DETECTING FILE:// AND EXPOSED COMPONENT VULNERABILITIES IN ANDROID APPS

DAOYUAN WU

M.Phil

The Hong Kong Polytechnic University

2015

The Hong Kong Polytechnic University
Department of Computing

Detecting file:// and Exposed Component Vulnerabilities in Android Apps

Daoyuan Wu

A thesis submitted in partial fulfillment of the requirements for the Degree of Master of Philosophy

August 2014

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

Signed: 吴道远

Name of student: Daoyuan WU

Abstract

In only a few years, smartphones have already become indispensable tools for many people to manage their daily lives. However, our privacy and security are constantly threatened by mobile malwares and vulnerable mobile apps. Detecting these malwares and uncovering vulnerable apps is therefore one of the most pressing problems confronting the security research community.

This thesis considers two main security problems in Android platform, the most popular mobile operating system to date. First, we identify four types of attacks in Android browsers, collectively known as FileCross that exploits the vulnerable `file://` interfaces to obtain user's private files, such as cookies, bookmarks, and browsing histories. We design an automated system to dynamically test 115 browser apps collected from Google Play and find that 64 of them being vulnerable to the attacks. They include the popular Firefox, Baidu and Maxthon browsers, and the more application-specific ones, including UC Browser HD for tablet users, Wikipedia Browser, and Kids Safe Browser.

A detailed analysis of these browsers further shows that 26 browsers (23%) expose their browsing interfaces *unintentionally*. In response to our reports, the developers concerned promptly patched their browsers by forbidding `file://` access to private file zones, disabling JavaScript execution in `file://` URLs, or even blocking external `file://` URLs. We employ the same system to validate the ten patches received from the developers and find one still failing to block the vulnerability.

The second problem is related to the fundamental feature of Android—the component-based communication—in which apps can utilize other apps' exported components for flexible coding and data sharing. In return for this convenience, the exported components, if not well designed, will run into serious security risks. In this study, we consider a general class of vulnerabilities occurred in exported components, named *exposed component vulnerability* (ECV), which exposes privileged capabilities or private resources to other unauthorized apps.

To detect these ECVs, the prior works use a set of sinks pertaining to the ECVs under detection. We argue that a more comprehensive and effective approach should start from a systematic selection and classification of *vulnerability-specific sinks* (VSinks). The set of VSinks employed in our study is much larger than those used in the previous works. Based on these VSinks, our sink-driven approach can detect different kinds of ECVs in an app in two steps. First, the VSinks and their categories are identified through a typical forward reachability analysis. Second, based on each VSink's category, a corresponding detection method is used to identify the ECV via a customized backward dataflow analysis. We also design a semi-automated guided analysis and validation for system-only broadcast checking to remove some false positives.

We implement our sink-driven approach in a tool called ECVDetector and evaluate it with the top 1K Android apps. We use ECVDetector to successfully identify a total of 49 vulnerable apps across all four ECV categories we have defined. To our knowledge, most of them are previously undisclosed, such as the very popular Go SMS Pro and Clean Master. Moreover, the performance of ECVDetector is high, requiring only 9.257 seconds on average to process each component.

Acknowledgements

How time flies! It has been three years since I came to study at PolyU. I am lucky, to be able to study at such a convenient campus without other worries. During this period, I meet, know, and get familiar with the following people, who help me grow in many ways. I would like to give my sincere thanks to all of them.

First, I feel so honored to have Prof. Rocky Chang as my chief supervisor. His kindness, optimism, humor, and healthy lifestyle affect me as a person. His critical thinking, professional writing revision, and numerous discussions help revise my papers to become clearer, more focused and convincing. I really appreciate his patience for tolerating my poor English writing, his encouragement and advice for improving my English, and his company before several paper deadlines. I have learnt a lot from his presentation, teaching manners, and the dedication to high-quality research. I am also grateful to his sincere criticism, pointing out my shortcomings in research and character. Hope one day he can be proud of me as an independent researcher. In addition, my great gratitude goes to him for his trust and help during my most difficult time.

Special thanks are given to the two external examiners, Prof. Lucas C. K. Hui and Prof. Wing Cheong Lau, for their careful reviews and critical comments. I would like to also thank them for writing recommendation letters for my PhD application after the thesis defense.

I am very grateful to the past and current members in the Internet Infrastructure and Security Group: Dr. Daniel Xiapu Luo for his mentorship and guidance at my early research stage, Dr. Brent Peng Zhou for his endless research discussion with me in the last year, and Weichao Li, Ricky Mok, and Waiting Fok for their useful discussion at weekly group meetings and kind help for handling various local issues, and Ang Chen, Yujing Liu, Cong Xu, Wei Yu, Jack Chan, Peixin Chen, Curtis Yung, Chenxiong Qian, Chengyan Wang, Lei Xue, and Monica Leung for their friendship.

I would like to also thank the following professors who taught me courses at PolyU: Dr. Julia Chen on her interesting and practical thesis-writing class, Prof. Wei Lou for

his mobile computing class, and Prof. Iris Benzie for teaching me several important principles of research ethics.

My deepest thanks go to my family, especially my parents for their constant love and support. My mom, the most important person of my life, always says that my happiness and healthiness are her biggest wishes. She is the best example for guiding me to become a good and enthusiastic person. My dad is strict with me (when I was a child), his way of expressing his love to me. From him, I learn how to be an aspiring and hard-working man. Special thanks also go to my girlfriend for the happiness, encouragement, and love she gives to me.

Last but not least, I thank the administrative staff in the General Office of Computing and Research Office for helping me with various administrative issues. Thanks also go to the anonymous reviewers from the 2014 Information Security Conference for their professional reviews. This work is supported by two grants (ref. no. ITS/073/12 and GHP/027/11) from the Innovation Technology Fund in Hong Kong.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years the popularity of smartphones has become an integral part of many people's lives. They are changing our life-style in many ways. In particular, smartphone applications or commonly known as *apps* have made significant impacts on our social interactions with others and personal entertainment. For example, we keep in touch with friends through communication apps like Whatsapp and WeChat, play all kinds of mobile games whenever we can, and surf the Internet anywhere via mobile browsers. According to [1], Google Play has in store over 1.3 million apps for the Android platform, the most popular mobile operating system to date. Another popular mobile system, iOS, also attracts 1.2 million apps in its app store [2].

However, at the same time our privacy and security are also under constant threats of mobile malwares and vulnerable apps. Taking Android as an example, Zhou et al. [3] have collected over 1,200 malware samples in the period of August 2010 to October 2011. Further, they found one third of these samples leveraging root exploits to fully compromise users' Android devices. Besides malware, application-level vulnerability is another serious threat to Android. By examining eight phone images, Grace et al. [4] showed that 11 privileged permissions were leaked from vulnerable system apps. Moreover, apps developed by third-party vendors may contain more security mistakes. For example, over thousand apps were found vulnerable to one kind of data leak attacks [5].

In this thesis, we focus on two important security problems in Android platform. First, we show that the broken same-origin policy (which will be elaborated more in the next chapter) on enforcing `file://` requests will cause numerous Android browsers vulnerable. This might not be a big issue in desktop environment, but the risky `file://` support will hamper the security model of mobile systems that each app's private files should not be accessed by other apps. The second problem is on the (in)security of inter-app communication in Android. Since Android is designed with the principle of openness,

an app usually opens its services to other apps. However, such inter-app communication mechanism, if not understood well by developers, may cause victim apps exploitable by other local apps. We explain these two problems in details below.

1.1 Two Security Problems

1.1.1 Risky `file://` Support in Mobile Browsers

Using `file://` to browse local files is very common in desktop browsers. However, this file protocol mechanism, when applied to mobile platforms, could cause unexpected security risks. In modern smartphone systems, notably Android and iOS, each app's sensitive files are stored in their own system-provided private file zones, which cannot be accessed by other apps or users. Supporting `file://` without additional access control in mobile browsers, however, will break such security boundaries. This `file://` vulnerability is further aggravated in Android, because Android browsers usually accept external browsing requests which, in the absence of any user interaction, can be issued by another (malicious) app. Unlike Android, these requests in iOS must be invoked by users' clicking.

Supporting external `file://` browsing requests (or termed as external `file://` URLs) is only a necessary condition for realizing actual attacks. In this thesis, we show that combining with the capability of accessing private file zones through `file://`, JavaScript support, and other browsers' flaws (such as auto-file download), a malicious app in Android can launch four different types of attacks to steal a victim browser's private files (e.g., users' cookies, bookmarks, and browsing histories) or a victim website's private files (e.g., cookie or content). We refer to this class of attacks as *FileCross*, in which all attack vectors are delivered through the `file://` protocol between a browser app and an attack app. The attack app can automatically download a private file to the public SD card for exporting, steal a private file by compromising same-origin policy (SOP [6]) on the "host" level, steal the content of another website by compromising SOP on the protocol level (`file://` and `http(s)://`), and steal a private file by exploiting a SOP flaw in handling symbolic links.

Several isolated incidences on stealing browsers' private files were reported for Chrome and Firefox [7–9]. However, as we will show in this thesis [10], these attacks are just instances of the FileCross attacks. To characterize the prevalence and impact of the FileCross attacks, we develop a system based on dynamic analysis to automatically test over 100 browser apps in Android. The main approach is to mimic actual attacks and use them to test the browsers on real smartphones. This system determines whether a

browser app is vulnerable to the four FileCross attacks. It also analyzes whether the app, before and after patching, supports `file://`, allows access to private file zones through `file://`, and supports JavaScript.

1.1.2 Insecurely Exposed Components in Android Apps

To ease and accelerate the app development, Android takes a modular programming paradigm that empowers developers to focus on essential building blocks (i.e., components [11] in Android terminology). Moreover, Android apps could expose their components for cooperating with other apps. For example, both Facebook and Twitter apps leverage an existing photo-capturing component exposed by a camera app by simply sending a request to it. The photo-capturing component in this case is an *exposed component* that serves external requests from other apps.

In return for this convenience, exposed components, if not well designed, might run into security risks. In fact, vulnerabilities might exist if a dangerous API inside exposed components can be triggered by other (malicious) apps. We refer to this class of vulnerabilities as *exposed component vulnerability* (ECV), and the dangerous APIs in ECVs are the sinks of potential attack flows. Usually, ECVs could be exploited by an attacker to perform dangerous operations by simply sending crafted inputs from a regular app to a victim app, both installed on the same phone.

Several methods for detecting specific ECVs [4, 5, 12–15] have been proposed in the past. In all of these works, detecting these ECVs use a set of sinks pertaining to the ECVs under detection. Specifically, Woodpecker [4], DroidChecker [12], and the very recent IntentFuzzer [15] are designed to detect *permission leakage* in Android apps, and they focus on a specific kind of sinks that would directly leak permissions once victim apps are exploited. A recent work, SEFA [14], further considers some database-related sinks that are aimed by ContentScope [5] for a special kind of components (i.e., Content Provider). Finally, CHEX [13] discovers potential vulnerabilities related to another kind of sinks, which are the data sinks that might cause unauthorized read or write operations on sensitive resources.

In this study [16], we argue that a more comprehensive and effective approach should start from a systematic selection and classification of sinks. Note that in the context of ECV detection, sinks should be vulnerability-specific (i.e., vulnerability-specific sinks, or VSinks) in contrast to the general data sinks for privacy leak detection [17]. This approach will help resolve two major issues in the previous detection methods. First, the set of sinks obtained from our approach is much more comprehensive. It will therefore help the previous methods to discover new ECVs. Another and also more important issue

is that the prior methods are tightly coupled with individual analysis requirements of their selected sinks. They therefore cannot collaborate with one another to form a more general detection method. Our approach, on the other hand, breaks this coupling by admitting different kinds of sinks and categorizing them for different analysis methods.

Using this sink-driven approach, we adopt a systematic strategy to select VSinks and classify them into multiple categories according to their different analysis requirements. In this strategy, we combine multiple metrics (e.g., permission semantics and API names) to systematically define rules. These rules are made according to a simple, but practical, rule syntax. We further write a rule interpreter to automatically select and classify VSinks according to the defined rules.

Based on the categorized VSinks, our sink-driven approach can detect different kinds of ECVs in an app in two steps. First, VSinks and their categories are identified through a typical forward reachability analysis. We employ an iterative intra-procedural algorithm with flow sensitivity to perform this reachability analysis. Second, based on each VSink's category, a corresponding detection method is used to identify the ECV via a customized backward dataflow analysis. The backward, instead of prior forward, dataflow analysis is chosen to adapt to more categories of sinks. Furthermore, we design a semi-automated guided analysis and validation capability for system-only broadcast checking for removing some false positives.

1.2 Contributions

In this thesis, we make the following three major contributions:

First, we show from our automated testing that the `file://` vulnerabilities in Android browsers are much more prevalent and damaging than previously thought. We summarize our main findings obtained from our analysis of 115 browser apps below.

1. More than half of the browsers tested are vulnerable to the FileCross attacks. In particular, 50% of the most popular browsers (e.g., Firefox, Baidu, and Maxthon) are also vulnerable. Similarly, many major browsers in different categories could leak out private information through the FileCross attacks. Among the four different attacks, the three attacks that are based on compromising SOP affect 55% of the browsers on Android 4.0, 4.3 or 4.4.

2. The `file://` vulnerabilities are exploitable in all Android versions (including the latest 4.4), and even occur in different web engines. Specifically, our system identifies 46 browsers being vulnerable in 4.4 (across all four FileCross attacks). This

result contradicts the general belief that Chrome-based new system engine will no longer contain these flaws by default. We are also contacting Google Android security team to fix one common flaw at the engine level. Moreover, we detect three vulnerable browsers (Firefox, UC Browser HD and Sogou) out of 15 browsers that employ custom engines.

3. A further analysis reveals that 23% of the browsers expose their browsing interfaces unintentionally. Had the developers realized the browser interfaces' exposure, one third of them will not be vulnerable to the FileCross attacks. Moreover, 65% of the browsers accept external `file://` browsing requests, and 62% even allow `file://` access to the private file zones. The latter is necessary for three FileCross attacks. Moreover, 63% support JavaScript execution in `file://` URLs which makes three FileCross attacks possible.

4. In response to our vulnerability reports, 19 developers followed up with our findings. We have so far received nine patches from them (and will receive more). An analysis of the patches shows that the patching methods include disabling the access to unrenderable private files, blocking external `file://` URLs, or disabling JavaScript execution in `file://` URLs. Most of them could effectively thwart the attacks. However, our system developed for testing browsers finds that one patch failed to block the vulnerability, because the patch missed a second attack entry.

Second, we propose a new sink-driven approach to systematically tackling the ECV detection problem. This approach includes a systematic strategy for VSink selection and classification, a general detection method to identify all categories of potential ECVs, and a semi-automated guided analysis for excluding some sink-specific false positives. Moreover, we implement a tool and conduct experiments with real apps, with the details below.

- We implement our sink-driven approach in a tool called ECVDetector. We also design three analysis enhancements in ECVDetector, and the major one is that ECVDetector can validate broadcast checking, a capability that could significantly reduce false positives. Moreover, ECVDetector identifies a total of 372 VSinks across four categories, as well as 183 data source APIs. We have released this dataset at https://github.com/daoyuan14/VSinkDataset.

- We evaluate ECVDetector with the top 1K Android apps from Google Play. In total, we identify 49 vulnerable apps across all the four ECV categories. To our knowledge, most of them are previously undisclosed, such as the very popular Go SMS Pro and Clean Master. Moreover, the performance of ECVDetector is high, requiring only 9.257 seconds on average to process each component.

Third, our works on detecting Android vulnerabilities are making real social impact. Specifically, we have identified vulnerabilities in very popular apps, such as those from Firefox, Baidu, Tencent, and Cheetah Mobile. In particular, three apps with over or close to 100 million installs were identified by us as vulnerable. They are Clean Master (over 200 million), GO SMS Pro (over 75 million), and Firefox (over 50 million). Besides discovering the security weaknesses, we also ethically reported the identified vulnerabilities to their vendors, and help developers to fix their vulnerabilities and test their patches. These efforts make us receive the nice acknowledgements from those developers. In acknowledging our efforts of improving the security of the apps, Baidu sent us two bug bounty gifts. One of our reports were even elected as the most valued vulnerability report of Baidu (http://sec.baidu.com/index.php?announce/detail/26).

Chapter 2

Background

In this chapter, we review some background knowledge that is required for understanding the rest of this thesis. First, we introduce the basics of Android security, which is necessary for understanding both problems. We then go through the basic browser security knowledge that is related to our first study on `file://` vulnerabilities. Further, we briefly describe some basic program analysis concepts to help readers better understand the proposed sink-driven approach in our second study.

2.1 Android Security Basics

2.1.1 Sandbox-based App Isolation

One fundamental security mechanism of Android is its sandbox-based app isolation. Each app is resided in its own sandbox with the internal data and code execution isolated from other apps. Such isolation is enforced at the kernel level. Thus all third-party apps, system apps, application framework, and operating system libraries are all contained within each sandbox. Specifically, Android uses the mature Linux user-based protection to isolate each app. That is, each app is treated as an independent user, and run in a separated process with a unique user ID (i.e., `uid`).

The underlying app isolation enables two important security features in Android. The first feature is *data separation*. Each app's sensitive files are stored in their own system-provided private file zones, which cannot be accessed by other apps or users. The private file zones in Android have the following naming convention: suppose an app's package name is `com.example.app`, then its private file zone is located at `/data/data/com.example.app`. Knowing this convention is necessary for launching our proposed FileCross attacks.

