Vulvet: Vetting of Vulnerabilities in Android Apps to Thwart Exploitation

JYOTI GAJRANI, MEENAKSHI TRIPATHI, and VIJAY LAXMI, MNIT Jaipur, India

GAURAV SOMANI, Central University of Rajasthan, Ajmer, India

AKKA ZEMMARI, LaBRI, Bordeaux INP, University of Bordeaux, CNRS, France

MANOJ SINGH GAUR, Indian Institute of Technology Jammu, India

Data security and privacy of Android users is one of the challenging security problems addressed by the security research community. A major source of the security vulnerabilities in Android apps is attributed to bugs within source code, insecure APIs, and unvalidated code before performing sensitive operations. Specifically, the major class of app vulnerabilities is related to the categories such as inter-component communication (ICC), networking, web, cryptographic APIs, storage, and runtime-permission validation. A major portion of current contributions focus on identifying a smaller subset of vulnerabilities. In addition, these methods do not discuss how to remove detected vulnerabilities from the affected code.

In this work, we propose a novel vulnerability detection and patching framework, *Vulvet*, which employs static analysis approaches from different domains of program analysis for detection of a wide range of vulnerabilities in Android apps. We propose an additional light-weight technique, *FP-Validation*, to mitigate false positives in comparison to existing solutions owing to over-approximation. In addition to improved detection, *Vulvet* provides an automated patching of apps with safe code for each of the identified vulnerability using bytecode instrumentation. We implement *Vulvet* as an extension of Soot. To demonstrate the efficiency of our proposed framework, we analyzed 3,700 apps collected from various stores and benchmarks consisting of various weak implementations. Our results indicate that *Vulvet* is able to achieve vulnerability detection with 95.23% precision and 0.975 F-measure on benchmark apps; a significant improvement in comparison to recent works along with successful patching of identified vulnerabilities.

CCS Concepts: • **Security and privacy** → **Domain-specific security and privacy architectures**;

Additional Key Words and Phrases: Android, vulnerabilities, security, protection, static analysis

ACM Reference format:

Jyoti Gajrani, Meenakshi Tripathi, Vijay Laxmi, Gaurav Somani, Akka Zemmari, and Manoj Singh Gaur. 2020. *Vulvet: Vetting of Vulnerabilities in Android Apps to Thwart Exploitation*. *Digit. Threat.: Res. Pract.* 1, 2, Article 10 (May 2020), 25 pages. <https://doi.org/10.1145/3376121>

The work is partially supported by DST-CNRS project IFC/DST-CNRS/2015-01/332 at MNIT Jaipur.

Authors' addresses: J. Gajrani, M. Tripathi, and V. Laxmi, MNIT Jaipur, Rajasthan, India; emails: {2014rcp9542, mtripathi.cse, vlaxmi}@mnit.ac.in; G. Somani, Central University of Rajasthan, Ajmer, Rajasthan, India; email: gaurav@curaj.ac.in; A. Zemmari, LaBRI, Bordeaux INP, University of Bordeaux, CNRS, Bordeaux, France; email: zemmari@labri.fr; M. S. Gaur, Indian Institute of Technology Jammu, J&K, India; email: director@iitjammu.ac.in.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

2576-5337/2020/05-ART10 \$15.00

<https://doi.org/10.1145/3376121>

1 INTRODUCTION

Smartphone devices aggregate personal data related to our daily activities through the applications (apps). The data accessed by apps are very sensitive and private. In spite of development of a multitude of anti-virus and anti-malware analysis frameworks, protection of Android users' data continues to be a challenging problem. One of the major contributors to data security problems are vulnerabilities introduced by weak code implementation in the Android applications, which can be exploited for various mischievous activities. Multiple reports show that most of the developers seldom code from the perspective of security [1, 2]. Developers think more in terms of providing better functionality and rich GUI. Malware targets unintentional mistakes of app developers, which ultimately leads to end-user losses. The malicious activities include leaking sensitive data, privilege escalation, unauthorized access, hijack attacks, unintended invocation of components, phishing attacks, denial of service attacks, unauthorized read from and/or write to content providers, and spoofing attacks [3].

A large number of research contributions focus on vulnerability detection in web-based applications [4–8]. In comparison to the web applications, Android has other vulnerabilities owing to a weaker permission model (permission with *normal* protection level and users granting permissions without information on implications), inter-component communication (ICC) through Intents with no security validation of information passed through Intents, webview, broadcast receivers, and services [3]. The popular listing of publicly disclosed security vulnerabilities and exposures, CVE lists approximately 4,300 vulnerabilities related to the Android OS, firmware, system apps, and user apps [9]. The research community proposed solutions to identify subsets of these diverse sets of vulnerabilities. However, vulnerability identification lacks solutions for possible removal of these vulnerabilities. In this work, our main focus is on app vulnerabilities. Section 2 details the vulnerability aspects by discussing how insecure usage of the Android features leads to security exploits.

In this article, we propose and develop *Vulvet*, a vulnerability vetting, assessment, and patching framework. We propose to apply multiple static analysis approaches for detection of a wide range of vulnerabilities and propose automatic patch generation using bytecode instrumentation. Our automatic patch generation technique efficiently mitigates the vulnerabilities without affecting app functionality. The following are important contributions of the proposed technique:

- We present a novel multi-tier multi-pronged vulnerability analysis technique to reduce the false positive rate (FPR) and false negative rate (FNR).
- We identify and apply various static-analysis techniques suited for different vulnerabilities to lower the false negative rate.
- We augment control-flow analysis and Android-specific component validation approach to lower the false positive rate.
- We propose automated patch generation for various vulnerabilities by instrumenting fixes of identified vulnerabilities in the byte code.
- We implement *Vulvet* by leveraging the features of Soot [10] such as scanning intermediate code, callgraph analysis, data-flow analysis, parameter analysis, taint analysis, return value analysis, and API analysis.
- We ascertain the efficacy by evaluating *Vulvet* on a dataset composed of 3,700 apps from benchmark and other Android markets. Evaluation results reveal that 10.46% of evaluated apps are vulnerable to different kinds of exploits.

The rest of the article is organized as follows: Section 2 briefly presents motivation of this work and background of various vulnerabilities. Section 3 presents system design of *Vulvet*. We detail evaluation results in Section 4 and limitations of *Vulvet* in Section 5. Section 6 reviews related work. We conclude the work in Section 7.

2 BACKGROUND AND MOTIVATION

Each Android app has four types of components: *Activity* for graphical user interface; *Service*, which does not have any UI and executes in the background to provide uninterrupted service; *content provider* for data storage

and sharing; and *Broadcast Receiver*, which listens asynchronously to broadcast messages. A mandatory file in each app, *AndroidManifest.xml*, facilitates the configuration of app components. In Android, each user-installed app runs in a separate sandbox.

Each app can save data to either the internal storage or the external storage. The former is the app's own private storage, while the latter can be shared and/or used as a private region. The shared external region can be accessed only when the app has *EXTERNAL_STORAGE* permission. The private external region is the app's private external region (*/storage/sdcard0/Android/ data/package*), which does not require any permission for the app itself but requires *EXTERNAL_STORAGE* permissions for other apps to access this region.

Android as an OS facilitates communication among its various components and apps through a message passing mechanism known as *Intent*. There are two ways to select the target component in a communication:

- (1) A component may call target component by explicitly specifying the name of the target class using an *Explicit Intent*.
- (2) Instead of explicitly selecting target with its name, a general *action* is written using an *Implicit Intent*. The Android system selects only the target components that specified the corresponding *action* in their manifest file. The *actions* are specified for components in manifest file through *intent-filter* as shown in Listing 1 (lines 2–4).

```

1 <service android:name=".MyService" android:enabled="true" android:exported="false">
2   <intent-filter>
3     <action android:name="cs.service"/>
4   </intent-filter>
5 </service>

```

Listing 1. Declaration of intent-filter with custom *action* in manifest file.

2.1 ICC Vulnerabilities

ICC category includes vulnerabilities originating due to the weak validation of Intents, PendingIntents, Services, Broadcasts (Simple, Ordered, and Sticky Broadcast), and Path Permissions. These features may be attacked in two different ways: (1) Spoofing attacks [11] and (2) Hijacking attacks [12]. Spoofing attack facilitates unintended invocation and occurs when a component registers an intent-filter but does not validate the identity of the sender or data provided by the sender. A malicious app creates an implicit Intent with same *action* and thus invokes the component performing sensitive operations or exploits any software bug in the component handling Intent by passing invalid data. While registering an Intent filter to receive implicit Intents, the system makes the respective component exported by default, which further increases the risk as the component becomes publicly accessible.

```

1 <receiver android:name="LowMemRevr" android:exported="true">
2   <intent-filter>
3     <action android:name="android.intent.action.DEVICE_STORAGE_LOW"/>
4   </intent-filter>
5 </receiver>

```

Listing 2. Declaration of intent-filter with system action restricting the invocation through implicit intent.

Listing 2 (component declaration in manifest) and Listing 3 (component's class file) show an example that is exploitable to Spoofing attack. The app registers broadcast receiver for system event *DEVICE_STORAGE_LOW*, which deletes files to avoid low storage issue. Although, this component cannot be invoked with implicit Intent, as system Intents cannot be sent by the user apps. It can, however, be invoked by malicious apps with explicit Intent, provided the broadcast receiver does not validate the *action* in the received Intent before performing sensitive operation.

```

1 @Override
2 public void onReceive(Context context, Intent intent) {
3 //Code Snippet of deleting file
4 boolean delete = intent.getBooleanExtra("Delete", false);
5 String sensitivefile = intent.getStringExtra("file");
6 if (delete) {
7     File dir = getFilesDir();
8     File file = new File(dir, sensitivefile);
9     if (file.exists())
10        file.delete();
11 }
12 }

```

Listing 3. Component vulnerable to spoofing attack.

