concluded that if `mHasFeature == false`, then no permission is required (see line 3 and 4), i.e., it was able to infer conditional access control checks, which are mostly undocumented. Other approaches that construct a permission map for Android, such as PScout [5], do not cover hidden features, which are only available to the system or dedicated system apps.

Also, this case study demonstrated that the slicing step is essential in enabling annotation inference. Without slicing, all parameters as well as all member variables of the surrounding class must have been considered and initialized by different

values by the test program. Due to slicing, we limited search space only to parameters and member variables that are used within the method. Without slicing, the trace file was simply too large to be analyzed by the Daikon engine, e.g., in case of the `DevicePolicyManagerService`.

The focus of this case study lies 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.

VI. 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 [20] and Z3 [17].

Fig. 8 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) [19] 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.

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

False Negatives: We have already mentioned that WALA's `NO_HEAP` option led to false negatives in some situations. One possibility to mitigate this problem is to add further slices. For example, by adding the criterion

```
mSignature.initSign(priv);
```

one can also follow the signing key `priv` if this dependency is missed by the slicer.

Furthermore, some false negatives are inherent in the analysis approach rather than in the tool support. For example, consider the method `sendDiagnosis()` in Fig. 2. If we use the cryptographic and programmatic access control API calls as slicing criteria, then the statement

```
sendPatientData(
          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. One possible approach to address this problem is to add a **forward slice**, i.e., all the statements that depend on the slicing criterion. This can be easily implemented as WALA, for example, also contains this option. In our case, the slicing criteria again are the `doFinal()` and `sign()` calls, but now as seeds for forward slices. Then the forward slices contain all the statements that are influenced by the encrypted/signed data. Although the aforementioned statement will still not occur in the slice, an analyst will at least better understand where the encrypted data are used (and where not).

Similar remarks apply to `HttpsUrlConnection` objects (see Fig. 2). Performing a forward slicing step (in combination with backward slicing) w.r.t. the API call `connect` gives a broader picture about http(s) communications of the application. In general, forward slicing allows tracking the usage of all objects on which Security APIs have been applied; these objects are mostly security-critical and of interest for a security analyst.

To sum up, our technique does not necessarily detect all security-relevant code locations, and we are aware of the fact that related tools are neither sound nor complete. 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 will never 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 [45]. 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

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 II: 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 III: 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 IV: Permissions of the PackageManagerService in selected Android versions.

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 [27]. 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

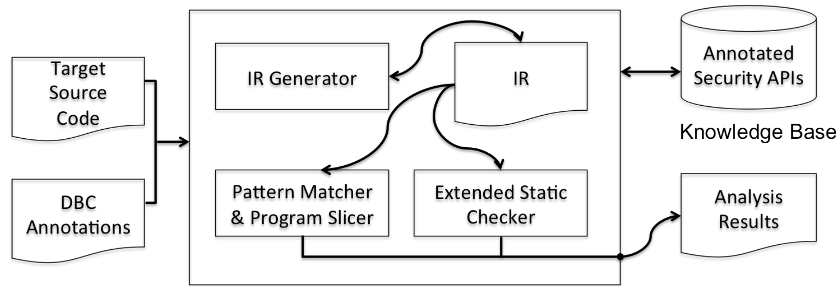

Fig. 8: Analysis infrastructure.

project. Equipped with a knowledge base of specifications (see Fig. 8), 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.

VII. 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 [42], [4], [12], [10], [26], [21]. Important research prototypes from static analysis are e.g. MOPS [10], Eau Claire [12], and LAPSE [42]. 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 [42]. Felmetsger et al. employ the Daikon tool [24] to dynamically infer security specifications for web applications. Thereafter, they use a model checker to detect application logic vulnerabilities violating the specifications [26].

Similarly to our approach, the CryptoLint tool uses program slicing [21]. 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. Furthermore, as CryptoLint works on Android bytecode, it cannot be used for Java software in general.

Some of the research prototypes evolved into commercial tools such as Fortify SCA [29] and Coverity Prevent [16]. 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 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 [33]. 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 [48]. 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 [18]. 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) [43] 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 [56]. Contrary to our proposed technique, all these approaches neither consider the relationship between DBC and Security APIs nor the aspect of security program comprehension.

VIII. 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) (2013), 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] Coverity: Coverity Prevent (2015), http://www.coverity.com

[17] 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)

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

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

[20] 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)

[21] 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)

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

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

[24] 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)

[25] 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)

[26] 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)

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

[28] 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)

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

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

[31] 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)

[32] 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)

[33] 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

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

[35] Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. SIGPLAN Not. 39(4), 229–243 (Apr 2004), http://doi.acm.org/10.1145/989393.989419

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

[37] 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
[38] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649 –678 (2011)
[39] Krinke, J.: Advanced Slicing of Sequential and Concurrent Programs. Ph.D. thesis, Universität Passau (2003)
[40] Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary Design of JML: A Behavioral Interface Specification Language for Java. SIGSOFT Softw. Eng. Notes 31(3), 1–38 (May 2006)
[41] Leino, K.R.M., Müller, P.: Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs (2009)
[42] Livshits, B., Lam, M.: Finding Security Vulnerabilities in Java Applications Using Static Analysis. In: Proc. of the 14th USENIX Security Symposium (Aug 2005)
[43] 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)
[44] 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)
[45] McGraw, G.: Software Security: Building Security In. Addison-Wesley (2006)
[46] Meyer, B.: From structured programming to object-oriented design: The road to Eiffel. Structured Programming (1), 19–39 (1989)
[47] 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
[48] 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)
[49] Oracle Inc.: The Java EE 5 Tutorial (2013), http://docs.oracle.com/javaee/5/tutorial/doc/bnbyk.html
[50] OWASP: OWASP Enterprise Security API (2012), https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API
[51] Pivotal, Inc.: Spring security 3.1.2 (2013), http://static.springsource.org/spring-security/site/index.html
[52] 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)
[53] springsource community: Documentation (2013), http://www.springsource.org/documentation
[54] 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)
[55] The Apache Software Foundation: Apache shiro 1.2.1 (2013), http://shiro.apache.org/
[56] Warnier, M.: Language Based Security for Java and JML. Ph.D. thesis, Radboud University, Nijmegen, Netherlands (2006)
[57] Weiser, M.: Program slicing. In: Proceedings of the International Conference on Software Engineering. pp. 439–449. IEEE Press, Piscataway, NJ, USA (1981)
[58] Zeller, A.: Why programs fail - a guide to systematic debugging. Elsevier (2006)