The second feature is *permission-based privilege control*. In Android, all privileges are granted based on permissions. For example, the privilege of making a phone call is authorized according to the permission of `CALL_PHONE`. Each app needs to claim what permissions it will use during the installation. These granted permissions cannot be changed in the whole life cycle of apps. The sandbox isolation ensures no extra permissions can be obtained from system or other apps.

However, the aforementioned two security features might be broken when an app contains the following two insecure practices: (1) risky `file://` support is enabled in exposed browsing interfaces, and (2) the privileged operations are invokable from exported components. Both two are related to the concept of exposed Android components.

2.1.2 Exposed Components in Android

Components are the essential building blocks of Android applications, and each app is composed of one or more components. Different from traditional C or Java programs that one program has only one main entry point, components in Android apps have their own entry points and can be activated individually. Therefore, each component in an app can perform a logically independent task itself, independent of other components in the app. In Android, there are four types of components:

- An *Activity* component represents a single user screen and is the only component type that has user interfaces.
- A *Service* component performs background operations, usually running continuously in the background.
- A *Broadcast Receiver* component monitors and responds to system-wide broadcasts.
- A *Content Provider* component manages a structured data set, and provides SQL-like interfaces.

To facilitate rapid coding and data sharing among apps, Android allows an app to export its components to other apps. Such components are called *exported components* in Android's terminology. Meantime, reliable permissions can be defined to protect these exported components. For example, an app restricts its exported components can be only accessed by apps from the same developer. In our threat model, we therefore consider *exposed components* as those that are fully exposed to other normal apps. It is worth noting that such threat model is also adopted in previous works [4, 5]. Specifically, we detect exposed components according to the following rule:

RULE 1 (exposed component determination). A component is considered as an exposed component if it satisfies both conditions:
C1: It must be enabled so that it could be successfully instantiated by the system; AND
C2: It must be explicitly or implicitly exported so that other app could access it without permission or only with `normal` *level permission.*

Each Android app contains an `AndroidManifest.xml` file, which defines a set of component attributes. Therefore, the rule for the exposed component determination is to inspect corresponding attributes. Using *C1*, we can exclude those useless components (i.e., those who set `enabled` attribute as false), such as the already deprecated components. For *C2*, the explicitly exported component are those with their `exported` attribute set to true. The implicitly exported components are by default exported by Android convention. For example, Intent-based components with `intent-filter` tag and Content Provider components without `exported` attribute are implicitly exported (Note that this default Provider convention is disabled since Android 4.2, but all other lower Android versions have to be compatible with this convention.).

2.2 Browser Security Basics

2.2.1 Same Origin Policy

Same-origin policy (SOP [6]) is a standard security mechanism implemented in modern browsers. It guarantees a malicious origin (or domain and website) cannot access or manipulate the contents originated from another origin. This policy is important for browsers to securely render contents which are from different origins. Fail to enforce the policy appropriately will result in serious security consequences, such as the so-called universal cross-site scripting (UXSS).

The SOP is enforced according to the consistency of a combination of scheme, domain, and port number. Two URIs can access each other's web resources only if all their three SOP ingredients are the same. Under such SOP guideline, a given URI "`http://www.example.com/dir/page.html`" can be accessed by another URI "`http://www.example.com/dir2/other.html`", because they share the same scheme ("http://"), domain ("www.example.com"), and port number (80). In contrast, the URI "`http://en.example.com/dir/other.html`" will be prohibited by SOP to access the given URI, since the domain part ("en.example.com") is different from the given one.

The specific SOP enforcement, however, also relies on the detailed browser implementation. According to a previous survey study [18], browsers' implementations on SOP

enforcement can be surprisingly error-prone. Indeed, our first study also shows SOP violation on handling `file://` requests is prevalent in Android browsers.

2.2.2 DOM and XMLHttpRequest

The Document Object Model (DOM) is a language-independent interface for accessing and interacting with contents in HTML and XML documents. In the view of DOM, every piece of documents is an object, on which can be operated through language like Javascript. For example, `document.URL` represents the URL address of a document, while `document.cookie` is the object of a document's cookie. A complete description of the DOM objects is available from [19].

XMLHttpRequest is a special Javascript object, which can be used to exchange data with a server without performing a full page refresh. XMLHttpRequest is widely used in AJAX programming. In our FileCross attacks, we use XMLHttpRequest to send asynchronous requests after the original attack HTML files are loaded. Such asynchronous operations enables our attack HTML files to steal contents of target origins and sensitive files. For example, the target contents are retrieved in the `responseText` field of an XMLHttpRequest object. Interested readers can refer to [20] for more details.

2.3 Program Analysis Basics

Several basic program analysis concepts are explained below. For further resources, we recommend Professor Aldrich's 15-819M course [21] and the book of *Data Flow Analysis Theory and Practice* [22].

- Control flow graph (CFG) is a graph to represent program execution flows as paths, in which a node is a basic block of program instructions, and an edge is the execution relationship between two blocks.
- Data flow graph (DFG) is a graph to reflect the propagation of variable values. Data dependency graph is one kind of DFG.
- Static single assignment (SSA) is one form to represent program instructions. It is also one property of an intermediate representation (IR). In SSA form, each variable is defined before it is used, and assigned only once.
- Intra-procedural analysis is a light-weight program analysis technique that only considers instructions within a function procedure and ignores the cross-function analysis.

- Inter-procedural analysis is a program analysis technique that considers the cross-function analysis. It usually constructs call graph to facilitate the analysis.
- Context-sensitive analysis is a property of program analysis that considers different contexts under inter-procedural analysis.
- Path-sensitive analysis is a property of program analysis that considers different path conditions along analysis execution.

Chapter 3

Analyzing Browser Apps for `file://` Vulnerabilities

3.1 The `file://` Vulnerabilities

3.1.1 The FileCross Attacks

We have discovered from our evaluation, which will be further elaborated in Section 3.1.2, that 113 out of 115 browsers in Android expose their browsing interfaces, and 75 out of the 113 browsers support external browsing requests from other apps through `file://`. As illustrated in Figure 3.1, an attack app can issue a "malicious" browsing request to a victim browser through the `file://` channel. The attack can steal sensitive files directly or indirectly from the victim browser's private file zone by having the URL in the browsing request point to a target sensitive file or a malicious HTML file, respectively.

The direct method exploits the fact that some browsers allow `file://` requests to access their private file zones. The indirect method, on the other hand, exploits the same-origin policy (SOP [6]) flaws in handling `file://` requests, and it also requires the JavaScript support for executing the malicious HTML file. In our evaluation, 71 browser apps (out of the 75 that support `file://`) allow the requests received from `file://` to access their private file zones, and 72 permit JavaScript execution in `file://` URLs. Moreover, the indirect method can be used to steal sensitive files from websites.

Figure 3.1 shows examples of four FileCross attack patterns. The first one uses the direct method, whereas the last three use the indirect method by compromising the SOP. The first and fourth attacks are in fact first reported by an individual hacker. We discovered the other two from the Android developer document. We thus do not claim the discovery

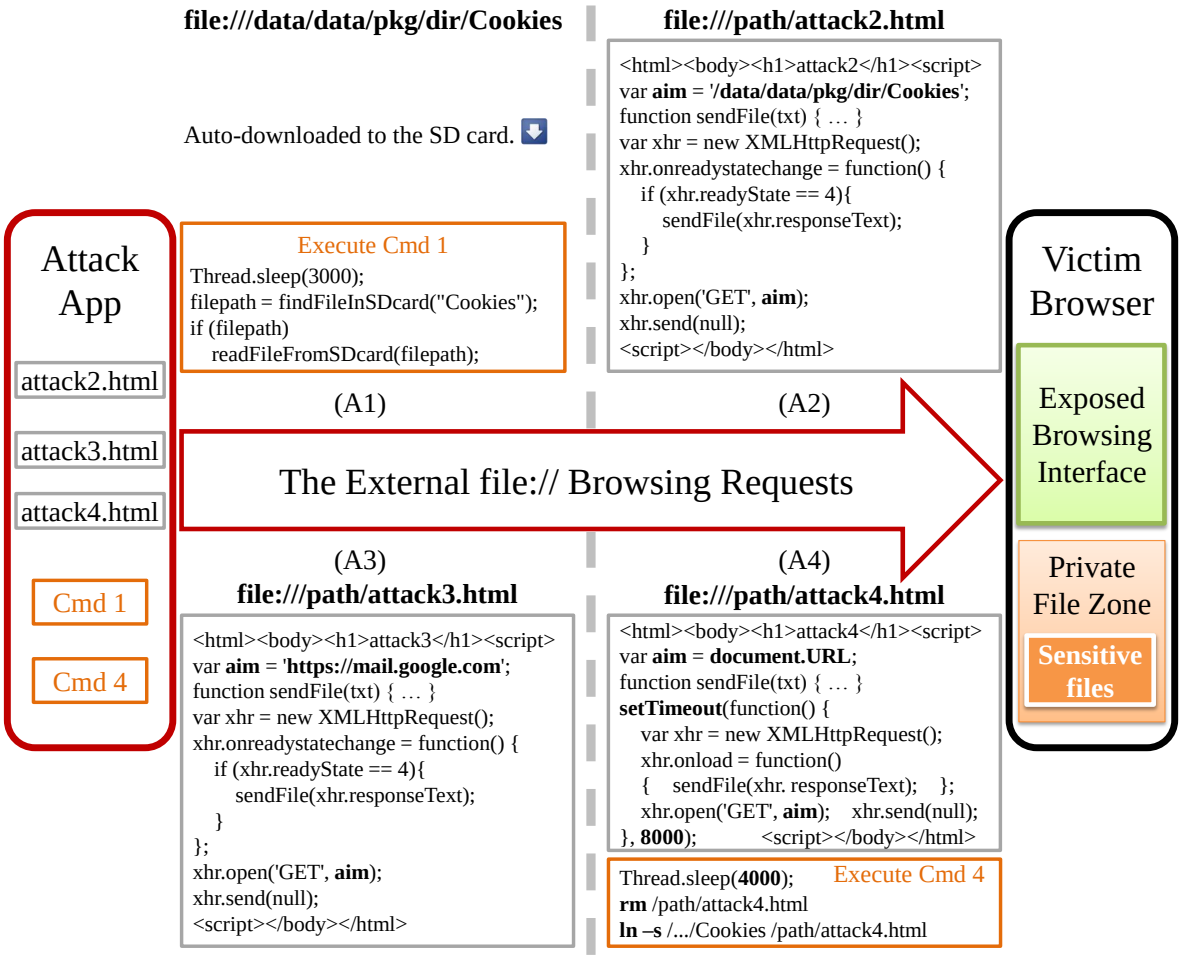


FIGURE 3.1: Examples of four FileCross attacks (A1 to A4).

of these attacks as our main contribution. But we are the first to identify them as a unified attack model (i.e., FileCross) and conduct automated testing to analyze their prevalence in Android browsers. In addition, our system presented in the next section could be extended to discover new attack patterns.

Attack 1 (A1): The `file://` URL points to a sensitive file (`Cookies` in the figure) in the victim browser's private file zone. Some browsers automatically download the requested file to the `Download` directory on a SD card. The attack app can use keyword search to find and read the target file from the SD card (see `Cmd 1`). The auto-download feature has been identified as a flaw responsible for a successful FileCross attack against Chrome for Android [7].

Attack 2 (A2): The `file://` URL points to a malicious HTML file `attack2.html`. The attacker prepares the HTML file for the browser to retrieve a sensitive file (`Cookies` in the figure) from its private file zone. Once the attack HTML file is loaded, an

asynchronous request (e.g., via the XMLHttpRequest API [20]) is issued to retrieve the sensitive file (`xhr.responseText` in the figure). After this, `sendFile(txt)` is invoked to send the file to a remote server that can be accessed by the attacker. The fundamental problem enabling this attack is compromising SOP for `file://` requests (i.e., a local file should not be allowed to read contents of another file). Our evaluation shows that 63 browsers are vulnerable to this attack.

Attack 3 (A3): The `file://` URL points to a malicious HTML file `attack3.html`. The attacker prepares the HTML file for the browser to retrieve sensitive content from a remote website (`mail.google.com` in the figure). Similar to the last attack, the content is retrieved by an asynchronous request and sent to a remote server via `sendFile(txt)`. The fundamental problem is again compromising SOP, but this time on the protocol level (`file://` and `https`). Our evaluation uncovers 56 vulnerable browsers. This attack can also steal cookies of a website, but the details are omitted here.

Attack 4 (A4): The `file://` URL points to a malicious HTML file `attack4.html`. While the objective of this attack is the same as A2, it sets the target (in the `aim` JavaScript variable) as the current URL (i.e., `document.URL` in the figure), thus not violating SOP. However, the codes will not be executed until after 8000 ms. The attack app in the meantime removes `attack4.html` and builds a symbolic link for the removed file using the target sensitive file `Cookies`. Now when the time comes for the browser to execute the codes, it may load `Cookies` according to the link and return its contents to JavaScript. This flaw of loading a symbolic link to a file when the file cannot be found exists in modern browsers, including Chrome [8] and Firefox [9]. Our evaluation reveals 57 vulnerable browsers.

The last three attacks exploit the flaws on enforcing SOP for external `file://` requests. For *webkit*, Android's default web engine, the SDKs prior to 4.1 suffered from flawed SOP enforcement. Although the flaws in attacks A2 and A3 have been fixed by the default setting introduced to Android 4.1, the `file://` vulnerabilities still remain for two reasons. First, we notice that the two new APIs introduced in 4.1 still suffer from the SOP flaws. Therefore, developers may still use these vulnerable APIs, especially when they cannot find the security implications from Google's Developer Document. Second, developers must compile their apps using recent SDKs to block the vulnerabilities. Our evaluation, however, shows that over 30 browsers on Android 4.3 are still vulnerable, because the developers still used the old SDKs to compile their apps.

Starting from the latest Android 4.4, the system web engine is changed to Chrome's Blink engine. A general belief is that Chrome-based engine will no longer contain these

flaws by default (we even made this mistake earlier via preliminary manual testing, since file paths are changed in 4.4). But surprisingly, our automated testing finds 46 browsers are still vulnerable in 4.4, across all four FileCross attacks. In particular, we notice Android 4.4 does not provide by-default patches for the SOP flaw (in A4), causing 40 browsers still exploitable in 4.4 by attack A4. We are contacting Google Android security team to fix this common flaw at the engine level. Moreover, similar to the Android 4.3 cases, apps compiled with old SDKs (i.e., below 4.1) cannot be protected by system-level defenses for attacks A2 and A3, even running on Android 4.4. Additionally, the flaw in A1 is application specific. In summary, mitigating the FileCross flaws in all Android versions still require browser developers' careful implementations. Therefore, our evaluation system to be presented in Section 3.2 is designed to test browser implementations but not specific web engines.

3.1.2 Attack Conditions

Table 3.1 summarizes the conditions required for launching the four FileCross attacks. Exposing browsing interfaces and supporting `file://` are obviously necessary for all of them. Allowing `file://` access to private file zones is also necessary for major FileCross attacks that aim at stealing browsers' private files. In addition, attacks A2, A3, and A4 require JavaScript execution in `file://` URLs for constructing the corresponding exploits (as shown in Figure 3.1). Although it is always possible for some advanced attackers to invent non-JavaScript exploits for these three attacks, we believe this JavaScript condition is currently required and therefore include it into our FileCross threat models.

TABLE 3.1: The required conditions for the four FileCross attacks.

Attack IDs	Required Attack Conditions				
	Exposed browsing interface	Support `file://` URLs	`file://` access to private file zones	JavaScript execution in `file://` URLs	Major flaws
A1	√	√	√		Auto-download file to SD card
A2	√	√	√	√	SOP bypass for two `file://` origins
A3	√	√		√	SOP bypass for `file://` and `http(s)://` origins
A4	√	√	√	√	SOP bypass in handling symbolic links

Before moving to the next section, it is instructive to understand how browsing interfaces are exposed. As mentioned above, 113 of our tested 115 browsers expose their browsing interfaces to other apps. By inspecting their manifest files, we further infer that some browsers expose their browsing interfaces *unintentionally*, although most express *explicit* intentions to accept external browsing requests. We summarize these intentionally and unintentionally exposed patterns in Figure 4.8, and also give a simple Exposed Browsing Interface (EBI) example in Figure 3.2(b). Our inference for intentional exposures is

based on the presence of an Intent with the `action` of "VIEW" and the `category` of "BROWSABLE," because this type of Intent is usually delivered to browsers [23].

EBI Category	Major Related Attributes	
Intentionally exposed	intent-filter	action: "android.intent.action.VIEW"
		category: "android.intent.category.BROWSABLE"
		data <android:scheme>: "https", "http", "file", …
	android:exported="true"	
Unintentionally exposed	intent-filter	action: "android.intent.action.MAIN"
		category: "android.intent.category.LAUNCHER"

(a) Intentionally or unintentionally exposed browsing interface and their related attributes.

```
<activity android:name="it.nikodroid.offline.ViewLink" …>
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:mimeType="text/*" />
  </intent-filter>
</activity>
```

(b) The intentionally exposed browsing interface (`.ViewLink`) in Offline Browser (`it.nikodroid.offline`).

FIGURE 3.2: A summary of EBI (Exposed Browsing Interface) patterns and an EBI example.