A safe handling is to validate the *action* of incoming Intent as shown in Listing 4 (line 3), or to prevent the component invocation by setting *android:exported="false"* in the *AndroidManifest.xml* file, which limits the use of this component to the components of the same app only. The developers sometimes export the components for reuse in their other apps. In this case, the exported components can be protected using permissions as shown in Listing 5. A better way to limit the access of components to the apps by the same developer or by system apps is setting the value of *protectionLevel* attribute in permission tag with the value *signatureOrSystem*. In Android, each developer has a unique signature (a digital code) that identifies him/her on store.

```

1 @Override
2 public void onReceive(Context context, Intent intent) {
3 if (intent.getAction() != null && intent.getAction().equals("android.intent.action.
    DEVICE_STORAGE_LOW")) {
4 //Code Snippet of deleting file
5 boolean delete = intent.getBooleanExtra("Delete", false);
6 String sensitivefile = intent.getStringExtra("file");
7 if (delete) {
8     File dir = getFilesDir();
9     File file = new File(dir, sensitivefile);
10    if (file.exists())
11        file.delete();
12 }
13 }
14 }

```

Listing 4. Validation of *action* restricts the invocation through explicit intent. {*Vulvet* Methodology: Control-Flow Analysis}

```

1 <permission android:name="safepermission" android:protectionLevel="SignatureOrSystem"/>
2 <receiver android:name="LowMemRevr" android:exported="true" android:permission="
    safepermission">
3 <intent-filter>
4 <action android:name="android.intent.action.DEVICE_STORAGE_LOW"/>
5 </intent-filter>
6 </receiver>

```

Listing 5. Broadcast receiver protected with permission having protectionLevel SignatureOrSystem. {*Vulvet* Methodology: Tag & Attribute Analysis}.

Figure 1 illustrates hijacking attacks where a malicious app hijacks implicit Intent generated by benign app and steals information (image in this example) embedded in the data passed as parameter in Intent call. A major success factor for these attacks is the fact that the same Intent filter can be registered by the malicious apps. One of the possible solutions is to pass sensitive information through explicit Intents only or to validate data exchange through Intents by Android itself.

Unlike normal broadcasts, which are received by the intended receivers and get terminated thereafter, ordered broadcasts can be forwarded in a chain. A receiver of ordered broadcast may update the received information and then re-broadcast. A malicious app may set its priority high for receiving ordered broadcast before benign app,

Fig. 1. Use of implicit Intent leading to Intent hijacking by malicious app. {Vulvet Methodology: Data-Flow Analysis}.

change its contents, and then rebroadcast it to benign app. A possible solution is to restrict *sendOrderedBroadcast* API in any class that is inherited from *LocalBroadcastManager* class (restricts broadcasts to be sent within the app itself) or *sendOrderedBroadcast* with permission, which restricts the broadcasts to those apps that have the specified permission.

We could find numerous examples of real-world Play Store apps exhibiting ICC vulnerabilities. The authors in Reference [13] manually analyzed seven open-source mobile apps and found that *iFixit*¹ used implicit Intent for sensitive communication and has exported components. *Twicca* app's (in versions 0.7.0 through 0.9.30) activity can be launched by other apps resulting in privilege escalation and access to the SD card or network. These apps can upload images or movies stored on the SD card to a social networking service with the Twitter account of *Twicca*'s user [14]. The *AirDroid* application by SAND STUDIO (up to versions 1.1.0) mishandled implicit Intents, which allowed attackers to gather sensitive information through crafted malicious applications [15]. To validate the found vulnerability, we constructed successful proof of concept attack on *AirDroid*. The app was also vulnerable due to its exported components. We developed a malicious app that invokes exported broadcast receivers and activities of this target app. We send Intent to "*SmsSendedorReceivedReceiver*" receiver along with action "*ad.sms.sended*" as parameter, which executed this receiver's code of sending SMS. We could call other activities of the app insecure.

2.2 Storage Vulnerabilities

Android apps use content provider, internal storage, or external storage for storing data on local device. Android provides distinct predefined APIs for reading from and writing to these storage devices. Read operations may read data injected (through overwriting genuine data) by malicious apps (dirty read) and writing operations may write sensitive data that can be read and thereafter leaked by a malicious app. Possible solutions may not store the sensitive data on external storage or encrypt the data before storage and save the hash of last written data. This hash must be validated before reading operation. Writing user input files to internal storage without proper sanitization is vulnerable to directory traversal attacks [16]. Even allowing internal storage data to be downloaded on external storage makes the app vulnerable, as the data may be sensitive and thus can be leaked. Non-parametrized SQL queries on content providers are vulnerable to SQL injection attacks [17]. We could find many real-world apps that have such vulnerabilities. *CamiApp* (*jp.co.kokuyost.CamiApp*) is an app for maintaining digital notebooks and its version 1.21.1 and other older versions had vulnerable content providers that allowed attackers to bypass access restrictions and read database information through a malicious app [18].

¹<https://www.ifixit.com/Apps>.

The content provider in the *MovatwiTouch* app (a Twitter client app) before version 1.793 did not adequately restrict access to authorization information. This allowed attackers to hijack Twitter accounts of the users of *MovatwiTouch* app. Another app, *mixi* (*jp.mixi*) app (a Japanese social networking app) prior to version 4.3.0 used to store comments of friends on an SD card that allowed apps with external storage permission to read these comments.

2.3 Web Vulnerabilities

Allowing web pages and resources (JavaScript, CSS files, etc.) to load and execute in webview without validation of server, ignoring SSL errors during loading of incorrect SSL certificates, execution of JavaScript methods accessing sensitive information without permission, and adding interface to JavaScript for sensitive classes are a few vulnerabilities that can be exploited with MITM (man-in-the-middle) attack [19]. Webview vulnerabilities allow hackers to install malicious software, send SMSs, perform cross site scripting attacks, and so on [20]. There are numerous methods for setting various properties related to webview, such as *setAllowFileAccess*, *setJavaScriptEnabled*, *setAllowFileAccessFromFileURLs*, *addJavaScriptInterface*, and so on. Therefore, each feature of webview and its security implications must be well understood by developers before implementation. To ensure such vulnerabilities are not part of code, *Vulvet* analyzes each of the features to check if it is securely coded or not. This also becomes important for Android version 4.1 and earlier, in which default value is *TRUE* for these methods.

```

1 public class MyWVClient extends WebViewClient {
2 @Override
3 public WebResourceResponse shouldInterceptRequest(WebView wv, WebResourceRequest req) {
4     return null;
5 }
6 }

```

Listing 6. Returning null always may load malicious resources. {*Vulvet* Methodology: Return Value Analysis}.

Listing 6 shows an example of another webview vulnerability. The method *shouldInterceptRequest* is called while loading a resource in webview. Returning *null* without any checks allows any resource to load in the webview and thus, increases the probability of attack.

2.4 Cryptographic Vulnerabilities

Android apps use encryption for protecting data stored on device or communicated out of device. Compromises in encryption such as use of block cipher algorithms in *ECB* mode, static parameters, non-random initialization vector (IV), password creation using a constant salt, repeating same key over multiple sessions, and storage of encryption key on device are a few examples of cryptographic vulnerabilities. *Oral-B²* app (version 5.0.0) employs static Twitter keys, Twitter secret, and stored device values for AES encryption [21]. The reason behind use of insecure primitives is that most of the app developers are not aware of security primitives and their weaknesses.

2.5 Runtime-permission Vulnerabilities

Many times app developers intentionally export components for usability and perform permission checks at runtime using predefined methods such as *checkPermission* and *enforcePermission* to ensure that caller component has requisite permissions. However, improper validations using these methods will make the component vulnerable. As shown in Figure 2, vulnerability lies in improper usage of Binder API methods—namely, *getCallingPid()* and *getCallingUid()*, which are used to check the unique process identifier (PID) and unique user identifier (UID) of the calling component. However, the call of these APIs from the main UI thread will return the UID and PID of current process (if any such process is running in the background) and not the calling process [3]. This helps malicious attackers to construct an attack using this feature as outlined in the following paragraph.

²<https://goo.gl/W5qJsC>.

Fig. 2. Vulnerability due to improper usage of Binder API methods. {Vulvet Methodology: Data-Flow Analysis along with Control-Flow Analysis}.

MainActivity of benign app starts *MyService*, which enforces the caller to have “benign.permission” (Step 1). As the benign app has the permission, it is capable of executing the sensitive operation. The service, however, does not stop after performing operation in the app code. Afterwards, malicious app deliberately invokes *MyService*, which is not stopped (Step 2). The code in *enforcePermission()* method of main UI thread again validates the permission. However, as mentioned above, the binder APIs return UID and PID of benign app (current process) and not the calling process (malicious app), as permission enforcement code is a part of main UI thread. In this manner, the malicious app is able to perform a privilege escalation activity. Secure solutions call these methods only when request comes through binder (in a service interface method) or protects such components using permissions in *AndroidManifest.xml* file and lets Android validate the caller’s permission access.

2.6 Networking Vulnerabilities

Network communication between apps and remote servers via open ports and/or transfer of data with a weak or no encryption is vulnerable to MITM attacks. Many popular Play Store apps such as *Store and Share* (V2.0.18) (*sg.com.singnet.mystorage.android*) [22] and *iStunt2* (V1.1.2) (*com.miniclip.istunt2*) [23] with more than 10M downloads were found to have SSL vulnerabilities. Checkmarx³ showed analysis of *Tinder*,⁴ which is a location-based social networking app and reported two important vulnerabilities related to the insecure HTTP connections and encryption, and a predictable HTTPS response size. Owing to these vulnerabilities, the authors showed that an attacker on the same Wi-Fi network could reconstruct the victim’s session, access victim’s photos on *Tinder*, and add other photos in the victim’s photo-stream. The authors show this exploit using their “*TinderDrift*” app [24].

These examples demand an integrated and lightweight approach that can detect and patch the insecure codes in Android apps. A static analysis approach with high recall and precision applicable for app’s Dalvik bytecode will, therefore, be valuable. Section 3 details architecture and approach of *Vulvet*.

3 VULVET: ARCHITECTURE AND APPROACH

Vulvet is an integrated framework for analyzing and automatic patching of various categories of vulnerabilities discussed in Section 2. The challenges of state-of-the-art vulnerability analysis approaches are high false-positive

³<https://www.checkmarx.com>.

⁴<https://goo.gl/3vp11F>.

Fig. 3. Multi-tier and multi-pronged *Vulvet* architecture.

rate (FPR) and high false-negative rate (FNR). Most of these techniques are based on a single analysis method and inadequate for analysis of most of the vulnerability categories that lead to high false negatives. Moreover, the techniques based on simple taint analysis suffer from high false positives, because a taint slice that is protected by proper validations through the use of control statements is not vulnerable. Similarly, protection of component with proper permission validations or changing these components into “non-exported” in the manifest file, e.g., Listing 5 removes the possibility of exploits.

To overcome the above set of challenges, we propose a novel multi-tier and multi-pronged vulnerability detection and patching approach, *Vulvet*. We show the multi-tier architecture of our approach in Figure 3, which is in contrast to the traditional single-tier approaches proposed in the state-of-the-art. In our approach, the first tier performs manifest analysis and multi-pronged class analysis. This involves methods such as call-graph analysis, data-flow analysis, parameter analysis, taint analysis, return-value analysis, and API analysis. The first tier identifies the suitability of a particular static analysis technique for a given vulnerability and performs analysis of the vulnerabilities (please refer to Table 1) and label them as “true” or “candidate” vulnerabilities. The candidate vulnerabilities may be false alarms and should be reported as “true” only after proper analysis to lower the FPR.

We identify that the precision of any vulnerability analysis approach can be improved by performing proper validation of candidate vulnerabilities. The code Listing 4 shows component in which the precision of analysis depends on conditional checks before performing sensitive operation. An app is considered secure if it properly validates all the incoming Intents and the broadcasts before performing any sensitive operation. Similarly, the code Listing 3 has a vulnerable taint slice (from Source *getStringExtra()* to Sink *delete()*); however, the corresponding declaration in the Listing 5 prevents the component from being exploited. The second tier aims at reducing FPR by incorporation of control-flow analysis (CFA) and Android-specific component validation approach—not used in vulnerability assessment hitherto, to the best of our knowledge. Through our experiments, we identify cases of vulnerabilities for which we have a possibility to reduce the FPR. We show these cases (Sno 5-7, 9, 14-17, 24, 30-32) in Table 1. Later on, we make a novel contribution by automatic patching of the identified vulnerabilities through code instrumentation using Soot. We show the code instrumentation to patch the identified vulnerabilities in tier 3 of Figure 3.

The following subsections describe all three tiers in detail.

Table 1. Vulnerability Vetting and Patching by Vulvet

Sno	Category	Vulnerability	FOIs	Methodology	Ⓔ	Ⓜ	Ⓥ		Ⓐ
							Analysis	Patching	
1	Crypto-graphic	Block cipher in <i>ECB</i> mode	<i>cfeatures</i>	PA	✓	✗	✓	R(PR)	✗
2		Non random IV	<i>cfeatures</i>	DFA	✓	✗	✓	R(SR)	✗
3		Non random <i>salt</i>	<i>cfeatures</i>	DFA	✗	✗	✓	R(SR)	✗
4		Non random <i>key</i>	<i>cfeatures</i>	DFA	✗	✗	✓	R(SR)	✗
5	ICC	Dynamic broadcast receiver	<i>cfeatures</i> <i>mfeatures</i>	PA	✗	★	✓	R(PR)	✗
6		Non-validation of Broadcast	<i>cfeatures</i> <i>mfeatures</i>	TA & DFA & CFA	✗	✗	✓	R(CR)	✓
7		Non-validation of Ordered Broadcast	<i>cfeatures</i> <i>mfeatures</i>	TA & DFA & CFA	★	✗	✓	R(CR)	✓
8		Sending Sticky Broadcast	<i>cfeatures</i>	API	★	✗	✓	R(PR)	✓
9		Non-validation of Implicit Intent	<i>cfeatures</i> <i>mfeatures</i>	TA & DFA & CFA	✗	✗	✓	R(CR)	✓
10		Empty PendingIntent	<i>cfeatures</i>	DFA	✗	✗	✓	F	✗
11		Implicit PendingIntent	<i>cfeatures</i>	DFA	✗	✗	✓	F	✗
12		Service Invocation with Implicit Intent	<i>cfeatures</i>	DFA	✗	✗	✓	F	✗
13	Storage	Inadequate path permission	<i>mfeatures</i>	T&AA	✗	✗	✓	R(MR)	✗
14		External storage (write)	<i>cfeatures</i> <i>mfeatures</i>	TA & DFA	★	✗	✓	F	✗
15		External storage (read)	<i>cfeatures</i> <i>mfeatures</i>	DFA	★	✗	✓	F	✗
16		Internal storage (write without sanitization)	<i>cfeatures</i> <i>mfeatures</i>	TA & CFA	★	✗	✓	F	✗
17		Internal to External Storage	<i>cfeatures</i> <i>mfeatures</i>	TA & CFA	✗	✗	✓	F	✗
18		SQLite raw query	<i>cfeatures</i>	DFA	✓	✗	✓	F	✗
19		SQLite injection	<i>cfeatures</i>	DFA	✓	✗	✓	F	✗
20	Web	DB with world writable mode	<i>cfeatures</i>	PA	✓	✗	✓	R(PR)	✗
21		Data transfer using HTTP connection	<i>cfeatures</i>	PA & TA	✗	✗	✓	F	✗
22		Incorrect hostname verification	<i>cfeatures</i>	RVA	✓	✓	✓	R(ER)	✗
23		Invalid certificate authority	<i>cfeatures</i>	RVA	✓	✓	✓	R(PR)	✗
24		Non-validation of Intent with URI	<i>cfeatures</i> <i>mfeatures</i>	TA & CFA	✗	✗	✓	R(CR)	✓
25		File access allowed in webview	<i>cfeatures</i>	PA	✗	✓	✓	R(PR)	✗
26		@JavaScriptInterface annotation	<i>cfeatures</i>	TA	✓	✗	✓	F	✗
27		Ignoring SSL warnings	<i>cfeatures</i>	API	✗	✓	✓	R(PR)	✗
28		Weak intercept request in webview	<i>cfeatures</i>	RVA	✗	✗	✓	F	✗
29		Overriding URL in webview	<i>cfeatures</i>	RVA	✗	✗	✓	F	✗

(Continued)

Table 1. Continued

Sno	Category	Vulnerability	FOIs	Methodology	Ⓢ	Ⓣ	Ⓥ		Ⓐ
							Analysis	Patching	
30	Runtime Permission	Check caller or self permission	<i>cfeatures</i> <i>mfeatures</i>	API	✗	✗	✓	R(MR)	✗
31		Check caller permission	<i>cfeatures</i> <i>mfeatures</i>	DFA & CFA	✗	✗	✓	R(MR)	✗
32	Networking	Communication over open ports	<i>cfeatures</i>	TA & CFA	✗	✗	✓	F	✗
33		Unencrypted socket	<i>cfeatures</i>	TA & API	✓	✗	✓	R(SR)	✗
34	Miscellaneous	Weak permission	<i>mfeatures</i>	T&AA	✗	✗	✓	R(MR)	✗
35		Exported Component	<i>mfeatures</i>	T&AA	✗	✗	✓	R(MR)	✗
36		Component with weaker permission	<i>mfeatures</i>	T&AA	✗	✗	✓	R(MR)	✗

Ⓢ - Findseccbugs, Ⓣ - JAADS, Ⓥ - *Vulvet*, Ⓐ - AppSealer, PA - parameter analysis, T&AA - tag & attribute analysis, DFA - data-flow analysis, TA - taint analysis, RVA - return value analysis, CFA - control-flow analysis, ✓ - supported analysis, ✗ - not support analysis, ★ - partially supported analysis, F - Forewarning, R(CR) - Reformation using control-flow instrumentation, R(PR) - Reformation using methods or parameters reconstruction, R(SR) - Reformation using secure method calls augmentation, R(MR) - Reformation using manifest modification, R(ER) - Reformation using code elimination.

Definition 1: Features of Interest (FOIs) are programming constructs belonging to apps including Source⁵ & Sink⁶ APIs, parameters of sensitive⁷ APIs, sensitive base classes of components, sensitive overridden methods of base classes (referred to as *cfeatures*), and *attributes* values of various *tags* defined in manifest file such that their improper usage may lead to exploits (referred to as *mfeatures*).

3.1 Class Analysis

This section covers the details of *Class Analysis* along with the importance of combining different analysis methods in *Vulvet*.

3.1.1 Motivating Examples. Taint analysis is an approach for finding information-flow between two specified points in the application, known as “Source” and “Sink.” Many of the solutions such as References [11, 25] use taint analysis for extracting vulnerable paths from Source to Sink. However, our observation on various examples portrays that many vulnerabilities remain undetected using taint analysis alone. Listings 7 and 8 illustrate this fact with the help of an example of password-based encryption. Both the code listings (7 and 8) show the code for generating secret key for symmetric encryption algorithm, i.e., Advance Encryption Standard (AES). The Listing 7 uses the *salt* in *init()* method of *PBEKeySpec* class for generating *KeySpec* (line 2). Here, *salt* is defined in some resource of the app and thus is not random (line 1). This suggests that the key can be reconstructed by an attacker if the attacker has access to the password. The authors in Reference [3] have shown successful attack due to a weak salt.