The unintentionally exposed cases, in our understanding, are mainly caused by the Android's implicit Intent mechanism [24]. Specifically, Android requires each app to register an Intent filter with the `action` of "MAIN" and the `category` of "LAUNCHER" for the first user interface component, so that the app can be launched by the default launcher. This behavior, however, will implicitly cause the corresponding component to be exposed to other apps. It may happen for some browser developers to register their browsing interfaces with such Intent, thus exposing them as EBIs even without claiming to receive "BROWSABLE" intents. Hence, these EBIs cannot be triggered by normal browsing requests. We thus believe they are unintentionally exposed by developers in terms of serving external browsing requests. Due to the limited pages, we refer readers to Section 5.4 of [25] for a general discussion on such implicit intents.

3.2 Automated Testing of Android Browsers

We design and implement a system for testing browsers for the `file://` vulnerabilities. In order to test all browser apps available in Android markets, our system can automatically test all of them without human intervention. Using the system, we could test over 100 Android browsers in less than four hours. Since our ultimate goal is to report vulnerable browsers to their developers for patching, it is not enough to just demonstrate that private files can be accessed by invoking JavaScript's `alert(content)` function. Instead, our system mimics the actual attacks to "steal" victim browsers' private files and tests the browsers on actual smartphones. Besides detecting the vulnerabilities, the system also helps determine whether the external browsing interfaces are open intentionally and analyze the patches obtained from the developers.

3.2.1 The System Design

Figure 3.3 shows the architecture and workflow of our testing system. The three main components in this system are *Commander* for controlling the entire testing process, *Attack Executer* for launching the FileCross attacks, and *Web Receiver* for validating whether the attacks are successful. The Commander running in a PC host controls the connected Android devices (which can be emulators or real phones) via Android Debug Bridge (ADB) channels (from ADB host to ADB daemons on devices). We implement Commander in pure Python language for avoiding the instable issues of MonkeyRunner [26] reported in [27]. Moreover, we implement parts of the failure controlling mechanisms proposed in [27] to improve the stability of ADB over long runtimes and use multiple threads to concurrently control each device for testing multiple Android versions in parallel.

We implement the Attack Executer as an Android app and install it in each tested device. Like a real attack app, it launches the FileCross attacks to steal private files from the target browsers. Moreover, its attack behaviors are fully controlled by the Commander through each incoming attack command (including target browser information and attack parameters). Once receiving the attack commands, it generates the corresponding exploits on-the-fly and loads them into target browsers via the Intent channels. The Web Receiver, on the other hand, is a server-side program responsible for accepting the stolen private files and validating the attack results. An attack is considered successful if the stolen file is received.

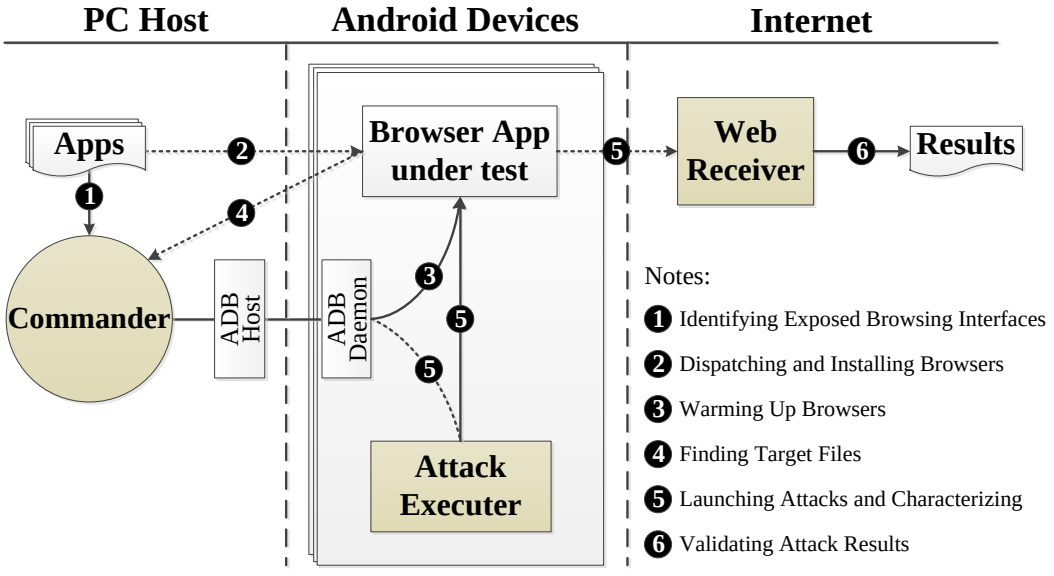


FIGURE 3.3: The architecture and workflow of our testing system.

3.2.2 The Major Testing Steps

Figure 3.3 shows six major testing steps in our system. We discuss them below in three pairs.

Identifying exposed browsing interfaces We propose a lightweight but effective scoring mechanism to identify EBIs in Android browsers. The basic idea is to score each component based on our summarized EBI patterns in Section 3.1.2 and select the component with a maximal score as the EBI. That is, a component with the maximal score is most likely to be an EBI. This maximal score also helps us locate the major (or true) browsing interface. For instance, Chrome's `ManageBookmarkActivity` exhibits EBI patterns but is not functional for handling browsing requests. In this case, our scoring mechanism can help identify the right browsing interface `chrome.Main`, which shows more explicit EBI patterns, thus a higher score. When several EBIs have the same score, we handle such case by randomly selecting one EBI for dynamic testing. In addition, if all components score zero, we conclude no EBI in the browser. In our experiments, we find that this scoring mechanism can accurately identify the EBIs in 113 browsers out of the tested 116 browsers. For the remaining three cases that have no EBIs, one of them is only a browser add-on, and the other two do not expose their browsing interfaces.

The detailed scoring algorithm works as follows. We use six bits to flag five specific EBI patterns (two bits are set for one pattern under different situations). Figure 3.4 illustrates the detailed rules for scoring the EBI patterns under different scenarios. For example, if one component has an Intent filter which defines the `action` of "VIEW" and the `category` of "BROWSABLE," we set bit 2 (i.e., a score of 4). If this Intent filter also registers the `data` scheme of "http," we further set bit 3 (i.e., a score of 8). Now the component has a total score of 12, which can be used also for reversely inferring the EBI patterns using its binary representation.

Bit Id	5	4	3	2	1	0
0/1	1	1	1	1	1	1
Score	32	16	8	4	2	1

Bit Id	**0**	**1**	**2**	**3**	**4**	**5**
EBI pattern	MAIN LAUNCHER	file	VIEW BROWSABLE	http	https	MAIN LAUNCHER
Pre condition	Bits (1 – 4) are all empty	Bit 2 is set	_	Bit 2 is set	Bit 2 is set	One of bits (1 – 4) is set

FIGURE 3.4: The detailed rules for scoring EBI patterns using six bits.

These scoring rules (with different weights) are summarized according to our manual analysis of a dozen of EBI patterns. First, we treat the basic EBI pattern (i.e., "VIEW" and "BROWSABLE") as a reference pattern. On the basis of this pattern, we further

assign weights to three data schemes, if any. Among them, we score the "https" scheme higher than "http," because we find accepting "https" is more likely to represent an EBI. On the other hand, we lower the "file" scheme even below the reference pattern, for removing the potential noises introduced by "file". The noises can occur when "file" is registered for browsing document or video files. So such components are actually document viewers or video players, instead of browser components. Finally, we observe the "LAUNCHER" pattern, if exists, can add more weights when the aforementioned patterns also occur. That is, a component with both "BROWSABLE" and "LAUNCHER" patterns will be always the major EBI, compared with those non-launcher "BROWSABLE" components. In addition, a component with only the "LAUNCHER" pattern should be scored less than other "BROWSABLE" components.

Warming up browsers and finding target sensitive files The goal of warming up browsers is to produce some private files as the target sensitive files. To do so, the system automatically sends several normal browsing requests before launching the attacks. Specifically, the tested browsers are instructed to browse several Alexa top 10 websites using HTTP or HTTPS. This warming-up step can also help validate the EBIs identified by the scoring mechanism. That is, if an EBI is correctly identified, we can effectively warm up the corresponding browser. Otherwise, the browser will not respond according to our external browsing requests.

After warming up the browsers, our system continues to find as many target sensitive files as possible from the newly generated private files. To do so, the system searches browsers' private file zones (i.e., `/data/data/package/`) using a set of prioritized keywords (e.g., "cookie", "password", and "bookmark") and certain file formats (e.g., ".sqlite" and ".db" files). Note that accessing private file zones, which is normally disabled on unrooted phones, is only used for finding target sensitive files in our system (and attackers can also use this method to obtain the same information for their attacks). The actual FileCross exploitation is still conducted by the Attack Executer through the normal Intent channels.

Automatic attack validation and characterization Another challenge in designing our system is how to *automatically* validate attack results and conduct further characterization. Unlike manual testing, we cannot rely on human intervention, such as naked-eye inspection. To address this issue, we pre-define patterns that describe the attack details given by the Commander and embed them into each attack request sent by exploit scripts, which will be finally received and interpreted by the Web Receiver. In particular, we embed five patterns into the attack requests: an app package's name (for identifying the tested browser), an attack ID (for differentiating different attacks), a device version (for characterizing attacks on different Android versions), contents (for

transmitting and validating potential private files), and a key ID (for authentication and differentiating different experiments).

To further characterize the FileCross attacks, we adopt the similar methods as for launching attacks, except that the attack scripts are now replaced by other scripts for characterization purposes. Specifically, we design HTML files to characterize the `file://` support (loaded from SD card or private file zones) and JavaScript execution in `file://` URLs. For example, the following HTML file is for characterizing the `file://` support. The Attack Executer loads this HTML file from both SD card and private file zones (with different attack IDs, such as `atk=5`), and sets the current Android version (e.g., `ver=4.3`).

```
<html><body>  <img src='http://ourserver.com/req?pkg=example.package
&atk=5&con=reqflag&ver=4.3&kid=keyid'>  </body></html>
```

As another example, the HTML file for characterizing JavaScript execution in `file://` URLs is given below, which is relatively complex.

```
<html><body>
<script>
var d = document; var img = d.createElement('img');
img.src = 'http://ourserver.com/req?pkg=example.package&atk=7&con=reqflag
&ver=4.3&kid=keyid';
d.body.appendChild(img);
</script>
<noscript>
<img src='http://ourserver.com/req?pkg=example.package&atk=7&con=
&ver=4.3&kid=keyid'>
</noscript>
</body></html>
```

3.3 Evaluation

3.3.1 The Dataset and Experiments

Dataset Our dataset consists of 115 browser apps collected from Google Play on January 21, 2014. Initially, we searched the keyword "Browser" on Google Play and fetched 139 browsers, after excluding several non-browser apps. We further revisited these 139 browsers on March 21 for characterizing their meta information (e.g., the install numbers) using the Selenium scripts [28]. Based on the results, we further excluded 23

browsers in which 14 of them were no longer updated for more than one year, and 9 others had been withdrawn from Google Play. Among the remaining 116 browsers, one more was excluded, because it was only a browser add-on.

Experiments We run our experiments using three Android phones: Sony Xperia J (with Android 4.0), Google Nexus 4 (with Android 4.3), and Nexus 5 (with Android 4.4). These phones are connected to a Dell Studio XPS desktop machine with Ubuntu 12.04 64-bit system through USB cables. We do not use Android emulators in previous studies [27, 29–32], because they are not stable and a number of apps cannot be correctly installed or run on emulators. However, accessing apps' private file zones via ADB on real phones is disabled by default. We thus root the phones to enable it for our automatic testing.

In theory, different handsets running the same image of Android OS will have the same vulnerability result for the same app. However, vendors may custom the systems in their mobile devices. We therefore anticipate there may be vulnerability differences across different handsets. We currently focus on three aforementioned devices, because we do not have more phone resources. We leave testing more Android devices for `file://` vulnerabilities as our future work.

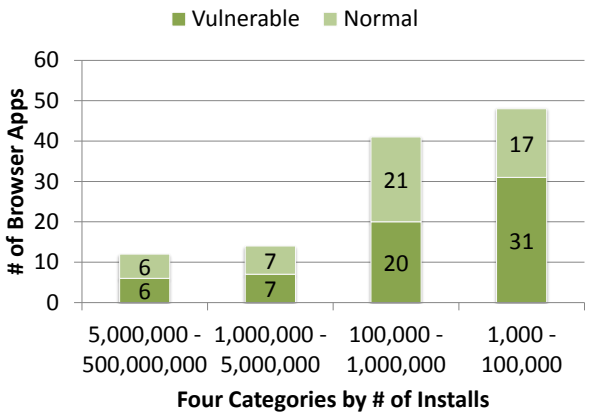
In this section, we report the results obtained from three independent experiment runs conducted on March 27 and June 18 (when the 4.4 device newly joined). Our system incurs no false positives but may incur some false negatives due to the possible instability[1] of dynamic testing. To mitigate this possibility, our final result is a union of the results from these three runs. Regarding the testing performance, each run takes around four hours (i.e., 3 minutes per app). We use a relatively long timeout (12 seconds) before starting a new browsing request to obtain stable results and duplicate the app testing on three phones for observing possible different results in the three major Android versions.

3.3.2 Vulnerability Results

Overall results Our system identifies 64 vulnerable browsers and a total of 177 File-Cross issues, as shown in Figure 3.5(a). The results clearly show that the vulnerabilities are prevalent in Android browsers: 55.7% of browsers are affected and on average 2.77 issues per vulnerable browser. Furthermore, according to their distribution by the number of installs, 13 out of 26 popular browsers with over million installs each are found vulnerable. They are from top browser vendors, including Firefox, Baidu, and Maxthon.

[1] It is worth noting such instability is caused by Android apps' compatibility issues that some recent apps cannot run on old devices. Therefore, this is not our system's limitation.

In other words, the FileCross attacks are not easy to discover and were not known to them before our disclosures.



(a) The distribution of browsers with(out) vulnerabilities.

IDs	# of Browsers	
A1	1	
A2	63	62 (4.0)
		35 (4.3)
		25 (4.4)
A3	56	55 (4.0)
		31 (4.3)
		22 (4.4)
A4	57	57 (4.0)
		49 (4.3)
		40 (4.4)
Sum	177	

(b) Detailed results for each attack.

FIGURE 3.5: Overall detection results in our dataset consisting of 115 Android browsers.

Figure 3.5(b) shows the detailed results for each FileCross attack. In our dataset, we only discover one auto-file download issue, i.e., attack A1. However, we observe that 71 browsers actually load and display the contents of their private files when challenged by attack A1. Therefore, they will face the potential risk of screen-shot attacks, although we do not consider such risk as a vulnerability in this thesis.

For attacks A2, A3 and A4, the number next to (4.0) (or (4.3) and (4.4)) is the number of browsers vulnerable to the attack on Android 4.0 (or 4.3 and 4.4). The number next to these three is the total number of vulnerable browsers for that attack. Some browsers are vulnerable on only one system. These three attacks have a similar number of vulnerable browsers, around 60. Moreover, attack A4 is much less affected by different Android versions than A2 and A3. In the following sections, we thus do not differentiate the results of attack A4 on the three versions. As for attacks A2 and A3, there are over 30 vulnerable browsers for each attack on Android 4.3 and over 20 on Android 4.4, mainly because the developers still use the old SDKs to compile their apps. Thus, their browsers cannot benefit from the webkit patch in Android SDK 4.1.

Representative vulnerable browsers Table 3.2 summarizes 20 representative vulnerable Android browsers identified by our system. To make it simple, we only use the app package name to refer to each browser, and their full app names can be obtained from Google Play. We also include the number of installs for each browser to underscore the scope of the impact. For each vulnerable browser, we list their detailed assessment results of the four FileCross attacks launched by our system. The red "y" means a

successful attack, and the black "n", otherwise. In addition, a blank space represents the case where our attack scripts cannot send response requests to our server, mainly because the target browser is either invulnerable or not stable on some Android versions (e.g., 4.3 and 4.4). For such cases, they are assumed invulnerable if no further manual efforts are involved.

TABLE 3.2: Representative vulnerable Android browser apps identified by our system.

Categories	App Package Names	A1	A2			A3			A4	# of Installs
			4.0	4.3	4.4	4.0	4.3	4.4		
Popular	org.mozilla.firefox	y				n	n	n		50,000,000 - 100,000,000
	com.baidu.browser.inter	n	y		n	y	n	n	y	5,000,000 - 10,000,000
	com.mx.browser	n	y	y	y	y	y	y	y	5,000,000 - 10,000,000
	com.jiubang.browser	n	y	y	y	y	y	y	y	5,000,000 - 10,000,000
	com.tencent.ibibo.mtt	n	y			n			y	1,000,000 - 5,000,000
	com.boatbrowser.free	n	y	y	y	n	n	y	y	1,000,000 - 5,000,000
	com.ninesky.browser	n	y	y	y	y	y	y	y	1,000,000 - 5,000,000
Tablet	com.uc.browser.hd	n	y	y	y	y	y	y	y	1,000,000 - 5,000,000
	com.baidu.browserhd.inter	n	y		n	y	n	n	y	100,000 - 500,000
	com.boatbrowser.tablet	n	y	y	n	n	n	n	y	100,000 - 500,000
Privacy	com.app.downloadmanager	n	y	n	n	y	n	n	y	10,000,000 - 50,000,000
	nu.tommie.inbrowser	n	y	y	y	y	y		y	500,000 - 1,000,000
	com.kiddoware.kidsafebrowser	n	y	n	n	y	n	n	y	50,000 - 100,000
Fast browsing	com.ww4GSpeedUpInternetBrowser	n	y	y		y	y		y	1,000,000 - 5,000,000
	iron.web.jalepano.browser	n	y	y	y	y	y	y	y	500,000 - 1,000,000
	com.wSuperFast3GBrowser	n	y	y		y	y		y	100,000 - 500,000
Specialized	com.appsverse.photon	n	y	y	y	y	y	y	y	5,000,000 - 10,000,000
	com.isaacwaller.wikipedia	n	y	y	y	n	n	n		1,000,000 - 5,000,000
	galaxy.browser.gb.free	n	y	y		y	y		y	100,000 - 500,000
	com.ilegendsoft.mercury	n	y	n	n	y	n	n	y	100,000 - 500,000

We organize these vulnerable browsers into five categories, mainly according to their popularity and unique features. For example, in the "Popular" category, we present several popular browsers with over million installs each. In particular, we identify an auto-file download issue (i.e., attack A1) in Firefox for Android, which is quite popular and has at least 50 million installs. This security issue is ranked by Firefox a high impact one. Moreover, we discover more FileCross issues in other listed popular browsers. For example, Maxthon Browser (`com.mx.browser`) and Next Browser (`com.jiubang.browser`) suffer from three FileCross attacks in all Android versions we tested, which pose significant security threats to their five million users.