To prevent the exploit, Listing 8 uses *SecureRandom.nextBytes()* method, which provides a strong random *salt*. The analysis of vulnerable code requires flow analysis of variable *salt* instead of taint analysis. The resultant code is safe if the variable *salt* is randomized before being used in *PBEKeySpec*.

The other vulnerable example is observed while using *PendingIntent*, as shown in Listing 9. *PendingIntent* looks like sending a future Intent to other component that grants the authority to the receiver component to use

⁵One that allows access to sensitive information.

⁶One that leaks sensitive information.

⁷One that can be exploited.

```

1 String salt = getContext().getResources().getString(R.string.salt);
2 KeySpec kS = new PBEKeySpec>Password.toCharArray(), salt.getBytes(), 1000, 256);
3 SecretKeyFactory sKF = SecretKeyFactory.getInstance("AES");
4 byte[] kBytes = sKF.generateSecret(kS).getEncoded();
5 SecretKey key = new SecretKeySpec(kBytes, "AES");

```

Listing 7. Vulnerable password-based encryption. {Vulvet Methodology: Data-Flow Analysis}.

```

1 byte[] salt = new byte[16];
2 new SecureRandom().nextBytes(salt);
3 KeySpec kS = new PBEKeySpec(Passphrase.toCharArray(), salt.getBytes(), 1000, 256);
4 SecretKeyFactory sKF = SecretKeyFactory.getInstance("AES");
5 byte[] kBytes = sKF.generateSecret(kS).getEncoded();
6 SecretKey key = new SecretKeySpec(kBytes, "AES");

```

Listing 8. Secure password-based encryption.

```

1 Intent iserv = new Intent();
2 iserv.setAction("benign.myServ");
3 iserv.putExtra("password", "password");
4 PendingIntent pi = PendingIntent.getService(getApplicationContext(), 0, iserv, PendingIntent.
    FLAG_UPDATE_CURRENT);

```

Listing 9. Vulnerable PendingIntent. {Vulvet Methodology: Data-Flow Analysis}.

permissions of sender app to execute some predefined task. The component containing predefined task is specified in *PendingIntent* through a base Intent that may be explicit or implicit. However, specifying implicit Intent may lead to Intent hijacking attacks. As shown in Listing 9 (lines 1–4), base Intent (3rd parameter) of *PendingIntent.getService()* is implicit Intent to Service. Therefore, whenever any action on *PendingIntent* is performed, the malicious Service may hijack the Intent and steal sensitive information. To discover this type of vulnerability, we perform data-flow analysis (forward and/or backward).

3.1.2 Proposed Class Analysis Approach. Vulvet's current supported analysis belongs to categories including Cryptography, ICC, Storage, Web, Runtime-Permission, and Networking, as shown in Table 1. We employ data-flow analysis, parameter analysis, return value analysis, API analysis, and taint analysis for analyzing class files. Table 1 mentions the analysis methodology targeted for analyzing each vulnerability. These methods are discussed as follows:

- **Data-flow Analysis.** Data-flow analysis (DFA) of parameters is critical, as the vulnerable values lead to a possible exploitation. We utilize DFA in the backward and forward directions to find out the source of the parameters and confirm if the parameters are appropriately constructed or not before being used in the sensitive API calls. We perform backward slicing to find out the source of parameters and forward analysis for analyzing various constraints for given parameters of sensitive APIs. Vulvet creates backward slices of the parameters to the sensitive methods like *StartActivity()* method used in Figure 1. Vulvet performs analysis on all the critical parameters, such as Intent being invoked, salt, and permission of sensitive methods. A unit graph is a control-flow graph that contains program statements as nodes. There is an edge between two nodes N_1 and N_2 if a control can flow from N_1 to N_2 during runtime. We construct unit graph using Soot and compute backward slice starting from the sensitive instruction (let the location be L_E). The backward slice contains all the instructions that contribute to data-flow at this location. The extracted slice is then analyzed from bottom to find the latest definition (instantiation) of the required parameter (let this location be L_S) using def-use chains provided by Soot [10]. We perform flow-sensitive, and inter-procedural data-flow analysis. If def-use chain contains a user-defined function call, we further perform def-use chain analysis within the called function. After completing def-use chain analysis for

Table 2. *Vulvet Class Analysis Details*

Methodology	Description
Parameter Analysis	Presence/absence of parameters in sensitive APIs
	Values of parameters using Regular Expressions
	Invoke data-flow analysis for parameters requiring slicing
	Hardcoded parameters
Data-Flow Analysis	Backward slice computation for parameters
	Forward analysis to validate invocation of sensitive methods in the slice on the parameters
	Analysis of formal arguments of sensitive methods and their flow
	Forward analysis for type inference of parameters of APIs
Control-Flow Analysis	Check binder interface for permission enforcement APIs
	Analysis of control-flow edges
	Reformation and analysis of conditions of If/Else blocks
Return Value Analysis	Exit node values of sensitive methods
API Analysis	Use of vulnerable APIs in sensitive methods
Taint Analysis	Sources and Sinks configuration
	Analysis of paths from sensitive Sources to Sinks

called function, its return value, Jimple variables, and/or some parameters may be added in flow sets and def-use chain analysis is resumed back from the context of caller function. Therefore, the analysis process is inter-procedural. Once the definition is found, it further performs the analysis in forward direction starting from L_S up to L_E based on vulnerability. During forward analysis, various checks are performed based on vulnerability. For the example of Figure 1, it checks whether the instantiated Intent object is implicit or explicit based on all the methods used for this object (*init()* method in this example). Further, the analysis of type of parameters, i.e., *String* of *init()* method confirms the type of Intent. The vulnerability is reported if it is implicit Intent. In Android, there are various ways to define explicit and implicit Intents. We implemented analysis for all possible ways of using explicit and implicit Intents.

Vulvet computes backward slice from various methods used for reading file to avoid any read of overwritten data stored in external storage. If the file path is specified using external storage methods provided by Android like *getExternalFilesDir(null)*, then it computes forward slice to check that proper cryptographic hash function is applied or not to confirm the data validity and report vulnerability accordingly.

- **Parameter Analysis.** Parameter analysis aims to determine if certain parameters are present in calls of sensitive methods, e.g., presence of third parameter of *registerReceiver* method ensures that the calling component has sufficient privileges to call dynamically registered receiver. It analyzes parameters and checks if they have hardcoded values or values that satisfy given pattern. As mentioned in Table 2, validations are performed using regular expressions. It invokes data-flow analysis on parameters, if required.
- **Return Value Analysis.** The research community utilizes return values of methods in different ways in diverse applications. IntelliDroid [26] utilizes return values to guide target path construction for dynamic analysis. SUSI [27] is a machine learning-based classification approach that uses type of return value as a feature in classification. *Vulvet* analyzes return values of sensitive methods to confirm vulnerabilities at Sno 22, 23, 28, and 29 in Table 1. For this, *Vulvet* constructs unit graph of method whose exit values need to be analyzed. Soot provides *getTails()* method, which returns all exit nodes of a given method. For each exit node, if it is a return statement, then we validate the return value. One such example is shown

in Listing 6 that requires the analysis of return values of method *shouldInterceptRequest* whose base class is *WebViewClient*. The methods without any *non-null* value of exit nodes will be flagged as vulnerability by *Vulvet*.

```

1 public class MyWVClient extends WebViewClient {
2 @Override
3 public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
4 handler.proceed();
5 }
6 }

```

Listing 10. Vulnerable WebViewClient. {*Vulvet* Methodology: API Analysis}.

- **API Analysis.** The calls of certain APIs are vulnerable, such as shown in Listing 10. The call of *proceed* API of *SslErrorHandler* class inside *onReceivedSslError()* method (line 4) is vulnerable, as this means that app ignores the SSL Errors. *Vulvet* adds this check as part of API analysis. A safe solution is to call the *cancel()* method and display appropriate error message.
- **Taint Analysis.** Taint analysis finds information flow between Sources and Sinks. Taint is like a “tag,” which starts with Source *S* and flows along with *S* until a Sink is reached. Taint analysis tracks where the information retrieved by *S* is transferred up to any Sink. We perform analysis of sensitive parameters by implementing backward data-flow analysis while the flow of information between sensitive sources (which obtain values from external users or apps) and sinks with taint analysis. FlowDroid [28] proposes and implements taint analysis approach for Android apps. Therefore, we leverage FlowDroid to implement taint analysis and configuration file that contains sources and sinks. E.g., writing of any sensitive data to external storage without proper encryption is reported as vulnerability by *Vulvet*. As shown in Column 5 of Table 1, *Vulvet* analyzes encryption of data using data-flow analysis along with taint analysis. This shows *Vulvet* improves precision in comparison to the state-of-the-art techniques that are based on a single analysis.

Table 2 summarizes *Vulvet*’s projection in each of the analysis methods. The outputs of *Class Analysis* are TV_{CA} and CV_{CA} , where TV_{CA} is the set of true (independent) vulnerabilities identified from analysis of class files of App *A* (*cfeatures*) and CV_{CA} is the set of candidate vulnerabilities that require validation of false positives (*mfeatures*, *cfeatures* in column 4 (FOIs) or CFA in column 5 (Methodology) in Table 1).

3.2 Manifest Analysis

A number of frameworks [29, 30] use *Manifest Analysis* in their vulnerability detection methods; however, their analysis is limited to finding only a few vulnerabilities. These vulnerabilities include exported components and unsafe permissions. Our proposal *Vulvet* also checks for path permissions for content providers, multitasking vulnerabilities, components guarded by weak permissions, and so on. Moreover, *Vulvet*’s *Manifest Analysis* not only aims to find vulnerabilities but also focuses on analyzing if the usage of permissions and components is safe. Here, by *safe*, we mean that the adversary cannot exploit the component.