The second category ("Tablet") lists three vulnerable browsers built for Android tablets. Except for UC Browser HD (`com.uc.browser.hd`) that has over million installs, these browsers are not as popular as those in the "Popular" category. However, we notice from Google Play that they are essentially the only choices for users who want to install a dedicated tablet browser. This would entice attackers to launch more targeted attacks at tablet users.

In the third category ("Privacy") of Table 3.2, three representative vulnerable browsers built for users' privacy are selected. For example, Downloader & Private Browser (`com.app.downloadmanager`) is a quite popular browser that provides password protection for users' private files. However, other app can access and steal these sensitive files by exploiting this app, contradicting their users' original intention of using this browser to protect their privacy. Similarly, InBrowser (`nu.tommie.inbrowser`) is a browser supports incognito browsing by default. As another important example, the Kids Safe Browser (`com.kiddoware.kidsafebrowser`) that provides children a safe Internet surfing environment by content filtering jeopardizes children's privacy by the FileCross attacks.

Some users prefer the browsers that are optimized to provide fast browsing experience. We summarize three such vulnerable browsers in the category of "Fast browsing". 4G Speed Up Browser (`com.ww4GSpeedUpInternetBrowser`) and 3G Speed Up Browser (`com.wSuperFast3GBrowser`) are vulnerable to all attacks A2, A3, and A4.

In the last category of "Specialized," Photon Flash Player & Browser (`com.appsverse.photon`) and Galaxy Flash Browser (`galaxy.browser.gb.free`) are quite popular due to their dedicated support of Flash player in Android browsers. However, both of them are vulnerable to three kinds of FileCross attacks in all Android versions (except the latter cannot run stably in 4.4). The second case is a dedicated browser for browsing Wikipedia, called Wikidroid (`com.isaacwaller.wikipedia`), allowing users to save their browsing bookmarks. Attackers can launch the FileCross attacks to steal these bookmarks and use them to infer users' interests and profiles. The last case is called Mercury Browser (`com.ilegendsoft.mercury`) which is selected for its popularity in its iOS version. We suspect that some Android users who have migrated from iOS system will possibly install this browser.

3.3.3 Underlying Engine Analysis

It is useful to find out how many browsers do not use the default engine (which has inherent flaws). Implementing a custom web engine in Android usually requires embedding native codes as shared libraries (`.so` files). For example, Chrome uses `libchromeview.so` as its underlying engine to support browsing functionalities. Determining which `.so` files are web engines is hard and also out of our scope. Here, we adopt two tricks to infer which browsers embed their own engines. First, we use regular expression "`native.*loadUrl`" to locate five browsers that implement their own native version of "loadUrl" API, including Chrome, Yandex (`libchromiumkit.so`), Flash Browser (`libxul.so`), and even the vulnerable UC Browser HD (`libWebCore_UC.so`). However, this strategy is not robust

enough, because it even misses the Firefox engine. Therefore, we directly inspect each `.so` file name from 24 browsers which have `.so` files. The inspection (combined with existing knowledge) shows that another six browsers embed their own engines, such as Firefox (`libmozglue.so`), Dolphin Browser (`libdolphinwebcore.so`), and three Opera browsers (`libom.so`).

It is also a trend that more Android browsers will use custom engines. Our analysis of five popular Chinese browser apps (which were collected on May 1) shows that four of them define their own engines. They are QQ (`libmttwebcore.so`), Baidu (`libzeus.so`), Liebao (`libchromeview.z.so`) and Sogou (`libsogouwebcore.so`) browsers. In particular, our system identifies Sogou Browser being vulnerable to FileCross attack A4.

In summary, we have identified 15 (out of the total 120) browsers embedding their custom engines instead of the system default one. In addition, our system identifies three of them being vulnerable: Firefox, UC Browser HD, and Sogou browsers. These findings demonstrate the effectiveness of our system to uncover `file://` vulnerabilities in non-webkit browsers.

3.3.4 Vulnerability Reporting

Our reporting process We have spent considerable efforts on reporting our identified vulnerabilities to the developers. Our reporting process was started on February 7 and is still ongoing at the time of writing. So far we have reported 39 vulnerable browsers, including all the representative ones in Table 3.2. We continue to report the remaining cases and will release a project website to help developers better understand the vulnerabilities.

We report each case mainly in three steps. First, we send a notice email to the email address recorded in Google Play, mentioning that we have identified vulnerabilities in their browsers without details. After receiving their replies and confirming their identities, we explain the FileCross attacks that their browsers are vulnerable to. Finally, if they send us a patch, we analyze it using our system and report to them again whether their patch can correctly block the vulnerability. For the unresponsive developers, we contact them again until receiving their responses.

Responses from developers We have currently received 19 replies from the developers. Among them, 15 gave us further responses after being notified of the vulnerability details, and all of them confirmed our vulnerability reports. In particular, we have received two bug bounty gifts from Baidu company. Some excerpts of developers' responses can be found in Appendix A. Regarding the aforementioned Firefox issue for

attack A1, our discovery of this serious vulnerability was also made independently by another security expert. We were told by Mozilla that "*this flaw has been fixed in the latest version of Firefox for Android, version 28.0.1*" just the day before our reporting [33].

3.4 Further Analysis and Recommendations

3.4.1 Analyzing the Patches

An overview Table 3.3 summarizes the nine patches received so far. Our analysis reveals three kinds of patch methods adopted by the developers. First, similar to the method used by Chrome [8], Firefox's developer disabled the capability of accessing the contents of some unrenderable private files to address the auto-file download issue. However, unlike Chrome, Firefox still allows `file://` access to the private file zone and loading renderable files. We argue that accessing private file zone should be totally banned to mitigate all potential risks. Second, Lightning Browser (`acr.browser.barebones`) and InBrowser (in its beta version, `nu.tommie.inbrowser.beta`) directly blocked the external `file://` URLs from other apps. This fix suggests that supporting external `file://` URLs is not necessary for maintaining some browsers' functionalities. It is interesting to note that the developer of Lightning Browser also applied this method to protect his two other browsers (one is a paid version, and the other an unpublished new browser). Finally, the developers of most patched browsers chose to disable JavaScript execution in `file://` URLs, because it is the easiest way to thwart the three FileCross attacks that require JavaScript support. Although this patch does not eliminate all the possible risks (e.g., screen-shot attacks or origin-crossing attacks without JavaScript), it could be considered effective for the threat models considered in this thesis.

TABLE 3.3: An overview of the nine patches received from the developers.

Package Names	Patched Versions	The Patching Methods
org.mozilla.firefox	28.0.1	Disable accessing unrenderable private files
acr.browser.barebones	3.0.8a	Block external `file://` URLs and alert users
nu.tommie.inbrowser.beta	2.11-55	Block external `file://` URLs
com.baidu.browser.inter	3.1.2.0	Disable JavaScript execution in `file://` URLs
com.jiubang.browser	1.16	Disable JavaScript execution in `file://` URLs
com.baidu.browserhd.inter	1.2.0.1	Disable JavaScript execution in `file://` URLs
easy.browser.classic	1.3.6	Disable JavaScript execution in `file://` URLs
harley.browsers	1.3.2	Disable JavaScript execution in `file://` URLs
com.kiddoware.kidsafebrowser	1.0.4	Disable JavaScript execution in `file://` URLs

An interesting patching process During the process of analyzing the patches, we identified an interesting case which illustrates the importance of automatic testing even

for patches. The developers of Baidu Browser once sent us a version that they thought it was patched, because they had disabled the JavaScript execution. However, our system could still successfully exploit this "patched" version. By a careful manual analysis of the patched version, we have found that there were two rendering points in Baidu Browser's browsing interface: one is invoked when users manually input a URL in the browser bar, and the other is for external browsing Intents. Interestingly, the developers disabled the JavaScript support for `file://` URLs only for the first rendering point, thus leaving the real attack point intact. Since the developers did not have an actual attack app, they tested the "patch" manually and mistakenly thought it was patched.

3.4.2 Exposed Browsing Interfaces

Figure 3.6 shows the breakdown of the EBIs in our tested 115 browsers, of which 113 expose their browsing interfaces, meaning that exposing browsing interfaces is a common practice among Android browsers. However, we notice that 26 browsers (23%) expose their browsing interfaces *unintentionally*. Among them, eight are vulnerable. In other words, these eight browsers could originally avoid the FileCross issues, if they realized to close their unintentionally exposed interfaces.

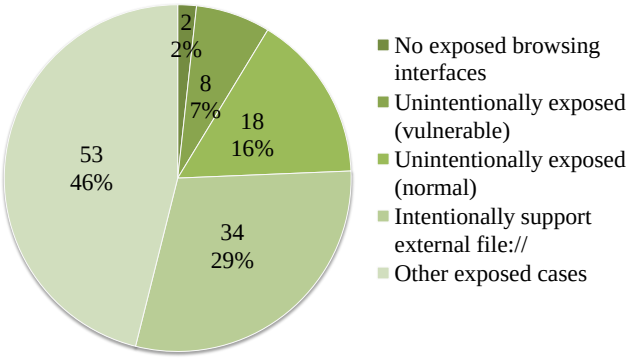


FIGURE 3.6: A breakdown of exposed browsing interfaces in the 115 tested browsers.

We also observe that only 34 browsers (29%) *explicitly* or *intentionally* accept external `file://` browsing requests. But our dynamic testing actually finds 75 browsers supporting external `file://` browsing requests. This discrepancy shows that the other 41 browsers may accidentally leak the `file://` channels to other apps. That is, they intend to support `file://` URLs only for internal uses (e.g., when users manually input a `file://` URL).

3.4.3 `file://` Support in Android Browsers

Based on our analysis, we report three major observations on the `file://` support in Android browsers. First, (at most) 40 of our collected 115 browsers do not support `file://` at all. Note that 40 is only a upper bound, because our system may not successfully characterize some browsers due to the limitation of dynamic analysis. Among the 40 unsupported ones, Opera Mini and UC Browser Mini are the very popular ones. Opera Mini explicitly mentions "*The protocol "file" is not currently supported*" when a `file://` URL is entered, whereas UC Browser Mini redirects users to a Google search page using the keyword of the entered URL. Other unsupported cases that we manually confirm are dedicated browsers, such as The Pirate Bay Browser for browsing torrents and SkyDrive Browser for accessing Microsoft's SkyDrive service. These cases collectively show that `file://` is generally not supported in lightweight and dedicated browsers, and this practice spares them from the FileCross attacks.

Second, we find that several popular browsers already forbids `file://` access to private file zones. Our system identifies four such cases, including Chrome, Dolphin (`mobi.mgeek.TunnyBrowser`), UC (`com.UCMobile.intl`) and Yandex browsers. All of them allow `file://` access to contents in SD card and permit JavaScript execution in `file://` URLs, but forbid `file://` access to their private file zones. Thanks to this security policy, they are robust to most FileCross attacks (i.e., except A2). We therefore recommend to adopt this practice for all Android browsers, because it can better meet the security model of mobile systems.

Finally, we observe three browsers actively disabling the JavaScript execution in `file://` URLs: 3G Browser (`com.mx.browser.free.mx100000004981`) and another two from the same developer (Maxthon Tablet and Maxthon Fast Pioneer browsers). Although the percentage of this practice is currently low (i.e., 3 out of 75), according to our analysis of the patches, we believe that more browsers will follow this practice.

3.5 Discussion

Our system currently focuses on detecting `file://` vulnerabilities in Android browsers. However, the FileCross attacks may also exist in other kinds of apps that use web engine APIs. For example, Facebook was identified vulnerable to attack A2 [34], although it only suffered with another issue called Next Intent [35]. Detecting `file://` vulnerabilities in these non-browser apps is a future work of our system. We plan to incorporate static analysis techniques to identify "similar" browsing interfaces which may not have clear EBI patterns.

The `file://` vulnerabilities have several differences or similarities with the recent mobile origin-crossing work [35]. A major one is that our FileCross attacks focus on stealing local private files, while [35] mainly concerns the sensitive web origin information. Another difference is that FileCross leverages malicious HTML files to launch attacks, while [35] does not use such attack vectors. One similar point is that both our FileCross attacks and one part of [35] attacks need to exploit Intent channels.

There are a few limitations in our current system and experiments. First, some browsers have the splash or welcome views in the front of their browsing interfaces, which may interfere with our automatic attacks. But we also notice several such cases (e.g., Next and Boat browsers) that actually do not affect the effectiveness of our attacks, because the underlying component is still the browsing interface although it is not visible. Second, our current experiments do not cover the default browsers which are pre-installed in devices, because we do not have enough phones to collect and test them.

3.6 Summary

In this chapter, we identified a class of attacks in Android called FileCross that exploits the vulnerable `file://` to obtain user's private files, such as cookies, bookmarks, and browsing histories. We designed and implemented an automatic system to detect the vulnerabilities in 115 browser apps. Our results show that the vulnerabilities are prevalent in Android browsers. More than half of our tested 115 browser apps were found vulnerable. A further detailed analysis gave more insights into the current browser practices, such as exposed browsing interfaces and allowing `file://` access to private file zones. Our vulnerability reports also helped around ten developers to patch their vulnerable browsers promptly. For one browser, our system helped discover that their first patch failed to block the vulnerability.

Chapter 4

A Sink-driven Approach for Exposed Component Vulnerabilities

4.1 Problem Statement

4.1.1 Overview of ECV

Exposed Component Vulnerability, ECV, is an Android version of the classic confused deputy vulnerabilities [36]. The exposed component mechanism in Android allows other apps to send any request or input to the victim component. But the victim component may not differentiate whether a request is from trusted parties, and blindly execute its own code for finishing the request. Consequently, it becomes a confused deputy. Furthermore, an ECV exists when the triggered "deputy" contains security-critical operations. Hence, the victim component is a stepping stone for attack apps, as well as the direct executor of attack behaviors.

Figure 4.1 illustrates a high-level ECV example. Initially, the attack app A and the victim app V are separated by the app boundary. However, since V contains an exposed component, A can send inputs to this exposed component in V. In particular, A can craft inputs (or exploits) to trigger the security-critical operations included in V. Consequently, as shown in the figure, A can leverage V to send SMS messages, although A itself does not own the `SEND_SMS` permission. This kind of ECV is also called *capability leak* or *permission leak* [4]. Moreover, A can send another crafted inputs to trigger database operations in V, such as deleting the private databases of V. This kind of ECV

incurs unauthorized access to private resources, although it does not leak permissions like the previous one.

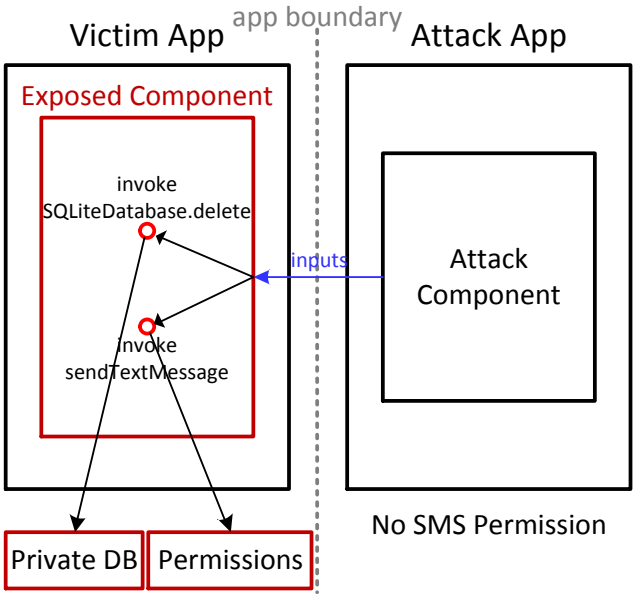


FIGURE 4.1: A high-level ECV example.

Exposed components are commonly used in Android, and not each exposed component will cause vulnerabilities. Many of them are just normal functionalities (e.g., sharing photos from camera), and some are more like bugs (e.g., causing null pointer exceptions). The true ECVs should contain the security-critical operations or APIs that could be unauthorizedly triggered by other apps. This vulnerability model is also adopted by two major related works, [4] and [13]. So security-critical APIs, or typically known as *sinks*, are crucial to ECV detection. We will present in the next section that what kinds of sinks should be covered in the scope of ECVs.