Manifest Analysis parses manifest file using *ProcessManifest* class of Soot and generates two outputs: TV_{MI} and SC_{MD} . Here, TV_{MI} is the collection of true (independent) vulnerabilities analyzed by manifest declarations in app. Table 1 shows these vulnerabilities (please refer to rows showing *mfeatures* in Column 4), *FOIs* that include insecure path permissions for content provider, weak permissions, i.e., custom permissions with *normal protectionLevel*, *exported* components, and components protected by weak permissions. SC_{MD} is the collection of all secure components prepared by identifying all permissions and components that are safe.

Other techniques such as AndroBugs [29], AppSealer [25], and SEALANT [2] exhibit higher number of false positives for considered vulnerabilities, as these do not employ any validation. We argue that exported components cannot be exploited if they are guarded by safe permissions. An important example of this is

InadequatePathPermission example from Ghera benchmark [3], which consists of an exported but properly guarded content provider with adequate path permissions. Androbugs reports this component as vulnerable based on exported property, however, *Vulvet* does not mark the component vulnerable, as *Vulvet* performs composite analysis of attributes of manifest to improve F-measure. In composite analysis, all relevant attributes are analyzed before raising alert.

```

1 /* Input from External Source */
2 String FILENAME = intent.getStringExtra("filename");
3 /* Reading from internal storage */
4 FileInputStream fis = openFileInput(FILENAME);
5 InputStreamReader isr = new InputStreamReader(fis);
6 BufferedReader bR = new BufferedReader(isr);
7 StringBuilder sb = new StringBuilder();
8 String line;
9 while ((line = bR.readLine()) != null)
10 sb.append(line);
11 /* Writing to external storage */
12 file = new File(getExternalFilesDir(null), FILENAME);
13 OutputStream os = new FileOutputStream(file);
14 os.write(sb.toString().getBytes());

```

Listing 11. Code snippet of DownloadService component. {*Vulvet* Methodology: Taint Analysis followed by FP-Validation}.

3.3 FP-Validation (False Positive Validation)

State-of-the-art solutions mostly analyze single type of vulnerability and also ignore false alerts. Our proposed approach, *Vulvet*, reduces number of false positives using FP-Validation. Next, we discuss the motivation for FP-Validation with an example. Listings 11 and 12 show code snippets of component *DownloadService* and its declaration in manifest file of an app from our test dataset.

As shown in Listing 11, the component *DownloadService* obtains *FILENAME* through Intent (line 2). It reads the data from specified file (lines 4–10) present in internal storage and further copies this data to external storage (lines 12–14). A taint analysis-based solution will mark this as vulnerable considering flow from *getStringExtra* (source) to *OutputStream.write* (Sink). However, component's declaration in Listing 12 depicts that the component is exported but at the same time it is prevented with signature-level permission (lines 1–2). Therefore, the Intent can be sent by apps of the same developer, which prevent Intent spoofing attacks. To avoid such false positives, *Vulvet* adds FP-Validation. Table 1 shows the cases (*cfeatures*, *mfeatures* in column 4 (FOIs) or CFA in column 5 (Methodology)) for which *Vulvet* performs FP-Validation to avoid generation of false positives.

```

1 <permission android:name="SecurePermission" android:protectionLevel="SignatureOrSystem"></
  permission>
2 <service android:name="DownloadService" android:exported="true" android:permission="
  SecurePermission">
3 </service>

```

Listing 12. DownloadService declaration in manifest. {*Vulvet* Methodology: Tag & Attribute Analysis}.

FP-Validation includes two main analysis approaches, Control-Flow Analysis and Component Validation as follows:

- **Control-Flow Analysis.** *Vulvet* augments Control-Flow Analysis (CFA) for improving precision as taint analysis computes slices from source to sink and reports the vulnerability if some source leaks data to sink. However, if the slice is within the control statements and performs proper checks as detailed in Listing 4, then this is no more a vulnerability. We performed analysis of control statements and their scope using CFA. Control-Flow Graph (CFG) is a collection of all paths that might be traversed in a program during its execution [31]. *Vulvet* constructs the bi-directional inter-procedural CFG using Soot's *buildBiDirICFG* method of class *DefaultBiDirICFGFactory*. With the help of CFG, the control-flow path

between the first unit of root method (top-most caller of current method) and the identified sensitive instruction is obtained. This is to ensure accuracy even when the method containing sensitive operation is being called by other methods. The conditions of *IF* statements along the obtained control-flow path are analyzed to ensure that sufficient validations are implemented by developers to ensure components are invoked only through implicit Intent. The receiver may also validate sender of Intent through conditional checks before any sensitive operation. These checks include validations of name of calling component, host name of sender if called by external app, package name of received Intent, *action* of the Intent, and so on. We implement all such checks in *Vulvet* to improve precision before reporting vulnerability. E.g., for Listing 4, *Vulvet* finds that the *actions* are properly checked and, hence, it is not a vulnerable code. Table 1 shows (CFA in methodology) all the cases for which control-flow analysis is implemented. The implementation of CFA is done as part of *Class Analysis*.

- **Component Validation.** This module takes sets CV_{CA} and SC_{MD} (outcomes of *Class Analysis* and *Manifest Analysis*, respectively) as inputs. It validates the candidate vulnerabilities present in app's components (CV_{CA}) using the results of *Manifest Analysis* (SC_{MD}). For each vulnerability of set CV_{CA} , *Vulvet* validates whether the corresponding component is present in set SC_{MD} or not. The presence confirms that the vulnerability cannot be exploited as component declaration is safe and cannot be invoked by unauthorized external apps. Therefore, this candidate is added to the set of "true" vulnerabilities.

3.4 Vulnerability Resolution

Vulnerability Resolution approach of *Vulvet* aims at generating patches in such a way to avoid any interruption in app's original functionality and still securing user against exploits by malicious apps. The patch instruments the safe code specific to vulnerability. All the instrumentations are done in Jimple, the intermediate representation language of Soot. *Vulvet* automatically resolves significant number of vulnerabilities by instrumenting patch. However, patching of some categories may not be straightforward and requires fixing by the developer. In this case, *Vulvet* adds appropriate warning to alert the user/developer. We call the former approach as *Reformation* and later as *Forewarning*. Table 1 (Column 9) shows the approach applied for each case by *Vulvet* for patching. The following subsections explain both in detail.

3.4.1 Reformation (R). *Reformation* automatically generates patches for most of the vulnerability categories mentioned in Table 1 through Jimple instrumentation. During this, *Vulvet* performs patching of different categories using one of the following approaches:

- **Control-flow Instrumentation (CR):** The spoofing attacks caused due to non-validation can be sealed by properly validating the way the component is invoked before performing any sensitive operation. Therefore, our approach of control-flow instrumentation constructs and instruments validation statements to confirm that the component is called with only implicit Intent (Sno 6, 7, 9, and 24 of Table 1). The instrumented validation statements perform runtime check to keep track that *action* of incoming Intent is not *null* (line 3). Further, we add runtime check for Intent *action* value if it is defined during corresponding component declaration in manifest. For this, we retrieve the value of *name* attribute of *action* tags for the component from the manifest file. These *action* values are used to construct parameters of comparison statement and then *if* statement in Jimple intermediate representation (IR) (lines 4–5) of Soot. These validation statements are instrumented as the successor of the Source statement, i.e., *getAction()* statement. Listing 13 shows the instrumented validation statements in Jimple. Table 1 shows these cases marked as R(CR) in Column "Patching."
- **Methods/Parameters Reconstruction (PR):** *Vulvet* patches exploitable method calls by reconstructing safe method calls and swapping with the original exploitable call in Jimple IR. This includes following cases of Table 1 mentioned R(PR) in Column 9:

```

1 $r2 := @parameter1: android.content.Intent;
2 $r3 = virtualinvoke $r2.<android.content.Intent: java.lang.String getAction()>();
3 if $r3 == null goto label1;
4 $z0 = virtualinvoke $r3.<java.lang.String: boolean equals(java.lang.Object)>("android.
    intent.action.DEVICE_STORAGE_LOW");
5 if $z0 == 0 goto label1;
6 /*Original code*/
7 .....
8 .....
9 label1: /*Call to stop component*/

```

Listing 13. Instrumentation of Validation of action.

- Vulnerable encryption mode ECB of *Cipher.getInstance()* method is replaced with CBC mode (Sno 1).
- Registering broadcast receiver dynamically without permission makes it vulnerable. This is handled by adding permission as a parameter that validates the caller to have required permission (Sno 5).
- A Sticky Broadcast is replaced by a simple Broadcast (Sno 8).
- Files are created in MODE_PRIVATE instead of MODE_WORLD_READABLE (Sno 20).
- Listing 14 (line 5) shows vulnerable *MyTrustManager* (line 2) used as second parameter of *SSLContext.init()* (Sno 22). This method call is reconstructed as shown in Listing 15 with *null* as second parameter and swapped with original exploitable call using default but safe *TrustManager*.
- The other bug is ignoring SSL errors without validation through the use of method *proceed()* of class *SslErrorHandler* of class *WebViewClient*. *Vulvet* patches this by instrumenting a comparatively safe method *cancel()* on the same object (Sno 27).

```

1 $r2 = newarray (javax.net.ssl.TrustManager)[1];
2 $r7 = new edu.ksu.cs.benign.MyTrustManager;
3 specialinvoke $r7.<edu.ksu.cs.benign.MyTrustManager: void <init>()>();
4 $r2[0] = $r7;
5 virtualinvoke $r6.<javax.net.ssl.SSLContext: void init(javax.net.ssl.KeyManager[], javax
    .net.ssl.TrustManager[], java.security.SecureRandom)>(null, $r2, null);

```

Listing 14. Vulnerable TrustManager.

```

1 virtualinvoke $r6.<javax.net.ssl.SSLContext: void init(javax.net.ssl.KeyManager[], javax
    .net.ssl.TrustManager[], java.security.SecureRandom)>(null, null, null);

```

Listing 15. Safe TrustManager.