There are some related vulnerabilities with ECVs. A major one is the attack on *unauthorized Intent receipts* [25], which is also originated from insecure IPC communication in Android. However, this issue is mainly due to the ambiguity during the resolution of Intent messages, thus not belongs to ECVs. Other related issues include the *implicit capability leak* [4] via insecure sharing of `sharedUserId`, or in the general, the *colluding attack* [37]. But they either assume other vulnerable points (e.g., insecure signature leaks), or suppose the victim app itself intends to be vulnerable, or even both. We thus do not consider these issues as ECVs.

4.1.2 VSink and its Taxonomy

We use *VSink*, short for Vulnerability-specific Sink, to term the sinks that are in the scope of ECVs. Inspired by two previous works [4, 13], we consider VSinks mainly from the following two sources:
C1: APIs that will violate the security of permission-protected or privileged resources; OR
C2: APIs that will cause security risks to the private resources of apps.

An insight is that we find VSinks are diversified in terms of analysis requirements. In this work, we classify them into four coarse-grained categories. They are effective to illustrate our sink-driven approach in the context of ECV detection. More fine-grained classification can be conducted in a similar manner, if necessary. We now introduce these four kinds of VSinks as follows.

`VS_Direct`. This category of VSinks is usually related to privileged resources, and they can be directly used to launch an attack. For instance, the `removeAccount` is a `VS_Direct` sink. Another example is `SmsManager.sendTextMessage()`, which can send a SMS message to outside without the attention of user. Analyzing `VS_Direct` sinks is relatively straightforward, usually based on a path reachability analysis. A major difference between VSinks and sinks in other Android works is that not all permission-required APIs will be considered as VSinks, only the vulnerability-related ones. For example, `WAKE_LOCK` and `VIBRATE` APIs are excluded.

`VS_DirectByParam`. This category of VSinks is similar to `VS_Direct`, but they rely on the incoming parameters to exhibit different attack behaviors. Different parameters could cause different attack consequences. For instance, `ContentResolver.delete(Uri)` could exhibit different attack behaviors with different URI parameters, such as "content://sms" for deleting SMS. Analyzing such ECVs is more complex than `VS_Direct`, because a chain of variable dependences must be investigated, so that we can obtain the value of parameters. Note that this category of VSinks has not *generally* been considered in prior works (e.g., CHEX [13] only considers some similar APIs when external Intent can control their parameters, while such relationship between Intent and parameters in our `VS_DirectByParam` is not necessarily required).

`VS_Input`. This category of VSinks mainly fulfills the goal of misusing privileged resources, and this kind of misuse relies on attack inputs to flow into `VS_Input` sinks. Network-related sinks are the typical examples of `VS_Input`, such as `HttpClient.execute()`. Once attack inputs flow into them, they could be exploited to misuse protected Internet resources. Besides network-related `VS_Input`, APIs, such as `startService()` and

`startActivity()`, also belong to this category. Note that although `VS_Input` sinks usually contain parameters, this is not necessarily required. For instance, inputs flow into the caller object of `HttpURLConnection.connect()` will also cause `VS_Input` ECVs.

`VS_Public`. VSink APIs in this category are those which might not directly transmit privileged resources to attackers but will make them public to other apps. An attacker could then leverage a local app to steal privileged resources outputted from these `VS_Public` APIs. One typical kind of `VS_Public` sinks is the output methods defined in the `android.util.Log` class, such as debug-level method `d(String tag, String msg)`. Vulnerable components might put privileged resources (e.g., GPS locations) to these output methods, and an attacker could collect these log outputs in runtime.

Besides VSinks, data source APIs, as the role of "source", are also required for some ECV detection (e.g., tracking flows from source APIs to `VS_Public`). In this work, we consider the data source APIs as those can read permission-protected resources. Similarly, we use the term `VSource` to represent these source APIs. Take the `BLUETOOTH` permission as an example, `BluetoothDevice.getUuids()` and `BluetoothDevice.getAddress()` are its two `VSource` APIs.

There are some critical APIs we do not counter into our VSinks. A notable example is `WebView.loadUrl(String url)`, which is used to load the given URL. Prior work [35] and [10] have shown this API can potentially cause origin-crossing or file stolen vulnerabilities, when encounters malicious HTML files. However, we notice the root cause of these browser-related vulnerabilities is the same-origin-policy (SOP) bypass, rather than the confused deputy in exposed components. We thus do not consider such kind of browser APIs into our current version of VSinks.

4.1.3 Challenges

We anticipate our approach based on static analysis, as opposed to dynamic testing, for its scalability and extensive code coverage. Like two previous static methods [4, 13], we model the ECV detection problem as a sink-based flow analysis problem. But we notice there are some challenges that have not been tackled in prior works. These challenges, as elaborated below, motivate us to propose the sink-driven detection approach.

- **Systematic VSink collection**. To avoid false negatives, we require a systematic method towards collecting complete VSinks. This task is complicated by the fact that there are vast Android APIs (and they are continuously growing). A recent work [17] has employed machine learning to find and classify Android source and sink APIs. Unfortunately, it only considers data sinks, and some metrics they used

are not applicable for non-data sinks. The state-of-art sink selection in the context of ECV problem is 180 sources and sinks collected by CHEX [13]. We believe more sinks could be extracted using a systematic method.

- **Handling diversified VSinks**. Diversified analysis requirements of VSinks require a general but also efficient method. Simply combining previous works detecting specific ECVs cannot form a general method, because they cannot differentiate which detection policy should be adopted for a certain sink. For detecting all four kinds of VSinks, an approach should be aware of the VSink categories which are classified beforehand. But this is only one condition for an ideal method. To be also efficient, the general detection method should be modular and choose the right module to handle a specific ECV problem. For example, taking height-weight dataflow analysis for analyzing `VS_Direct` ECVs is not necessary and inefficient.

- **Reducing false positives**. It is known that static analysis inherently has false positives, because no actual dynamic execution is involved. Thus, how to reduce them is a challenge for all static analysis methods. It becomes especially important when we try to cover more sinks and detect multiple kinds of ECVs.

There are a few other typical challenges we do not intend to address in this work. First, the well-known technical puzzles for analyzing Java reflection and native codes are also out of our scope, like many previous Android works. Second, considering API call chains rather than pure sink-based analysis is also beyond the scope of this work. Indeed, no ECV detection tools had covered this complicated problem before, to the best of our knowledge. Finally, cross-component ECV detection via Android IPC communication is not considered in the current version of ECVDetector, but we plan to leverage [38] to tackle this problem.

4.2 ECVDetector Design

In this section, we present our design of ECVDetector, which implements the sink-driven approach to systematically tackling the ECV detection problem. As shown in Figure 4.2, ECVDetector first selects and classifies VSinks. Then based on these VSinks, we propose a general detection method to identify all categories of potential ECVs. This method organically combines forward reachability and backward dataflow analysis, and drives them by the characterized VSinks. In particular, we take the backward, instead of previous forward, dataflow analysis for adapting more categories of sinks, such as the `VS_DirectByParam` category.

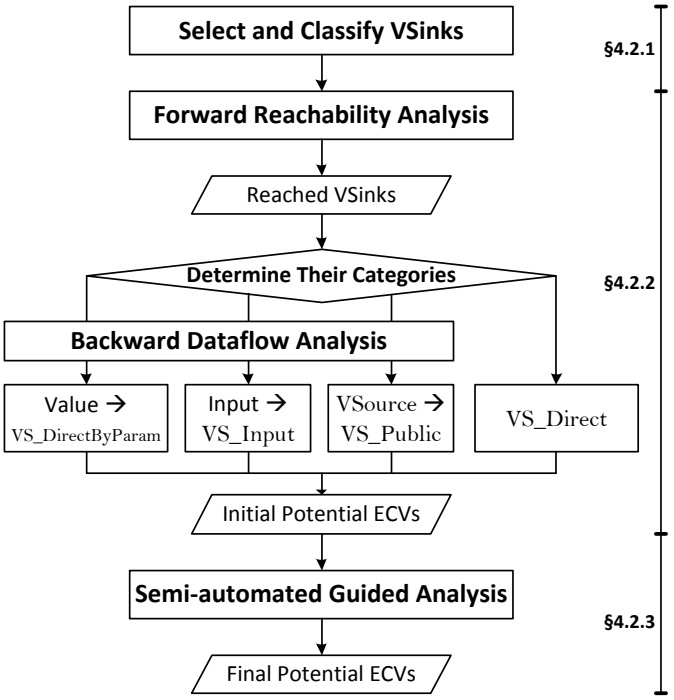


FIGURE 4.2: The overall work flow of ECVDetector.

More specifically, the first step produces a number of categorized VSinks, which are the dataset for the subsequent analysis. Then ECVDetector employs a core module with a typical forward reachability analysis to tackle the common analysis task (i.e., identifying all reachable VSink calls from attack entry points), and further leverages a customized backward dataflow analysis to handle different categories of VSinks in several dedicated modules. As the last step, a semi-automated guided analysis is conducted for excluding some sink-specific false positives. In the following three sections, we will discuss each critical analysis phase in more detail.

4.2.1 VSink Selection and Classification

We propose a systematic strategy for VSink selection and classification. The basic idea of this strategy is that we combine multiple metrics (e.g., permission semantics and API names) to systematically define rules. These rules are made according to a simple, but practical, rule syntax. We further write a rule interpreter to automatically select and classify VSinks according to the defined rules.

Most of VSinks are selected from permission-protected or privileged APIs, according to the *C1* channel in Section 4.1.2. However, Android does not provide a complete and

accurate mapping for permissions and their corresponding API calls. In fact, permission information provided by Android developer document is limited and might contain errors [39]. However, two great works [39, 40] have attempted to address this limitation. Specifically, Stowaway [39] constructs mappings for Android 2.2 framework using dynamic API fuzzing. In contrast, PScout [40] employs a version-independent static analysis method to extract permission specifications from multiple versions of Android.

In this thesis, we adopt API-to-permission mappings from Stowaway instead of PScout for two reasons. First, although PScout provides a significant number of undocumented APIs, we only select VSinks from the range of documented APIs (see Section 4.1). In terms of documented APIs, PScout does not provide more permission mappings than Stowaway. Second, we find Stowaway produces a more accurate mapping than PScout, in terms of fewer false positives. This might due to the fact that nearly every mapping found by Stowaway is verified by dynamic execution. The accuracy of permission mapping is important for our VSink selection, because any incorrect VSink will directly introduce false positives to our ECV detection. Therefore, currently we only adopt the mappings from Stowaway. In the future, we could also take advantage of the version-independent feature of PScout.

We obtain a total of 456 validated documented APIs from Stowaway. Assessing each API semantic is practically infeasible for two reasons: (1) it would take a quite large manual workload, and (2) two much manual analysis without a principle may incur some inaccuracy (e.g., assign wrong VSink tags). Therefore, we propose to combine multiple metrics for systematic selection and classification. These metrics include permission semantics and levels, API names, parameter and return-value types. Among all metrics, permission semantic is the major metric for our system. In fact, we only extract 56 kinds of permissions from those 456 documented APIs. Therefore, we can divide them into 56 clusters (if an API needs multiple permissions, we choose its first marked permission), because APIs marked with the same permission are likely to share similar VSink nature. For instance, combining with quick scan of API names, we find all 6 APIs under `SEND_SMS` permission can be directly categorized into `VS_Direct`.

To facilitate multiple metrics based selection and classification, we further design a simple, but practical, rule syntax, as illustrated in Figure 4.3. Each rule is a five-element tuple, which describes what tag a particular API would be assigned, when one metric of this API satisfies a special string pattern. We define four kinds of pattern-matching actions, such as "START" for "start with" action. Finally, the tag in the syntax mainly includes four VSink categories. Besides them, we also define a special tag named "Tag_Delete", which can be used when we want to exclude an API. Based on this syntax, we define four general rules and a set of permission-specific rules for extracting

categorized VSinks from permission mappings. We then write a rule interpreter to automatically select and classify VSinks according to the defined rules.

```
<rule> ::= <metric> <action> Pattern TAG <tag>          (1)
<metric> ::= @class | @method | @param | @return        (2)
<action> ::= START | IS | CONTAIN | END                  (3)
<tag> ::= VSink | Tag_Delete                            (4)
```

FIGURE 4.3: The rule syntax for VSink selection and classification.

Besides the privileged APIs, another channel to select VSinks is the APIs that would make security impact on internal status of apps. Three general kinds of such APIs have been considered in our work. The first kind is the file operation APIs, such as the `write` and `append` methods from the `java.io.Writer` class. Another kinds of APIs is the database-related APIs, such as those from `SQLiteDatabase` and `ContentResolver` classes. The third one is the logcat APIs, i.e., logging APIs from the `android.util.Log` class. We then design three additional rules to automatically join them into our VSink set. Moreover, APIs from the `AndroidHttpClient` class (missed by both Stowaway and PScout) are similarly complemented into our final VSink results.

4.2.2 Forward and Backward Analysis

4.2.2.1 Forward Reachability Analysis

We employ an iterative intra-procedural algorithm with flow sensitivity to perform reachability analysis (summarized in Algorithm 1). Note that since reachability analysis only relies on control flow, context-insensitive analysis is enough in our forward module. The algorithm first constructs a control flow graph (CFG) for each method, and then traverses every statement in a particular order according to the search strategy (e.g., DFS). For each call site, we determine whether it is a VSink or can be resolved into a method defined by the app, respectively. In particular, this call site resolving procedure is performed on demand along the forward analysis. We maintain two lists for caching reached VSinks and resolved methods. After the lists stop growing, we obtain all reachable VSinks and then invoke specific modules to further process them. Finally, the algorithm will handle each resolved method in the same way.

We generate call chains to facilitate inter-procedural backtrack analysis. A call chain is a path of call graph, from an entry caller to an ending callee. Generating individual call chains, instead of a whole call graph, could ease the path track procedure of subsequent modules and allow them concentrate on the design of flow analysis. In contrast, traversal

Algorithm 1 Forward Reachability Analysis

```
Input: entryPts = [entry point], vSinks = [VSink]
 1: for all ep ∈ entryPts do
 2:     IterativeForward(ep, initchain)
 3: end for
 4:
 5: procedure IterativeForward(method, chain)
 6:     reachedS = [], nextM = [], chain.add(method)
 7:
 8:     cfg ← BuildCFG(method)
 9:     for all cs ∈ CallSites(DFS(cfg)) do
10:         if cs ∈ vSinks then
11:             reachedS.add(cs)
12:         else
13:             rc ← ResolveCall(cs)
14:             if rc ∈ [app defined method] then
15:                 nextM.add(rc)
16:             end if
17:         end if
18:     end for
19:
20:     for all r ∈ reachedS do
21:         category ← JudgeCategory(r, vSinks)
22:         InvokeSpecificModule(category, r, chain)
23:     end for
24:     for all m ∈ nextM do
25:         IterativeForward(m, ShadowCopy(chain))
26:     end for
27: end procedure
```

between two nodes on call graph might involve several parallel paths. Moreover, an additional dataflow fact *join* operation needs to be considered in the dataflow analysis using non-linear graph traversal.

Our call chain captures not only caller and callee methods, but also precise calling context information by recording call-site heap locations and their call strings (both are not shown in Algorithm 1, for simplicity). On one hand, the heap location context information is essential for backward modules to jump back to each original call site. Otherwise, ambiguity might arise when a caller method contains multiple similar call sites targeting the same callee. On the other hand, we leverage call string to avoid re-analysis of callee method with the same dataflow value context. More specifically, with the help of SSA IR form[1], we can distinguish call sites with different entry dataflow values through simply investigating their call strings.

[1]SSA represents static single assignment, while IR is short for intermediate representation.

4.2.2.2 Backward Dataflow Analysis

Except for `VS_Direct`, the other three kinds of VSinks require further backward dataflow analysis. Although each backward analysis is independent for a dedicated task, they share a common backward dataflow analysis method. We thus first design this common backward method and then apply it to the three dedicated modules.

Our backward method relies on call chain (generated by forward module) to achieve inter-procedural context-sensitive backtrack analysis (shown in Algorithm 2). The core of this algorithm is an iterative backward analysis procedure. This iterative procedure starts with extracting SSA-form method body from call chain according to current chain index. Then, it initializes a intra-method taint set and joins the incoming tainted variables. After this, we need to locate the starting call site according to the provided calling context. If calling context is the null (this can happen when we backtrack between two originally disconnected callback functions), we directly take the last call site as our starting point. Similarly, we join the new tainted variables obtained from starting call site into the taint set. We then loop all call sites before the starting point. For each tainted call site, we further determine whether we need to propagate the taint into new variables. Moreover, different dedicated modules may choose to mark result for tainted `VSource` or constants at this time. After the loop, some modules would further inspect whether there are tainted inputs. Finally, the algorithm will judge whether it needs further backtracking, according to the current index number and variables in taint set. For example, if we already backtrack to the first method in chain, or no parameters and fields tainted, the algorithm will terminate.

Generating proper taint objectives is important for our backward analysis. First, we need to obtain appropriate initial taints from reached VSink calls. Since our current VSinks have no fine-grained information to indicate which parameter is critical, we then take a conservative approach that taints all encountered parameters to avoid any false negative. Moreover, some VSink APIs have no parameters involved (e.g., previously mentioned `HttpURLConnection.connect()`), we thus taint their caller object. Second, we need to maintain a mapping for tainted parameters during the procedure of method switching, so that we can obtain the correct variable format under different SSA method bodies.

We further design three dedicated modules for tackling ECV detection problems under `VS_DirectByParam`, `VS_Input`, and `VS_Public`. With the help of proposed backward method, these modules are relatively easy to design. Specifically, for `VS_DirectByParam`, we mark the result from tainted constants and inputs. Because sometimes static analysis cannot obtain accurate parameter values, e.g., when they rely on dynamic execution

Algorithm 2 Backward Dataflow Analysis

Input: *cy*: category, *r*: a reached VSink call, *chain*, *vSrc*

```
1: ii ← chain.size()-1, itv ← GetTVarsFromCallSite(r)
2: IterativeBackward(ii, itv, r)

   ▷ i: index, tv: tainted variables, cc: calling context
3: procedure IterativeBackward(i, tv, cc)
4:     sb ← chain.get(i).GetSSABody()
5:     ts ← InitTaintSet(), ts.join(GetTaintedVars(tv))
6:     cs_start ← GetStartCallSite(cc)
7:     ts.join(GetTVarsFromCallSite(cs_start))
8:
9:     cs_old ← cs_start
10:    while sb.hasPreviousCallSite(cs_old) do
11:        cs_new ← sb.getPreviousCallSite(cs_old)
12:        if isThisCallSiteTainted(cs_new, ts) then
13:            ts.join(Determine-PropagateTaint(cs_new, ts))
14:            MarkResult_VSrc(cy, cs_new, vSrc)
15:            MarkResult_Constant(cy, cs_new)
16:        end if
17:        cs_old ← cs_new
18:    end while
19:    MarkResult_Input(cy, ts, chain)
20:
21:    if isContinueIterative(i, ts, cy, chain) then
22:        lcc ← chain.get(i).GetLastCallContext()
23:        IterativeBackward(i-1, TransformTVars(ts), lcc)
24:    end if
25: end procedure
```

outputs. Therefore, we adopt every tainted constants into `VS_DirectByParam` result, to mimic the ideal values. While for `VS_Input` and `VS_Public`, our main task is to backtrack whether there are tainted inputs and `VSource`, respectively

4.2.2.3 Analysis Enhancements

We also design three kinds of enhancements to the basic forward and backward analysis in ECVDetector. In general, these enhancements can help ECVDetector reduce false positives, avoid unnecessary analysis overhead, and output more expressive result logs.

The first enhancement is to validate system-only broadcast checking in the flow analysis. Broadcasts are system-wide events (e.g., battery is low), which will be delivered to registered Broadcast Receivers when the corresponding events occur. For example, an Broadcast Receiver with the name `com.example.BootReceiver` (see Figure 4.4(a)) registers the `BOOT_COMPLETED` broadcast, which will then trigger `BootReceiver` when

the system has booted. Note that although third-party apps can also declare their own broadcasts, many broadcasts are only sent by the operating system and prohibited by non-system senders [39]. Therefore, attackers cannot inject a fake broadcast Intent with `BOOT_COMPLETED` as the action name into `BootReceiver`. However, it is still insufficient to prevent external crafted inputs, since attackers can directly trigger `BootReceiver` by set the explicit Intent target name. A safe and common way to mitigate this issue is to explicitly check the action name of system broadcasts in the code, as suggested in [25] and [38]. A typical code pattern for such system-only broadcast checking is illustrated in Figure 4.4(b).

```
<receiver android:name="com.example.BootReceiver">
  <intent-filter>
    <action android:name=
          "android.intent.action.BOOT_COMPLETED"/>
  </intent-filter>
</receiver>
```

(a) An example Broadcast Receiver registering the `BOOT_COMPLETED` broadcast.

```
1 public class BootReceiver extends BroadcastReceiver {
2
3   public void onReceive(Context context, Intent intent) {
4     if ("android.intent.action.BOOT_COMPLETED".
          equals(intent.getAction())) {
5       // other codes
6     }
7   }
8 }
```

(b) A typical code pattern for checking system-only broadcasts.

FIGURE 4.4: Two examples on system-only broadcasts.

Since broadcast checking can efficiently protect exposed Broadcast Receivers, we thus design a validation capability for system-only broadcast checking in ECVDetector's flow analysis, as an enhancement to the forward analysis in Section 4.2.2.1. Besides helpful to reduce false positives, this validation can also improve the system performance, because ECVDetector can avoid the subsequent analysis once it identifies checking. Our validation is targeted at the code pattern in Figure 4.4(b) (i.e., the If-Else checking using `equals` API in Java's `String` class), since it is the most straightforward way to perform broadcast checking. We also believe it is easy for ECVDetector to cover other kinds of checking patterns, once their domain knowledge is provided.

To facilitate accurate validation, a critical job is to collect sufficient system-only broadcasts. Note that our goal is to identify enough broadcasts that could cover most app cases, and proposing a new way to dig out a complete set is out of this thesis scope. There are two existing related resources we can leverage. First, the `AndroidManifest.xml` file in each version of Android source code defines a list of system broadcasts with the tag name `protected-broadcast`. We thus collect 133 such broadcasts from the recent Android 4.3 platform. Second, Stowaway [39] uncovers 62 system broadcasts by dynamic testing, including those dynamically declared in the code. We then merge these two

broadcast sets into a new one, and finally obtain in total of 143 unique system-only broadcasts for ECVDetector.

Second, we avoid backtracking some uncritical parameters to reduce overhead. A typical example is the first `tag` parameter of logcat APIs, such as `Log.v(String tag, String msg)`. Developers usually assign insensitive values (e.g., string constants) into this parameter, thus there is no need to analyze it. Otherwise, the backward module may waste tracing several methods before arriving its initialization method.

Third, we output input-related variable values for more expressive result logs. Our backward analysis could taint a dependence between the input and parameter, like CHEX [13]. However, such coarse-grained dependence without detailed propagation knowledge sometimes is not enough. To this end, we choose to output some input-related variable values into our result logs, such as the log like `Input:r4 = r2.getStringExtra("referrer")`, where r2 is the original Input or Intent object. In this way, we can obtain more expressive and meaningful result logs.

4.2.3 Semi-automated Guided Analysis

"mount" to '$r10 = virtualinvoke $r9.<java.lang.Runtime: java.lang.Process exec(java.lang.String)>("mount")'
"logcat -d " to '$r14 = virtualinvoke $r3.<java.lang.Runtime: java.lang.Process exec(java.lang.String)>($r13)'
"referrer" to 'virtualinvoke r0.<android.database.sqlite.SQLiteDatabase: int delete(...)>("referrer", null, null)'

FIGURE 4.5: Logs showing example false positives in `VS_DirectByParam` category.

We need to further filter some false positives in the initial set of potential ECVs identified by previous analysis. These false positives are mainly from two VSink categories: `VS_DirectByParam` and `VS_Input`. The reason is that VSinks in these two categories generally rely on parameters to exhibit their specific behaviors. Therefore, analyzing these VSinks is usually sink-specific and parameter-specific, meaning that we have to combine detailed sinks and their parameter values for detection. However, it is nearly impossible for ECVDetector to automatically handle it, because too much domain knowledge is needed.

To address this issue, we propose semi-automated guided analysis to quickly filter false positives. Its basic process is illustrated in Figure 4.6. In particular, we take advantage of the result logs outputted by ECVDetector. Figure 4.5 shows the logs of example false positives in the `VS_DirectByParam` category. Specifically, we first collect all unique results logs from `VS_DirectByParam` and `VS_Input`. Then, we conduct manual analysis to find all false positive logs and their patterns. This step largely relies on expert

knowledge. Little effort is actually required because many logs are similar. Finally, we apply our extracted false positive log pattern to all related apps and take scripts to automatically filter those matched cases.

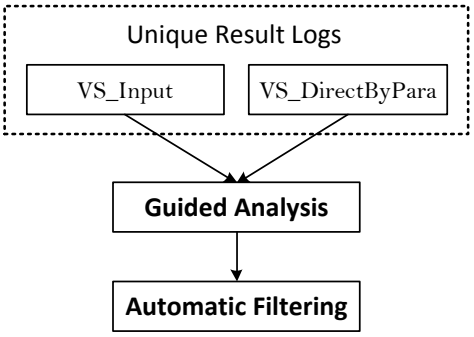


FIGURE 4.6: Basic process of semi-automated guided analysis.

4.3 ECVDetector Implementation

We have implemented ECVDetector with 4,565 lines of Java code, Python scripts and shell scripts. As shown in Figure 4.7, ECVDetector consists of four components. Specifically, Manifest Analyzer is implemented only in Python scripts, while the other three components (i.e., Entry Point Locator, Vulnerability Analyzer and VSink Selector) are based on the Soot framework [41]. ECVDetector contains four execution steps: one preparation step (i.e., step 0 at the bottom of Figure 4.7) and three analysis steps. The preparation step is executed only once for generating VSinks. Then, ECVDetector analyzes each Android app for ECVs in three consecutive analysis steps.

VSink Selector. By running VSink Selector with the inputs of Android framework code and Stowaway permission mappings, we obtain a total of 372 categorized VSinks. Among them, 137 APIs are `VS_Direct`, 23 `VS_DirectByParam`, 167 `VS_Input`, and 45 `VS_Public`. Moreover, to facilitate detecting ECVs involving sensitive sources, we also select 183 `VSource` based on the results of Stowaway and SuSi [17]. Regarding the data sinks in SuSi, we find most of them cannot serve our vulnerability-specific purpose, due to their selection motivation for privacy leak detection.

To adopt the Stowaway mapping into our VSinks, however, faces a technical challenge that method signatures in Stowaway mapping are incomplete. In fact, these incomplete signatures are only sub-signatures without return-value types, thus would incur inaccuracy for Vulnerability Analyzer. To this end, we design several estimation rules to restore complete method signatures by analyzing corresponding Android framework

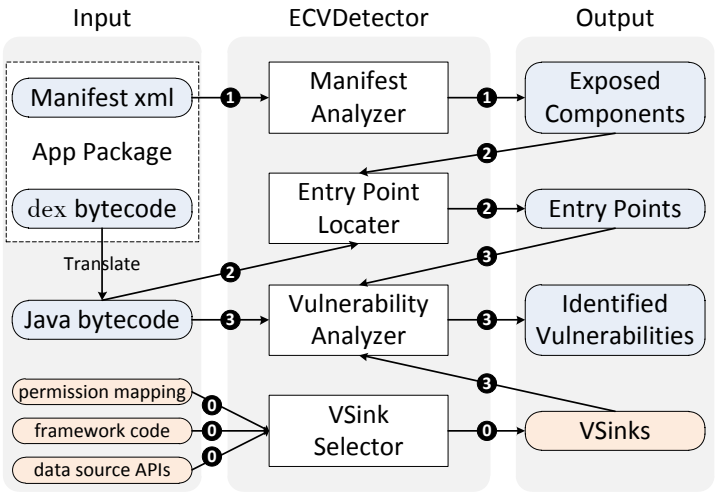


FIGURE 4.7: ECVDetector architecture. VSink Selector and Vulnerability Analyzer are two major components to implement our sink-driven approach.

code. Moreover, since our analysis program requires right method sub-signatures to facilitate signature restoring, we unexpectedly identify several kinds of inaccuracy issues existing in Stowaway results. These issues exist might because Stowaway also employs some manual efforts to produce the mapping, thus introduce some inaccuracies. We then ran our analysis program several times to inspect the error messages (in each run), which prevents us to successfully restore all signatures. For identified issues in each run, we manually fix them by batch replacing.

Manifest Analyzer. This component consists of two parts: an xml parser to extract all essential information inside each manifest file, and a script to identify exposed components by analyzing extracted information. We determine exposed components according to the rule in Section 2.1.2.

Entry Point Locator. The main task of Entry Point Locator is to locate all entry points, which would serve as the starting points for Vulnerability Analyzer. Basically, entry points are fixed callback interfaces defined by Android programming paradigm, such as onCreate and onStart. Particularly, onCreate is an initialization point which will be called when a component is started up. Moreover, several kinds of entry points are special, in terms of they can accept external attack inputs. While other points either take zero parameter (e.g., onResume and onStop), or only contain inputs cannot be manipulated by attacker (e.g., `Bundle` argument in Activity's onCreate entry point). Since `VS_Input` ECV detection is related to those special entry points, we identify them in Android framework and list in Table 4.1. Three kinds of components contain such entry points we concerned, while Activity needs to actively call `getIntent` fucntion to receive external inputs.

TABLE 4.1: Our aimed entry point functions.

Component	Callback functions that accept attack inputs
Service	onBind(**Intent** intent) onStart(**Intent** intent, int startId) onStartCommand(**Intent** intent, int flags, int Id) onHandleIntent(**Intent** intent) handleMessage(**Message** msg)
Receiver	onReceive(Context context, **Intent** intent)
Provider	query(**Uri**, String[], String, String[], String) insert(**Uri**, ContentValues) update(**Uri**, ContentValues, String, String[]) delete(**Uri**, String, String[]) openFile (**Uri**, String)

Another task is to model lifecycle for identified entry points. This is necessary because the entry points will not call each other due to their callback nature, thus cause disconnected static flows. Moreover, there are some initialization functions even before the onCreate interface, such as <clinit> and <init>. We also need to consider them in modeling lifecycle, so that backward module could backtrack to the initial definition. In the current ECVDetector, we model the lifecycle by defining several continuous phases. Each phase may contain a manually connected flow or several asynchronous entry points. For example, we define an "initial" phase to connect the initial flow (i.e., <clinit> → <init> → onCreate), and a "main" phase to cover all asynchronous special entry points in Table 4.1.

Vulnerability Analyzer. This component mainly performs the sink-driven forward and backward analysis in Section 4.2.2. In the process of implementing Vulnerability Analyzer, we come across two major technical issues due to: (1) object-oriented (OO) language used by Android development, and (2) Android's event-driven nature.

The first issue arises from the inheritance nature of OO language during the call site resolving in the forward module. To obtain the target method of a call site, it is essential to resolve the type of target class object. However, due to the inheritance, it is hard to statically determine what concrete class an object would represent. More specifically, the inheritance nature allows developer to use superclass or interface type to represent a subclass object, which causes the ambiguity. We tackle this problem to a great extent by leveraging typed Soot IR [41], which is calculated by a fast type inference algorithm [42]. Besides the object type resolving, OO language's inheritance nature also makes locating the right method definition complex, even if we have obtained the exact object type. For example, `toString` method invoked by `android.graphics.Bitmap` object is actually not defined by `Bitmap` class itself. Indeed, we have to track back to `java.lang.Object` class to obtain the method definition of `toString`. A straightforward way for mitigating this

problem is to maintain a comprehensive class hierarchy. In our ECVDetector prototype, we also take advantage of the automatic method resolution provided by Soot according to Java Virtual Machine Specification [43].

The second issue is the static flow discontinuity problem caused by Android's event-driven nature [4]. A typical disconnected example is the flow between `start()` and `run()` method in `java.lang.Thread` class. In fact, Android OS connects these two methods dynamically through the underlying thread scheduler. However, a static tool, without specific knowledge, cannot directly predict this kind of dynamic connecting behavior. Nevertheless, it is necessary for Android static analysis tools to tackle this problem in a satisfactory way. Otherwise, some false negatives would arise. In ECVDetector, we model a number of dynamic flow connecting behaviors as pre-defined knowledge to build continuous call chains. The main modeled flow connecting behaviors are summarized in Table 4.2. According to this table, ECVDetector is capable to connect the disconnected flows occurred in thread scheduling, timers, location updates, and so on.

TABLE 4.2: The main dynamic flow connecting behaviors modeled by ECVDetector.

Class name	Modeled Flows
Thread	start → run
AsyncTask	execute → doInBackground
Handler	sendMessage → handleMessage
Timer	schedule → run
LocationManager	requestLocationUpdates → *LocationListener*
TelephonyManager	listen → *PhoneStateListener*

4.4 Evaluation

To evaluate the efficacy and performance of ECVDetector, we carried out an evaluation with top 1K Android apps from Google Play. The reason we evaluated these popular apps is that the impact of ECVs relies on the popularity of vulnerable apps. In other words, vulnerable apps with few installs would not cause big security impact, because it is less likely to let those limited users also install the attack apps. Specifically, these top apps were selected according to their user review numbers, and most of them were crawled recently (between June and July 2013). Therefore, our app dataset could represent the recent versions of top Google Play apps that users may install in their phones. Moreover, our selected apps were fully unique in terms of the package names, which made our dataset more distributed.

In this section, we first discuss how we conducted the experiment with this dataset, including some findings and knowledge we obtained during the procedure of experiment. Then we report our identified ECVs and conduct case studies of some representative vulnerable apps. Finally, we depict the result of performance evaluation for ECVDetector.

4.4.1 Experiment and Findings

One essential step before the ECVDetector experiment is to extract the AndroidManifest.xml for each app. As mentioned in Section 4.3, we took apktool for unpacking the app and obtaining the manifest file. However, we found not all apps could be correctly handled by apktool. Indeed, six apps of our dataset fail and crash apktool with several Java exceptions, maybe due to the potential bugs in apktool or some apps try to protect itself from the decompiling of apktool [44]. This finding also gives a kind hint to all previous work based on apktool, such as [45] and [46].