- **Secure Method Calls Augmentation (SR):** A safe solution to prevent the crypto vulnerabilities is to securely randomize salt, IV, and key as described in Section 3.1.1-Listing 8. *Vulvet* randomizes these values by instrumenting *SecureRandom().nextBytes()* method for these parameters before being used. The vulnerability of unencrypted socket communication (Sno 33) is sealed similarly by augmenting safe calls using *SSLSocket*, as shown in Figure 4.
- **Manifest Modification (MR):** *Vulvet* proposes to modify declaration of components and permissions in manifest to heal various vulnerabilities that have FOIs as *mfeatures* (Sno 13, 34, 35, 36 of Table 1). In addition, *Vulvet* protects from vulnerabilities in Runtime-Permission category (Sno 30, 31 of Table 1) by adding permission checks for the components in *AndroidManifest.xml* file and letting Android validate the caller's permission access. The original declaration of *MyService* in *Androidmanifest.xml* of Figure 2 is shown in Listing 16 (line 1). *Vulvet* instruments this with modified declaration as shown in Listing 16 (line 2). By adding permission as part of component declaration ensures that Android validate the permission on every access to component, which is safe in comparison to binder APIs. This permission is same as parameter of APIs discussed in Section 2.5.

```

SocketFactory sf =
    SSLSocketFactory.getDefault();
SocketAddress saddr = new InetSocketAddress(HOST, PORT);
- Socket s = new Socket();
+ SSLSocket s = (SSLSocket)
  s.connect(saddr);
  sf.createSocket(HOST, PORT)
  if (s.isConnected()) {
    InputStream is = s.getInputStream();
    .....
  }

```

Fig. 4. *Vulvet* patches communication through SSLSocket.

```

1 <service android:name=".MyService" android:enabled="true" android:exported="true"/>
2 <service android:name=".MyService" android:enabled="true" android:exported="true"
  android:permission="benign.permission"/>

```

Listing 16. Manifest Instrumentation.

- **Code Elimination (ER):** Improper overriding of default implementations of various sensitive methods increases the chances of exploit even more. Therefore, it is safe to keep default implementations instead of overriding sensitive methods. *Vulvet* works on automated patching of these vulnerabilities by removing the vulnerable overridden methods. As shown in Table 1, *Vulvet* analyzes vulnerable custom implementations of the *X509TrustManager* and *HostNameVerifier* interfaces (Sno 22) and patches by removing overridden methods. This will invoke corresponding default implementations.

3.4.2 *Forewarning (F).* *Vulvet* works on automatically patching various vulnerabilities using Reformation approaches as discussed. However, it is not feasible to perform automated patching for some of the identified vulnerabilities without affecting original functionality of the apps. These are shown by “F” in Column “Patching” of Table 1. These include vulnerabilities having:

- component invocation using implicit Intent (Sno 13–15).
- external storage reading and writing sensitive data (Sno 14–15).
- improper sanitization of paths originating from external resources such as Intent and UI before writing to internal storage (Sno 16).
- copying data from internal storage to external storage (Sno 17).
- raw/non-parameterize queries (Sno 18–19).
- connection to server via HTTP (Sno 21).
- adding an interface to JavaScript for sensitive classes (Sno 26).
- allow loading of all resources in webview (Sno 28).
- non-validation of page requests in webview (Sno 29).
- communication to the server over open ports (Sno 32).

Vulvet instruments appropriate forewarning messages to alert the end-user for all these vulnerabilities. The vulnerability report highlights these cases so developer may correct these.

4 EVALUATION

Vulvet is implemented on top of Soot [10]. *Vulvet* uses Jimple Transformation Pack (jtp) of Soot for patching. We perform all the evaluation on Intel E5-2420v2 2.20 GHz processor with 80 GB memory. The aim of evaluation is to find out answers for the following important research questions:

- What is the accuracy of vulnerability detection for each category using *Vulvet* in comparison with other state-of-the-art tools?
- Is *Vulvet* scalable and capable to analyze real-world apps?

Table 3. Comparative Evaluation of Various Tools on an Open-source Test Dataset

Category	#App	#AV	FSB			Androbugs			Vulvet		
			#TP	#FP	#FN	#TP	#FP	#FN	#TP	#FP	#FN
Crypto	8	4	2	0	2	0	0	4	4	0	0
ICC	26	13	2	0	11	1	1	12	13	0	0
Networking	4	2	1	1	1	0	0	2	2	0	0
Storage	14	7	4	3	3	1	0	6	7	1	0
Runtime-Permission	8	4	0	0	4	4	2	0	4	0	0
Web	18	9	3	0	6	4	0	5	9	1	0
Miscellaneous	6	3	0	0	3	3	0	0	1	0	0
Total	84	42	12	4	30	13	3	29	40	2	0
Average App Size	1.5 MB										
Precision (P) = $TP/(TP+FP)$			75%			81.25%			95.23%		
Recall (R) = $TP/(TP+FN)$			28.57%			30.95%			100%		
F-measure = $2PR/(P+R)$			0.414			0.45			0.975		

- What is the role of code-based vulnerability analysis in alleviating exploits in Android apps?
- What is the overhead of using *Vulvet*?

Table 1 shows the approaches used and presents a comparison with other publicly available vulnerability analysis tools such as FSB [32] and JAADS [33] along with AppSealer [25] (patching framework). The reason for selecting FSB and JAADS is that these are widely available tools used by industries to measure code weaknesses both from source code and byte code. We had to exclude state-of-the-art mentioned in “Related work” from our experiments, because except for SMV-HUNTER, others are not available in public domain. SMV-HUNTER is a hybrid approach mainly for SSL/TLS vulnerability analysis. Our evaluation targets are Android app sets taken from nine different app repositories described in Table 4. Ghera is an open-source benchmark with known vulnerabilities within each app [3], while other repositories contain *apk* files.

For the comparative performance analysis, we analyze each of the above datasets using FSB [32], Androbugs [29], CogniCrypt [1], and our proposed *Vulvet* framework. CogniCrypt has adequately identified all cryptographic vulnerabilities, however, other categories are not in scope of CogniCrypt. Therefore, we do not include it in further evaluations. Table 1 shows partially supported analysis, which means lack of handling false alarms.

4.1 Precision and Recall

Table 3 shows results of three frameworks, i.e., FSB, Androbug, and *Vulvet* on open-source data set Ghera [3]. For each category of vulnerability, the table shows the number of apps analyzed in that category and actual number of vulnerable apps, true positives (TP), false positives (FP), and false negatives (FN) of three frameworks. FSB involves app analysis with various objectives including bad practice, correctness, performance, and so on. Our concern is only on security-related weaknesses. Table 3 evinces that *Vulvet* outperforms other tools with the highest precision (95.23%), highest recall (100%), and highest F-measure (0.975). FSB has lowest recall, as it covers very few vulnerabilities in each category. Androbugs’ recall is low, as it does not include detection of most of the categories. FSB’s precision is the lowest, as it does not include any approach for handling false positives.

4.2 Scalability

Vulvet is an automated framework that integrates static analysis approaches, which makes it scalable. We first analyzed 84 open-source apps as shown in Table 3 and then extended the evaluation on randomly selected *apks*

Table 4. Overhead and Vulnerability Analysis of *Vulvet*

Dataset Name	#App	App Size (Avg)	Time (in Min.)			# of Vulnerable Apps			
			FSB	Androbugs	<i>Vulvet</i>	FSB	Androbugs	<i>Vulvet</i> (WOFP)	<i>Vulvet</i> (WTFP)
Ghera	84	1.5 MB	13	13	15	12	13	43	40
Play Store [34]	944	2.96 MB	209	212	228	17	16	63	57
Playdrone [35]	106	11.82 MB	97	92	105	17	20	25	22
Nduo [36]	900	28 MB	2,075	2,070	2,100	42	46	124	120
Mobomarket [37]	68	18.1 MB	101	98	105	10	12	19	14
APK4Fun [38]	21	6.48 MB	10	10	12	3	4	9	6
Gfan [39]	718	10.34 MB	645	640	660	31	36	67	63
Androidpur [40]	466	4.56 MB	167	160	170	31	28	64	55
APPSAPK [41]	90	5.06 MB	41	41	44	9	8	15	11
Total	3,700	88.82 MB	3,358	3,336	3,439	172	183	429	388
On average per app		9.87 MB	0.908	0.902	0.929				

WOFP - Without FP-Validation, WTFP - With FP-Validation.

from eight different online stores and observed that increasing the number of apps increased analysis time linearly. Table 4 shows that *Vulvet* analyze 3,700 apps with an average size of 9.87 MB in 57 hours and 19 minutes (3,439 minutes). Further, for validation, we randomly select a few apps from each dataset and constructed malicious apps to exploit these. We could develop successful proof-of-concept attacks on ICC category, storage category, and runtime-permission categories. *Vulvet* identifies app⁸ with non-validation of broadcast senders vulnerability from *nduo* dataset. We exploit the app by successfully invoking two broadcast receivers of this app, namely, “*com.tapjoy.TapjoyReferralTracker*” and “*safiap.framework.CheckUpdateReceiver*” where broadcast receiver *safiap.framework.CheckUpdateReceiver* has the functionality of setting alarm with repeating period. We invoke receiver *safiap.framework.CheckUpdateReceiver*, along with package name of calling app and repeating duration as *extras* of Intent.