We then leveraged the Manifest Analyzer component of ECVDetector to discover exposed components from the rest of 994 apps. Among them, we successfully parsed and analyzed 992 manifest files. The two failure cases were due to the invalid encodings that cannot be handled by the Python XML DOM library. More specifically, one failed case (the popular Titanium Backup app) contains the Chinese and Korean characters for some component names. These two failed manifest files could be manually analyzed for obtaining exposed components, but in this thesis we just ignored them for the automatic experiment.

In summary, we identified a total of 7,664 exposed components from the remaining 992 apps, and 6,582 of them were unique in terms of the component name. The detailed amounts of exposed components classified by component types are illustrated in Figure 4.8. One major finding based on this figure is that there are significantly more Activity and Broadcast Receiver exposed components than the other two component types, around ten times. The reason behind this can be explained by these two facts. First, many apps need exposed Activities to finish the user intention of app switching, such as launching from the launcher app. Second, in order to receive system broadcasts, the corresponding Receivers have to expose themselves to the Android framework.

Our experiment for ECV detection focused on the exposed Services and Broadcast Receivers. More specifically, our target was 2,551 unique exposed components, including 378 Services and 2,173 Receivers (see Figure 4.8). The reason for skipping Activities is mainly because exploiting Activity vulnerabilities usually requires user's intention (e.g., the Adobe Activity vulnerability shown in [12]), and the complicated attacks against

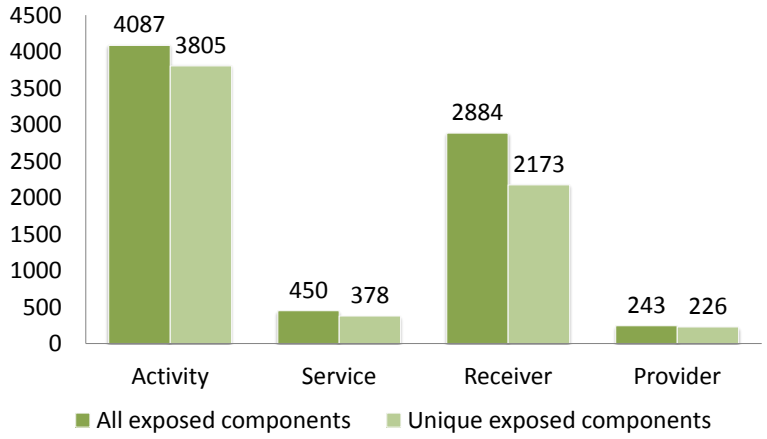


FIGURE 4.8: The amounts of all and unique exposed components across four component types.

Browser Activities [35] are out of this thesis scope (see Section 4.1.2). As for Content Providers, although ECVDetector is capable of statically detecting some potential Provider vulnerabilities, heavy manual efforts are still required even with the dedicated ContentScope [5] tool.

Next, we ran ECVDetector against those 2,551 exposed components. The experiment was performed on a single Dell PowerEdge storage server, equipped with four 2.40GHz CPUs and 12GB of RAM. To optimize the throughout, we set the timeout for processing each component to two minutes. This threshold was quite reasonable, because we found most of components could be analyzed within 10 seconds (see Section 4.4.3 later).

During the experiment, we found it was challenging to prevent Soot from crashing when loading Java classes for analysis. We spent many efforts to make ECVDetector successfully analyze all 2,551 components, and our knowledge listed as follows might be helpful other Soot-based Android tools [38, 47, 48]. The first knowledge is that we should provide as many underlying Android classes as possible to Soot. These classes would help Soot resolve most of Android-specific types. Specifically, we provided eight platforms of Android SDKs to Soot, from the old 1.6 to the recent 4.3. Moreover, two platforms of hidden and system Android APIs were also provided to Soot, as well as one version of the Google APIs (e.g., Google Map APIs). However, we still encountered many failed cases due to missing app-specific classes. In order to solve this issue, we made use of the second knowledge: ask Soot to only load classes on demand, and if errors still occur then set the Soot option `allow_phantom_refs`. This option would let Soot create a phantom class for each missing class.

Another interesting finding is about system-only broadcasts, which are discussed in Section 4.2.2.3. Our finding about these broadcasts for the 2173 exposed Receivers is

divided into three parts. First, we identify 433 of them contained the checking for system-only broadcasts in their codes. This result suggests our designed validation capability (Section 4.2.2.3) could effectively alert the false positives due to system-only broadcast checking. Second, total 40 broadcasts of our selected 133 system-only broadcasts are checked. Third, we also count the numbers of checked broadcasts, and the top three result is listed in Table 4.3. The fourth most checked broadcast is `TIMEZONE_CHANGED`, but with only 19 times.

TABLE 4.3: Top 3 checked system-only broadcasts.

The Checked System-only Broadcast	#
android.intent.action.BOOT_COMPLETED	157
android.net.conn.CONNECTIVITY_CHANGE	57
android.intent.action.PACKAGE_ADDED	43

TABLE 4.4: Four categories of identified ECVs and their representative vulnerable apps.

ECV Category	# of Apps	Representative Apps	
		Package Name	Vulnerability Description
`VS_Direct`	6	com.jb.gosms	Force it to send SMS to phone no. specified by input
		com.gau.go.launcherex.gowidget.switchwidget	Force it to enable or disable wifi and bluetooth
		com.bwx.bequick	Force it to open the camera as flash light
`VS_DirectByParam`	5	com.cleanmaster.mguard	Force it to silently uninstall arbitrary apps and etc.
		mominis.Generic_Android.Ninja_Chicken	Force it to delete its internal "MESSAGE" database table
`VS_Input`	25	com.zlango.zms	Force it to change the status of messages in SMS databases
		com.antivirus	Start the private core AVService with extras specified by input
		com.doubleTwist.androidPlayer	Start a private service with dangerous action named "delete_db"
		com.linkedin.android	Launch a network request with the attributes specified by input
		com.ebuddy.android	The beta update Receiver can be cheated to download fake apps
`VS_Public`	13	com.symantec.mobilesecurity	getLastKnownLocation() is outputted to the logcat
		air.com.bitrhymes.bingo	getDeviceId() along with post URL are outputted to the logcat
		com.levelup.touiteur	getAllNetworkInfo() is outputted to the logcat
		com.sec.spp.push	getConnectionInfo() is outputted to the logcat

Through the automatic analysis for the 2,551 exposed components of 992 apps, we discovered the initial set of potential ECVs with 348 affected apps. As previously shown in Figure 4.2, this result was analyzed only by the forward and backward analysis in ECVDetector. We still needed to perform the semi-automated guided analysis for two ECV categories (`VS_DirectByParam` and `VS_Input`), as discussed in Section 4.2.3. Specifically, there were 172 and 357 unique logs (need guided analysis) for `VS_DirectByParam` and `VS_Input`, respectively. Nevertheless, most of them were similar to each other, only 6 kinds of `VS_DirectByParam` APIs and 16 kinds of `VS_Input` APIs were identified.

Therefore, we did not spend much effort (around one hour in total) in validating these logs. ECVDetector then automatically filtered various sink-specific false positives according to our extracted log pattern. Eventually, we identified a total of 103 potential vulnerable apps. More specifically, there are 31 apps affected with potential `VS_Direct` ECVs, 25 for `VS_DirectByParam`, 56 for `VS_Input`, and 16 for `VS_Public`.

4.4.2 Identified ECVs

The 103 potential vulnerable apps (identified in the last section) were further manually verified for the real ECVs, due to the inherent limitation of static analysis, as discussed in Section 4.1.3. In our verification process, we tried to follow a relatively more conservative principle than the closest related work CHEX [13]. A typical example is that the vulnerable case of "Object embedded in input used to start Activity" in CHEX would not be directly treated as a vulnerability in our principle. Instead, we also require this action of starting Activity could make some security impact to the internal status of victim apps for becoming an ECV.

Overall results. In the end, we tagged 49 apps as vulnerable from 103 candidate ones. We confirmed them mainly by carefully auditing the decompiled codes and manifest files (with the help of result logs and call chains produced by ECVDetector), similar to the verification used by CHEX. Note that for each vulnerable app, we only tagged it to one major ECV category, even if this app might be affected by several ECVs. The main result is summarized in Table 4.4, including the number of each category of identified ECVs and their corresponding representative vulnerable apps. To the best of our knowledge, most of them are not disclosed previously, thus are the zero-day ECVs. Since these vulnerable apps are from top 1K in Google Play, their vulnerabilities may incur real security consequences among many users. We have reported several particularly serious cases to corresponding vendors, and received the acknowledgements from Go Dev team and LinkedIn.

The true positive rate of ECVDetector is not high (i.e., 47.6%, 49/103), but we argue this is acceptable when consider the following two factors. The first factor is the aforementioned conservative verification principle we used, which cause us to report less vulnerabilities. Some `VS_Input` APIs are the most affected by this principle, mainly the IPC APIs (e.g., `startActivity` and `startService`). This problem could be migrated when further cross-component inspection is involved. The second is due to the characteristics of some our selected sinks. For instance, the `removeAccount` `VS_Direct` API is commonly used in a reasonable scenario—when an user logs out her account, the corresponding app then removes her local account cache from the phone—which should

not be treated as an vulnerability. However, it is hard for ECVDetector to recognize this normal situation of API usage.

Case studies. We now conduct three cases studies to demonstrate ECVDetector's capability of detecting real serious ECVs under different categories. For each case, we further write an exploit in a zero-permission app to confirm the corresponding ECV could be effectively exploited. Automatically generating exploits is an interesting topic for aiding static analysis, but is out of this thesis scope and will be a future work. Here we hide app version information in case that some users are still using the vulnerable versions of apps.

Go SMS Pro (`com.jb.gosms`) is the top 1 messaging app with over 60 million installs. ECVDetector identified its exposed `CellValidateService` component leaked a security-critical flow path which arrived at the `sendTextMessage` sink API under the `VS_Direct` category. Moreover, its first parameter for assigning target phone number is completely controllable by external Intent. Thus, a zero-permission attack app can force Go SMS Pro to send SMS to arbitrary phone number. We also prepared a demo video (at https://www.youtube.com/watch?v=CwtNCwAHSRs) and reported it to Go Dev team on Sep 9 2013. From their active response, we knew we were the first reporter on this issue. This demonstrates the efficacy of ECVDetector, even when `sendTextMessage` is a commonly-used sink.

Clean Master (`com.cleanmaster.mguard`) is a quite popular clean up app with over 200 million worldwide installs. ECVDetector identified an external Intent can control its exposed `LocalService` to do privileged operations, such as clean up memory, restore apps, and silently uninstall arbitrary apps. Take silently uninstalling apps as an example, the flow to a `VS_DirectByParam` command execution API could be injected into attacker-supplied parameters (e.g., a victim app). Clean Master then executes the "pm uninstall" command to do silent uninstallation (of course, root permission needs to be pre-granted). This case demonstrates ECVDetector can uncover serious `VS_DirectByParam` vulnerabilities.

Lango Messaging (`com.zlango.zms`) is another top messaging app with over one million installs. ECVDetector identified an external Intent (at its `data` field) can flow into a `VS_Input` API (called `ContentResolver.update`) via the exposed `ZmsSentReceiver` component. We further found this exposed sensitive flow can enable a zero-permission app to maliciously change the status of some SMS messages. For example, a draft SMS can be changed into the "sent" status and this changing will be reflected in the UI of all installed messaging apps. Even worse, an incoming SMS can be set as an outgoing message, thus indirectly producing fake SMS messages. This case shows the capability of ECVDetector to detect `VS_Input` ECVs.

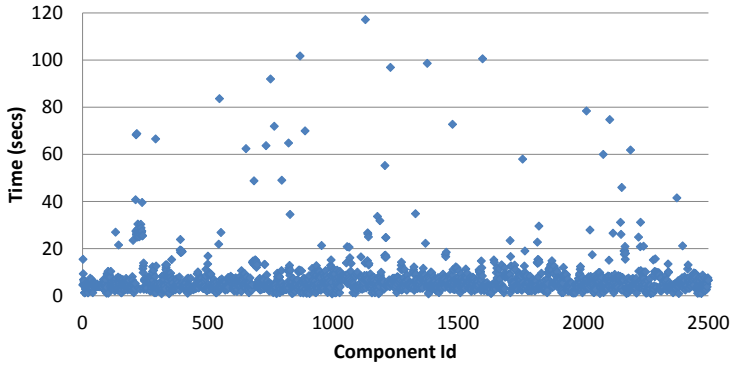


FIGURE 4.9: Detailed performance measurement of ECVDetector.

However, generating an effective exploit for Lango Messaging is surprisingly challenging, nearly taking us one-day effort. The major difficulty is how to obtain an appropriate `Uri` data field from infinite candidates with the format of "`content://`". The only hint is a domain knowledge required condition judgment (`MessageItem.getBoxId() != 2`). Eventually, we figure out the `MessageItem` object (from `MessageItemManager.get(Uri)`) refers to a SMS message, and its status should be not as "sent". Therefore, a target `Uri` can be "`content://sms/inbox/id`" or "`content://sms/draft/id`". The next step is to calculate a right message id. Since the attack app has no `READ_SMS` permission, it needs to launch brute-force attempts. In summary, this case shows static detection tools (like ECVDetector) and manual analysis are still necessary. By contrast, automatic exploit generation (like fuzzing [15, 49] or symbolic execution [50]) is nearly impossible for handling this case, due to the lack of domain knowledge and setting up related system conditions (in this case, some incoming and draft SMS have to be prepared).

`VS_Public` ECVs are relatively not so serious, and require more app-specific conditions to trigger the vulnerable paths. Therefore, we skip their case studies in this thesis.

4.4.3 Performance Evaluation

ECVDetector spent 6h 33m 35s to analyze the total 2,551 exposed components. Therefore, the average processing time for each component was 9.257s. This result suggests the threshold time of two minutes is quite reasonable. Among the 2,551 components, we found only 35 of them arrived this timeout value. The reason might be the complicated recursion codes that ECVDetector cannot recognize, although we have designed some mechanisms to help ECVDetector handle simple recursion. We further give the detailed performance measurement for the rest of 2,516 components, as shown in Figure 4.9. It is easy to determine that most of components could be analyzed within 10 seconds. Therefore, we conclude that the performance of ECVDetector is high.

4.5 Discussion

The false positives shown in the last section are mainly due to the variable semantics of the `VS_DirectByParam` and `VS_Input` APIs. It is hard to reduce these false positives even with dynamic analysis, because they also cannot determine which parameter values are critical and harmful. However, security analysts can extract the pattern of false positives from existing manual analysis, and feed them into the automatic filtering module of our guided analysis (Section 4.2.3). In this way, some false positives could be further excluded automatically.

Similar to other works, there are also some false negatives in the current ECVDetector prototype. However, it is hard to measure their quantity due to the lack of ground truth. Thus, we discuss several possible causes for these false negatives as follows. First, our modeling for connecting dynamic flows (see Table 4.2) is not intended to be complete, but we have tried our best to cover the common cases, similar to [4, 5]. Second, although we have proposed a systematic strategy to collect many sinks, it is impossible to cover a complete set. Third, our current implementation of backtracking call chains is also not prefect in that we skip analyzing other callee methods in the same level of hierarchy for code simplicity and performance consideration. Fortunately, only a small number produce false negatives is due to this limitation. Finally, the cross-component and cross-app ECVs [14] are not considered in our work, but we could leverage Epicc [38] to build a knowledge database and handle these cases.

4.6 Summary

In this chapter, we presented a new sink-driven approach to systematically tackle the ECV detection problem. This approach includes a systematic strategy for VSink selection and classification, and a general detection method to identify potential ECVs in Android apps. We implemented our sink-driven approach in a tool called ECVDetector. We successfully identified a total of 49 vulnerable apps across all four ECV categories in the top 1K Android apps. Future works include helping developers fix their vulnerable apps and deploying ECVDetector as a web-based detection service.

Chapter 5

Related Work

5.1 WebView and Mobile Browser Security

Webview security. The closest related works of our FileCross study are those on the security of WebView, which uses Android's default web engine (mainly webkit) APIs to help apps render web pages. However, most of these studies (e.g., [51–53]) mainly concern the insecure invocation between JavaScript and Java levels which may compromise a WebView app by misusing its exposed JavaScript interfaces. In particular, the file-based cross-zone scripting attack reported in [52] is similar to the FileCross attacks, but their attack follows the man-in-the-middle model where malicious JavaScript codes are injected by network adversaries. Without adopting a realistic threat model and proposing detailed attacks, they conclude that file-based cross-zone scripting vulnerabilities are *fortunately* fairly rare. In our study, we however show that `file://` vulnerabilities are prevalent in Android browsers. Additionally, our study is more general for testing major practices in the Android browser ecosystem (i.e., not limited to WebView flaws), and we also identify non-webkit vulnerable cases (notably Firefox and UC Browser HD).

Mobile browser security. Existing academic research on mobile browsers are more focused on their speed [54, 55], caching mechanisms [56, 57], energy consumption [58], and page load performance [59]. By comparison, there are no studies on their security issues, except for the two works by Amrutkar et al. on browser display security due to the inherent screen limitations of mobile devices. Their first work [60] evaluates the shortcomings of mobile SSL indicators, and the second [61] concerns the potential phishing attacks on mobile browsers. Compared to these attacks, our studied FileCross attacks do not require social engineering to phish users. Thus we believe FileCross attacks are more practical and might attract more interests from real-world attackers.