The results of *Vulvet* identified that some popular apps such as *FireChat* (*com.opengarden.firechat*) and *Flash-Light* (*com.peacock.flashlight*) also have vulnerable codes. We analyzed BHIM, a popular payment app using *Vulvet* and found that BHIM lacks proper randomization in the use of initialization vector and loads URL in webview passed as parameter. This shows the possibility of the future exploits according to the CWE-329, where weak initialization vectors are susceptible to the dictionary attacks with the chosen plain text [42]. The use of the same initialization vector for all the messages sent within a single connection may lead to brute-force attacks [43]. A simple improper input validation led to Heartbleed attack in the OpenSSL’s implementation of the TLS/DTLS, which allowed an attacker to retrieve a block of memory of the vulnerable server [44]. However, implementing a dictionary attack requires a large amount of time, computing power, and capturing a large number of instances, which is complex but not infeasible by expert attackers. While apps such as *WhatsApp* (*com.whatsapp*), *CleanMaster* (*com.cleanmaster.mguard*), *GoldenDictionary* (*mobi.goldendict.android.free*), *HD Video Player* (*com.elift.hdplayer*) are found to have secure implementation of various components. The experiments and results on 3,700 real-world Android apps collected from eight different Android markets demonstrate the scalability of *Vulvet*.

4.3 Role of Code-based Vulnerability Analysis

A majority of today’s smartphone users have a tendency to install apps without much understanding of different permissions acquired by the apps during the installation. Additionally, most of these users may not have

⁸Hash-061bd4424eeebb31384491261ce1af46, Packagename-“com.com2us.threekingdomdefense2.normal.freefull.mm.cn.android.common.”

information necessary to understand the privacy disclosure statements. This fact leads to the installation of malicious malware apps that are capable to perform exploits due to the presence of vulnerable and weak codes present in other benign apps available on the smartphone. Column 5 of Table 4 shows the number of apps having one or more vulnerabilities in each dataset. The evaluation results in Table 4 depict that 11.59% (429/3,700) of apps of experimental dataset are identified as “candidate” after second tier. This reduced to 10.46% (388/3,700) by third tier and labeled as “true” vulnerable. We also observe that most of the exploits are due to weak implementations of sending and receiving Intents, web access, and broadcasts. *Vulvet* could discover vulnerabilities in widely used apps (more than 1M downloads) such as *GeeksforGeeks (V9.0.11) (free.programming.programming)*, *Coolwinks (V2.42) (com.coolwinks.cwapp)*, and *Bazzi Now (V2.0.54) (com.brainbaazi)*. These vulnerabilities are a subset of common code-based implementation vulnerabilities. We see that ensuring developer knowledge of safe and secure coding practices involving risk assessment and threat modeling of smart-phone apps help in avoiding many of these vulnerabilities.

4.4 Overhead Analysis

Vulvet includes analysis for a wide range of vulnerabilities compared to the state-of-the-art approaches and also includes FP-Validation. Therefore, it requires slightly more time compared to approaches based on single type of analysis. To calculate the exact overhead of using *Vulvet*, we collate the time required by multiple tools for the app analysis. Table 4 shows the time taken by all three frameworks for different datasets. The average analysis time per app with FP-Validation is 0.929 minutes (55.77 seconds) for the average size of 9.87 MB app. The results manifest that, on average, *Vulvet* has very small, i.e., 2.3% timing overhead compared to others. We stress that the small overhead can be overlooked, considering high precision, recall, and F-measure.

5 LIMITATIONS OF VULVET

Vulvet provides an efficient security vetting framework; however, in terms of soundness, we try to add detection of most of the known vulnerabilities based on detailed study of vulnerable apps, but still *Vulvet* cannot be claimed to cover all possible vulnerabilities. In terms of completeness (i.e., free from false positives), analysis of a few vulnerabilities like read from external storage may be over-approximation. The reading is vulnerable, as the files may have been modified by some malicious apps; however, *Vulvet* does not perform any such check, and all reading operations from external storage are marked as vulnerable. *Vulvet* lacks in analyzing and patching vulnerabilities in native code. Due to non-support of Java reflection and dynamic code loading by Soot, which is the underlying framework, *Vulvet* may miss some flows that include these features.

6 RELATED WORK

A major focus of the security community related to Android apps has been on malware detection and classification [45], clone detection [46], repackaged malware analysis [47], user interface similarity detection [48], vulnerability analysis [25], and collusion analysis [49]. Research work in these directions employ static, dynamic, and hybrid analysis techniques. Our focus in this article is on vulnerabilities induced due to insecure coding by the programmers. Therefore, we limit our discussions in this section to some of the important state-of-the-art related to vulnerability analysis.

Mitra et al. [3] developed benchmark repository called Ghera, which consists of more than 80 open-source Android apps belonging to different vulnerability categories along with exploit(s) for each of them. The authors constructed proof of concept attacking apps to show that the unintentional flaws by developers can be exploited using simple traps. These examples put emphasis on the need to focus on vulnerability analysis of Android apps. Observing the effects of the insecure coding, countries including Japan [50], USA [51], and Taiwan [52] announced secure mobile app development guidelines for legal compliance.

Table 5. Comparison of *Vulvet* with Other Approaches in the State-of-the-Art

Framework	Category					
	ICC	Storage	Web	Cryptographic	Runtime-Permission	Networking
OpenCCE [53]	✗	✗	✗	✓	✗	✗
CogniCrypt [1]	✗	✗	✗	✓	✗	✗
Sadeghi et al. [54]	P (inter-app)	✗	✗	✗	✗	✗
Lei et al. [12]	P (Service)	✗	✗	✗	✗	✗
SEALANT [2]	✓	✗	✗	✗	✗	✗
AppSealer [25]	✓	✗	✗	✗	✗	✗
MAS [30]	✗	P	P	P	✗	✗
CHEX [55]	✓	✗	✗	✗	✗	✗
SMV-HUNTER [56]	✗	✗	P	✗	✗	✗
Androbugs [29]	✗	P	✗	P	P	P
JAADS [33]	✓	✗	✓	✗	✗	✗
AndroidLint [57]	✗	✗	✗	✗	✗	✗
Vulvet	✓	✓	✓	✓	✓	✓

✓ - Covered, ✗ - Uncovered, P - Partial.

A major list of past contributions in Android focuses on analysis of only single code-based vulnerabilities, as shown in Table 5. OpenCCE [53] and CogniCrypt [1] are two substantial tools focused on identification of cryptographic vulnerabilities at the source-code level on sensitive APIs and parameter analysis. The work mainly aims at generating secure code for various cryptographic operations based on user's requirement. This limits its applicability for wider use.

Sadeghi et al. [54] worked on inter-app security vulnerabilities using compositional analysis. A good discussion on detection and exploitation of Intent message vulnerabilities using taint analysis is presented in Reference [58]. Lei et al. [12] worked on vulnerabilities owing to invocation of services through implicit Intents. Android forbids implicit service invocations since version 5.0, however, the authors show that latest Play Store apps were still using implicit Intents. This indicates that all Play Store apps are not updated as per the requirement of new releases. Another framework, SEALANT [2], focused on detection of Intent spoofing and unauthorized Intent receipt vulnerabilities across apps. AppSealer [25], which is most relevant to *Vulvet*, also works on generating patches for vulnerable Android apps. However, it is limited to detection and patching of component hijacking vulnerabilities. Also, it does not perform control-flow analysis and manifest validation during detection, which leads to a low precision. MAS [30] implemented a method for identification of Android vulnerabilities. The approach is heavy-weight due to inclusion of dynamic analysis. The static analysis of MAS is limited to parameter analysis of APIs. CHEX [55] is a static analyzer based on data-flow analysis for detecting flows that may lead to Intent hijacking attacks. However, its low precision and capability to handle Intent hijacking vulnerabilities makes it unsuitable for actual analysis.

Poeplau et al. [59] have proposed super control-flow graph-based approach for analyzing apps vulnerable to injection attacks during dynamic code loading. However, the approach proposes changes in the Android source code for preventing DCL-based attacks, which becomes an obstacle for fast and widespread adoption. SMV-Hunter [56] worked on automated detection of SSL/TLS vulnerabilities due to custom implementation of the *X509TrustManager* and *HostNameVerifier* interfaces in Android apps.

Androbugs [29] takes *apk* as input and identifies exported components and dangerous permissions from extracted manifest file. Experiments by Oyetoyan et al. [13] using seven real-world open-source apps and our analysis on 3,700 apps shows that Androbugs has a large number of false negatives, as it does not include analysis of many vulnerabilities. AndroBugs marks vulnerabilities based on dangerous permissions required by apps,

Table 6. Comparison of *Vulvet* with Other Approaches in the State-of-the-Art

Framework	Feature			
	Technique	FPS	Automated Patching	FNs
OpenCCE [53]	API, Parameter	Low	✗	High
CogniCrypt [1]	API, Parameter	Low	✗	High
Sadeghi et al. [54]	TA	Low	✗	High
Lei et al. [12]	RA	Low	✗	High
SEALANT [2]	DFA, Pattern Matching	High	✗	High
AppSealer [25]	TA	High	✓	High
MAS [30]	API	High	✗	High
CHEX [55]	DFA	High	✗	High
SMV-HUNTER [56]	Hybrid	Low	✗	High
Androbugs [29]	Based on Androguard	Average	✗	High
JAADS [33]	TA	High	✗	High
AndroidLint [57]	XML Parsing	High	✗	High
<i>Vulvet</i>	Multi-tier, Multi-pronged	Very Less	✓	Very Less

✓ - Covered, ✗ - Uncovered.

TA - Taint Analysis, CFA - Control-Flow Analysis, RA - Reachability Analysis.

however, this may not be a cause of a vulnerability in many apps. The manifest check based only on exported components contributed to false positives for many apps. The dataset of apps writing to and/or reading sensitive data from external storage was not identified as vulnerable by Androbugs. The reason for the false negatives was that Android 4.4+ permit read/write access through API `context.getExternalFilesDir(null)` without any requirement of declaring dangerous permission in manifest.

Oyetoyan et al. [13] manually evaluated state-of-the-art static analysis tools against eight different mobile-related weaknesses. Out of six evaluated tools, four tools, Amandroid [60], AndroBugs [29], AndroidLint [57], and JAADS [33], were Android-specific and another two, FindBugs and FSB (Findsebugs), were not specific to Android and support Java and web applications. These tools were evaluated against seven real-world open-source apps having vulnerabilities. The authors concluded through evaluation that none of the work is individually sufficient in detecting most of the weak codes. Using Table 5, we conclude that almost all techniques focus on single domain of vulnerabilities, while *Vulvet* has wide coverage. Table 6 shows that past contributions suffer from high false positives and negatives in comparison to *Vulvet*. Also, except for AppSealer [25], none of the frameworks provide vulnerability patching after analysis. AppSealer focuses on resolving only Intent-based vulnerabilities.

7 CONCLUSION

In this article, we present *Vulvet*, an integrated approach for vetting and patching vulnerabilities belonging to a large set of categories within the Android applications. Our proposed approach integrates a number of analysis techniques from different domains to make detection of a diverse range of vulnerabilities feasible. *Vulvet* performs static analysis of both manifest and class files. Furthermore, the addition of FP-Validation to improve the F-measure and automated patching to mitigate exploits are two important novel aspects of *Vulvet*.

Evaluation shows that *Vulvet* has reduced false negatives in comparison to the state-of-the-art. *Vulvet* is evaluated using nine datasets that include both benchmark and real-world apps. We provide comparison with various vulnerability analysis and patching tools. The assessment depicts that *Vulvet* can precisely detect and patch the vulnerabilities of all the categories it is programmed to handle yielding the highest recall. *Vulvet* can be used by developers not trained in security for validations to secure sensitive data before the release of applications. *Vulvet* alleviates the exploits owing to the weak implementation of Android programming

constructs by instrumenting *apk* with secure code. The analysis process is fast, completely automatic, and confines the number of false positives and false negatives. In summary, we combine various static analysis approaches for finding candidate vulnerabilities with control-flow analysis and manifest revalidation to prevent false positives. Additionally, we propose automated vulnerability-specific patching for most of the identified vulnerabilities that distinguish *Vulvet* from the state-of-the-art. As part of future work, we will work on developing an approach that can automatically generate malicious exploits to trigger vulnerable behaviors. This will be beneficial for validating the identified vulnerabilities. Further, we work on adding parallelization to the technique, supporting analysis of native code, and improvement in precision of flow analysis.

REFERENCES

- [1] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. 2017. CogniCrypt: Supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 931–936.
- [2] Youn Kyu Lee, Jae Young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A SEALANT for inter-app security holes in Android. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE’17)*. IEEE, 312–323.
- [3] Joydeep Mitra and Venkatesh-Prasad Ranganath. 2017. Ghera: A repository of Android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 43–52.
- [4] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2016. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Trans. Reliab.* 65, 1 (2016), 54–69.
- [5] Yuji Kosuga, Kenji Kono, Miyuki Hanaoka, Miho Hishiyama, and Yu Takahama. 2007. Sania: Syntactic and semantic analysis for automated testing against SQL injection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC’07)*. IEEE, 107–117.
- [6] V. Benjamin Livshits and Monica S. Lam. 2005. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the USENIX Security Symposium*, Vol. 14.
- [7] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 6–pp.
- [8] Omer Tripp and Omri Weisman. 2018. Identifying stored security vulnerabilities in computer software applications. US Patent 9,904,786.
- [9] MITRE. 2018. Common Vulnerabilities and Exposures (CVE). Retrieved from <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Android>.
- [10] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: A retrospective. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS’11)*, Vol. 15. 35.
- [11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. ACM, 239–252.
- [12] Lingguang Lei, Yi He, Kun Sun, Jiwu Jing, Yuwu Wang, Qi Li, and Jian Weng. 2017. Vulnerable implicit service: A revisit. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS’17)*. ACM, New York, NY, 1051–1063. DOI: <http://dx.doi.org/10.1145/3133956.3133975>
- [13] Tosin Daniel Oyetoyan and Marcos Lordello Chaim. 2017. Comparing capability of static analysis tools to detect security weaknesses in mobile applications. In *CEUR Workshop Proceedings* Vol. 1977. 8–18.
- [14] Lori Flynn. 2015. DRD09. Restrict access to sensitive activities. Retrieved from <https://wiki.sei.cmu.edu/confluence/display/android/DRD09.+Restrict+access+to+sensitive+activities>.
- [15] Gaku Mochizuki. 2015. JVN#37825153 AirDroid for Android vulnerable in handling of implicit intents. Retrieved from <http://jvn.jp/en/jp/JVN37825153/>.
- [16] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. 2006. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the USENIX Security Symposium*. 121–136.
- [17] Yajin Zhou Xuxian Jiang and Zhou Xuxian. 2013. Detecting passive content leaks and pollution in Android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS’13)*.
- [18] Hiroshi Kumagai. 2014. JVN#55438786 Content Provider in CamiApp for Android fails to restrict access permissions. Retrieved from <http://jvn.jp/en/jp/JVN55438786/index.html>.
- [19] Alberto Ornaghi and Marco Valleri. 2003. Man in the middle attacks demos, 2003. <https://bbs.pku.edu.cn/attach/53/35/533561f8d8187eb6/ManInMiddle.pdf> (visited: 2020-05-04).
- [20] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Proceedings of the 27th Computer Security Applications Conference*. ACM, 343–352.
- [21] NIST. 2018. CVE-2018-5298 Detail. Retrieved from <https://nvd.nist.gov/vuln/detail/CVE-2018-5298>.

- [22] MITRE. 2014. CVE-2014-5930. (2014). Retrieved from <https://www.cvedetails.com/cve/CVE-2014-5930/>.
- [23] MITRE. 2014. CVE-2014-7609. (2014). Retrieved from <https://www.cvedetails.com/cve/CVE-2014-7609/>.
- [24] Checkmarx. 2018. Common Vulnerabilities and Exposures (CVE). Retrieved from <https://www.checkmarx.com/2018/01/23/tinder-someone-may-watching-swipe-2/>.
- [25] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'14)*. Citeseer.
- [26] Michelle Y. Wong and David Lie. 2016. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*, Vol. 16. 21–24.
- [27] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*, Vol. 14. Citeseer, 1125.
- [28] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive, and lifecycle-aware taint analysis for Android apps. *ACM Sigplan Not.* 49, 6 (2014), 259–269.
- [29] Yu-Cheng Lin. 2015. Androbugs Framework Project. Retrieved from https://github.com/AndroBugs/AndroBugs_Framework.
- [30] Chia-Wei Tien, Tse-Yung Huang, Ting-Chun Huang, Wei-Ho Chung, and Sy-Yen Kuo. 2017. MAS: Mobile-apps assessment and analysis system. In *Proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W'17)*. IEEE, 145–148.
- [31] Frances E. Allen. 1970. Control flow analysis. *ACM Sigplan Not.*, Vol. 5. ACM, 1–19.
- [32] 2018. Findsecbugs. Retrieved from <https://find-sec-bugs.github.io/>.
- [33] Edward Flanker and Anant Shrivastava. 2014. Joint Advanced Defect assessment for Android applications. Retrieved from <https://github.com/flankerhq/JADAS>.
- [34] Google Play market. Retrieved from <http://play.google.com/store/apps/>.
- [35] 2014 PlayDrone Android Apps. Retrieved from <https://archive.org/details/android>.
- [36] Nduo Market. Retrieved from <https://www.nduo.cn/>.
- [37] Mobomarket. Retrieved from <https://mobomarket.jaleco.com/>.
- [38] APK4Fun. Retrieved from <https://www.apk4fun.com/>.
- [39] GFAN. Retrieved from <http://apk.gfan.com/>.
- [40] Androidpur. Retrieved from <http://androidpur.org/>.
- [41] APPSAPK. Retrieved from <https://www.appsapk.com/>.
- [42] CWE-329. Retrieved from <https://cwe.mitre.org/data/definitions/329.html>.
- [43] Animesh Chhotaray, Adib Nahian, Thomas Shrimpton, Domenic Forte, and Mark Tehranipoor. 2017. Standardizing bad cryptographic practice: A teardown of the IEEE standard for protecting electronic-design intellectual property. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1533–1546.
- [44] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the Conference on Internet Measurement*. ACM, 475–488.
- [45] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1105–1116.
- [46] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. Wukong: A scalable and accurate two-phase approach to Android app clone detection. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 71–82.
- [47] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*. ACM, 317–326.
- [48] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. 2015. Detecting clones in Android applications through analyzing user interfaces. In *Proceedings of the IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 163–173.
- [49] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*. ACM, 71–85.
- [50] Japan Smart Phone Security Association. 2016. Android Application Secure Design/Secure Coding Guidebook. Retrieved from https://www.jssec.org/dl/android_securecoding.pdf.
- [51] Steve Quiroga, Jeffrey Voas, Tom Karygiannis, Christoph Michael, and Karen Scarfone. 2015. Vetting the Security of Mobile Applications. Retrieved from <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-163.pdf>.
- [52] Taiwan Industrial Development Bureau Ministry of Economic Affairs. 2016. Mobile Application Security Guideline. Retrieved from <http://www.mas.org.tw/spaw2/uploads/files/1050219-1.pdf>.
- [53] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. 2015. Towards secure integration of cryptographic software. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 1–13.

- [54] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2015. Analysis of Android inter-app security vulnerabilities using COVERT. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*, Vol. 2. IEEE, 725–728.
- [55] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 229–240.
- [56] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'14)*. Citeseer.
- [57] Google Developers. 2018. Android Lint. Retrieved from <https://developer.android.com/studio/write/lint>.
- [58] Daniele Galligani, Rigel Gjomemo, VN Venkatakrishnan, and Stefano Zanero. 2014. Static detection and automatic exploitation of intent message vulnerabilities in android applications, 2014. <http://www.ieee-security.org/TC/SPW2015/MoST/papers/s3p1.pdf> (visited: 2020-05-04).
- [59] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Vol. 14. 23–26.
- [60] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1329–1341.

Received April 2019; revised October 2019; accepted December 2019