5.2 Security of Android Exposed Components

From the view of our FileCross study, the exposure of browsing interfaces in victim browsers is one important condition for launching FileCross attacks. Many previous works (e.g., [4, 5, 13, 14, 25, 38]) have studied the general exposed component problem from the perspective of information flow analysis. They aim at the source-sink problem that other apps can trigger dangerous APIs (i.e., sinks) in an exposed component from its exposed entry points (i.e., sources). Compared to the FileCross attacks, constructing their exploits are less complicated (due to the main focus on the raw Intent fields) and do not require the domain knowledge of browser SOP and file protocol. The exploit for Facebook Next Intent issue in [35] is also launched from `file://`, but it does not aim at stealing Facebook app's private files as the Facebook FileCross attack reported in [34].

In general, this class of vulnerabilities has attracted much research effort recently, since the seminal work of Davi et al. [62] on the basic ECV model focusing on permission leaks. Subsequently, several detection methods with different foci have been proposed. One category of these methods includes ComDroid [25] and Epicc [38], both of which try to identify several potential security flaws during the Intent-based communication. They aim at discovering all attack surfaces and give security warnings. Hence, they do not conduct sink-based analysis as ours. As a result, they issue many potential ECV warnings, but it is hard to confirm the true ECVs that have security impacts.

Another category of methods performs sink-based flow analysis to identify vulnerabilities more accurately. Specifically, Woodpecker [4] and DroidChecker [12] leverage path reachability analysis to detect capability or permission leaks, which belong to our `VS_Direct` ECV category except for the non-ECV vulnerabilities on unauthorized Intent receipt. SEFA [14] further extends this idea by adopting content leak detection [5] in exposed content providers. CHEX [13], on the other hand, employs inter-procedural and context-sensitive dataflow techniques to locate suspicious ECV flows terminating at the data sinks (i.e., our `VS_Input` and `VS_Public` APIs).

Similar to other sink-based analysis systems, ECVDetector also conducts path reachability and dataflow analysis. However, unlike previous works, these two kinds of analysis techniques are selectively combined together and driven by the classified categories of VSinks in our approach. In particular, we take the backward dataflow analysis, instead of forward dataflow analysis used in previous works, for adapting to more categories of sinks, such as the `VS_DirectByParam` category. These categorized VSinks are obtained by our systematic VSink selection strategy, which also assists ECVDetector to cover more ECVs that are not addressed by previous work. Additionally, we further

design a semi-automated guided analysis and system-only broadcast checking capability to efficiently exclude some false positives.

Besides detection techniques, several defense methods are proposed to mitigate ECV issues. In general, they are dynamic enforcing systems, which require to modify Android source code. They propose to check IPC call chains [63, 64], or even other channels like sockets and files [65]. On the other hand, Kantola et al. [66] attempt to automatically reduce unnecessary exposed surfaces. Moreover, mandatory access control is tailored to Android [67, 68], and it could block ECV attacks with appropriate policies. Quite recently, AppSealer [48] aims to automatically generate patches for preventing attacks to exploit ECVs. All these works are complementary to our work, since they can be applied to prevent attackers from exploiting our discovered ECVs.

5.3 Android Dynamic Testing

Besides our FileCross testing system, there are a number of other Android dynamic testing systems proposed for various purposes. Systems from the software engineering community aim at improving the app test rates by covering more code paths (e.g., [31, 32, 69]) with lower costs [70] and in more flexible ways [71]. In contrast, systems for security testing focus on adding more dedicated components, such as taint tracking in [29], fingerprint generator in [30], and pre-performed static analysis in [27]. In our case, we also embed an EBI scoring module and two dedicated components (i.e., the Attack Executer and Web Receiver) into our system, making it the first system for detecting the `file://` vulnerabilities in Android browsers.

To perform the effective security validation, many dynamic systems choose to instrument or modify Android source codes. The instrumentation can occur in different levels of Android. Some are implemented in the kernel [72], some modify the Android framework [73], and most choose to instrument both kernel and framework [29, 74]. Among them, a notable work is TaintDroid [74] that tracks all sensitive information flow from Dalvik virtual machine to Android kernel. Compared to these efforts, our dynamic testing leverage server-side programs to validate security, instead of instrumenting Android codes.

Another related work is the dynamic tainted-analysis approach [75] proposed by Newsome et al. We emphasize that although it has potential to detect the triggered FileCross attack behaviors, it cannot be used to discover the `file://` vulnerabilities like our system. Such taint-based tracking can replace the role of our server-side validation program

(i.e., Web Receiver). But it is still lack of another two components (i.e., Commander and Attack Executer) in our system.

5.4 Sink Selection in App Analysis

Many app analysis tools rely on sinks to perform their individual analysis. However, most of their sinks are selected manually [4, 5, 74, 76, 77]. Some works make use of API-to-permission mappings [39, 40] to obtain their target sinks, such as [13, 78, 79]. However, due to the lack of systematic strategy and flexible rules like ours, it is not easy for them to systematically extract appropriate sinks and filter useless ones from all candidate mappings. Moreover, we craft centralized rules to adopt the privileged APIs not covered by existing mappings, as well as other non-privileged APIs like database APIs. Another related work is SuSi [17], which employed machine learning techniques to select data sinks for privacy leak detection. However, we cannot use SuSi to select our VSinks because of the different selection motivation and that we also consider non-data sinks.

There are also some related works on our semi-automated VSink classification. For example, SuSi and CHEX also define categories for their selected sinks, but their categories are not meant for capturing different analysis requirements like ours. In fact, the categories in SuSi are only the API types (e.g., Network and File sinks). Similarly, CHEX defines three sink tags for just differentiating different data sinks. In addition, both DroidChecker [12] and AdRisk [79] separate their sinks into two categories for analysis, but they are not fully aimed at detecting ECVs. Thus they not sufficient for all analysis requirements in ECV detection. Indeed, one of their categories is for detecting either unauthorized Intent receipt or privacy leak.

Chapter 6

Conclusion and Future Work

6.1 Concluding Remarks

In this thesis, we studied two important security problems in the Android platform. One is the insecurity of `file://` support in browser apps, and the other is the insecurely exposed components in Android apps. We analyzed the attack condition of these insecure practices. We further designed and implemented analysis techniques (based on dynamic and static analysis) to detect these two classes of vulnerabilities.

For analyzing the `file://` vulnerabilities, we proposed an automated browser testing system that can launch, validate, and characterize FileCross attacks in Android browsers. This system employs a lightweight but effective scoring mechanism to identify exposed browsing interfaces and the pre-defined cooperation patterns (between client exploiters and server validators) for achieving automatic attack validation and characterization. By testing browser apps using our system, we found that the `file://` vulnerabilities are prevalent in Android browsers. More than half of our tested 115 browsers from Google Play were found vulnerable. A further detailed analysis gave more insights into the current browser practices, such as exposed browsing interfaces and allowing `file://` access to private file zones. Our vulnerability reports also helped around ten developers patch their vulnerable browsers promptly. For one browser, our system helped discover that their first patch failed to block the vulnerability.

To systematically tackle the ECV detection problem, we presented a new sink-driven approach. This approach includes a systematic strategy for VSink selection and classification, and a general detection method to identify potential ECVs in Android apps. We implemented our sink-driven approach in a tool called ECVDetector. We successfully identified a total of 49 vulnerable apps across all four ECV categories in the top

1K Android apps. Among them are the very popular Clean Master (over 200 million installs) and GO SMS Pro (over 75 million installs).

6.2 Future Research

6.2.1 Detecting `file://` Vulnerabilities in Non-browser Apps

Our system currently focuses on detecting `file://` vulnerabilities in Android browsers. However, the FileCross attacks may also exist in other kinds of apps that use web engine APIs, as explained in Section 3.5. In addition, we have identified other types of file leak attacks in our on-going research [80].

Detecting the `file://` vulnerabilities in non-browser apps will pose more challenges. First, we have to incorporate more advanced static analysis techniques to identify exposed browsing interfaces which may not have clear EBI patterns. The taint analysis that tracks external inputs to WebView sink APIs (e.g., `loadUrl()`) will be helpful. Second, unlike normal browsers that only accept target URLs, triggering browsing interfaces in non-browser apps may encounter more constraint dependencies. How to resolve these constraints and generate valid inputs is a big challenge. In particular, some constraints arise from non-input conditions (e.g., system or login conditions), thus requiring appropriate app driven techniques (e.g., [71, 81]) to satisfy these conditions.

6.2.2 Automatically Generating Exploits for Validating ECVs

In the second study, we proposed a sink-driven approach based on static analysis to detect ECVs. However, a manual validation is still required for confirming the true vulnerabilities. A straightforward idea to address this limitation is to automatically generate exploits and validate them. There are some preliminary work in this direction, such as [50] and [82]. However, we have identified four challenges that the state-of-art techniques do not address. The details of these challenges were presented in an industry hacker conference [83]. Hence, our future work is to develop a more solid exploit generation technique that solves these challenges.

Appendix A

Excerpts of Developers' Responses

Besides acknowledging our reports, developers of Baidu Browser are also interested in our automated testing system, and their feedbacks are as follows.

> "*Thank you for your reporting. We confirm that those three vulnerabilities affect Baidu Browser inter version, but do not affect its Chinese version. Please provide us your contact address so we can send a gift for your nice work.*"
> "*I am security architect in Baidu Mobile-App-Team. your work is really valuable for us. further, please also scan our Baidu relative apps, ...*"
> "*Just as you say, the tablet version also suffers this vulnerability. We will fix it soon, and give you the patched version. ... We will send a gift to you for your excellent work.*" – Responses from Baidu Browser

Some developers keep us updated about their process working on the patches, such as the vendors of InBrowser and Kids Safe Browser.

> "*We're very grateful for the detailed error-report and our engineers are working on the issue as we speak. We'll publish those changes silently in our Beta stream to start with and then publish publicly within a couple of weeks.*" – Responses from InBrowser

Moreover, we notice some individual developers are more responsible for their security issues than some big companies. For example, the student developer of Lightning Browser and the individual developer of Easy Browser always notify us of their patching updates.

"*I'm a one man development team, so I handle everything. I'm really interested in the details of these vulnerabilities. P.S. Unfortunately, I can't offer any monetary compensation for discovering the vulnerabilities since I'm just a poor student and this browser is a side project.*"
"*Ah, thanks for the clarification, Daoyuan. I'll see about what I can do.*"
"*I have modified it to block all external requests to load file urls, which should block the vulnerabilities.*" – Responses from Lightning Browser

Bibliography

[1] Number of available Android applications. http://www.appbrain.com/stats/number-of-android-apps.

[2] Sarah Perez. iTunes app store now has 1.2 million apps, has seen 75 billion downloads to date. http://techcrunch.com/2014/06/02/itunes-app-store-now-has-1-2-million-apps-has-seen-75-billion-downloads-to-date/ 2014.

[3] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. IEEE Symposium on Security and Privacy*, 2012.

[4] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proc. ISOC NDSS*, 2012.

[5] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in Android applications. In *Proc. ISOC NDSS*, 2013.

[6] Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[7] Takeshi Terada. Chrome for Android download function information disclosure. https://code.google.com/p/chromium/issues/detail?id=144820, 2013.

[8] Takeshi Terada. Chrome for Android bypassing SOP for local files by symlinks. https://code.google.com/p/chromium/issues/detail?id=144866, 2013.

[9] Takeshi Terada. Mfsa 2013-84: Same-origin bypass through symbolic links. http://www.mozilla.org/security/announce/2013/mfsa2013-84.html, 2013.

[10] Daoyuan Wu and Rocky Chang. Analyzing Android browser apps for file:// vulnerabilities. In *Proc. Springer Information Security Conference (ISC)*, 2014.

[11] Android Application Components. http://developer.android.com/guide/components/fundamentals.html#Components.

[12] Patrick Chan, Lucas Hui, and S.M. Yiu. DroidChecker: Analyzing Android applications for capability leak. In *Proc. ACM WiSec*, 2012.

[13] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proc. ACM CCS*, 2012.

[14] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on Android security. In *Proc. ACM CCS*, 2013.

[15] Kun Yang, Lujue Zhou, Yongke Wang, Jianwei Zhuge, and Haixin Duan. IntentFuzzer: Detecting capability leaks of Android applications. In *Proc. ACM AsiaCCS*, 2014.

[16] Daoyuan Wu, Xiapu Luo, and Rocky Chang. A sink-driven approach to detecting exposed component vulnerabilities in android apps. *CoRR*, abs/1405.6282, 2014. URL http://arxiv.org/abs/1405.6282.

[17] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proc. ISOC NDSS*, 2014.

[18] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proc. ACM CCS*, 2007.

[19] Html DOM document objects. http://www.w3schools.com/jsref/dom_obj_document.asp.

[20] XMLHttpRequest. http://www.w3.org/TR/XMLHttpRequest/.

[21] Jonathan Aldrich. 15-819 M: Program Analysis. http://www.cs.cmu.edu/~aldrich/courses/15-819M-10sp/, 2010.

[22] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. Data flow analysis theory and practice. http://www.cse.iitb.ac.in/~uday/dfaBook-web/, 2009.

[23] Android category browsable. http://developer.android.com/reference/android/content/Intent.html#CATEGORY_BROWSABLE.

[24] Android. Intents and Intent Filters. http://developer.android.com/guide/components/intents-filters.html.

[25] Erika Chin, Adrienne Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proc. ACM MobiSys*, 2011.

[26] MonkeyRunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[27] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proc. ISOC NDSS*, 2014.

[28] Selenium - web browser automation. http://docs.seleniumhq.org/.

[29] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic security analysis of smartphone applications. In *Proc. ACM CODASPY*, 2013.

[30] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. NetworkProfiler: Towards automatic fingerprinting of Android apps. In *Proc. IEEE INFOCOM*, 2013.

[31] Saswat Anand, Mayur Naik, Mary Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proc. ACM FSE*, 2012.

[32] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *Proc. ACM FSE*, 2013.

[33] Roee Hay. Mfsa 2014-33: File: protocol links downloaded to SD card by default. http://www.mozilla.org/security/announce/2014/mfsa2014-33.html, 2014.

[34] Takeshi Terada. Facebook for Android - information diclosure vulnerability. http://seclists.org/bugtraq/2013/Jan/27, 2013.

[35] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proc. ACM CCS*, 2013.

[36] Norm Hardy. The confused deputy: (or why capabilities might have been invented). In *ACM SIGPOS Operating Systems Review*, 1988.

[37] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proc. ACM ACSAC*, 2012.

[38] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Proc. Usenix Security*, 2013.

[39] Adrienne Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proc. ACM CCS*, 2011.

[40] Kathy Au, Yi Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android permission specification. In *Proc. ACM CCS*, 2012.

[41] Patrick Lam, Eric Bodden, Ondej Lhotk, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[42] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. In *Proc. ACM OOPSLA*, 2008.

[43] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java Virtual Machine Specification, Java SE 7 Edition. http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf, 2012.

[44] Tim Strazzere. Slowing down Android reverse engineers. http://files.meetup.com/3970892/SlowingReversers.pdf, 2013.

[45] Karim Elish, Danfeng Yao, Barbara Ryder, and Xuxian Jiang. A static assurance analysis of Android applications. http://people.cs.vt.edu/~danfeng/papers/user-intention-PA-2013.pdf, 2013.

[46] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: Scalable and accurate zero-day Android malware detection. In *Proc. ACM MobiSys*, 2012.

[47] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for Android application. In *Technical Report TUD-CS-2013-0113*, 2013.

[48] Mu Zhang and Heng Yin. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proc. ISOC NDSS*, 2014.

[49] Amiya Maji, Fahad Arshad, Saurabh Bagchi, and Jan Rellermeyer. An empirical study of the robustness of inter-component communication in Android. In *Proc. IEEE DSN*, 2012.

[50] Jiagui Zhong, Jianjun Huang, and Bin Liang. Android permission re-delegation detection and test case generation. In *Proc. IEEE International Conference on Computer Science and Service System*, 2012.

[51] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the Android system. In *Proc. ACM ACSAC*, 2011.

[52] Erika Chin and David Wagner. Bifocals: Analyzing webview vulnerabilities in Android applications. In *Proc. Springer WISA*, 2013.

[53] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *Proc. ISOC NDSS*, 2014.

[54] Zhen Wang, Felix Lin, Lin Zhong, and Mansoor Chishtie. Why are web browsers slow on smartphones? In *Proc. ACM HotMobile*, 2011.

[55] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. How far can client-only solutions go for mobile browser speed? In *Proc. ACM WWW*, 2012.

[56] Zhen Wang, Felix Lin, Lin Zhong, and Mansoor Chishtie. How effective is mobile browser cache? In *Proc. ACM Workshop S3*, 2011.

[57] Feng Qian, Kee Quah, Junxian Huang, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Web caching on smartphones: Ideal vs. reality. In *Proc. ACM MobiSys*, 2012.

[58] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Singh. Who killed my battery: Analyzing mobile browser energy consumption. In *Proc. ACM WWW*, 2012.

[59] Michael Butkiewicz, Zhe Wu, Shunan Li, Pavithra Murali, Vagelis Hristidis, Harsha Madhyastha, and Vyas Sekar. Enabling the transition to the mobile web with websieve. In *Proc. ACM HotMobile*, 2013.

[60] Chaitrali Amrutkar, Patrick Traynor, and Paul Oorschot. An empirical evaluation of security indicators in mobile web browsers. In *IEEE Trans. on Mobile Computing*, 2013.

[61] Chaitrali Amrutkar, Kapil Singh, Arunabh Verma, and Patrick Traynor. Vulnerableme: Measuring systemic weaknesses in mobile browser security. In *Proc. Springer ICISS*, 2012.

[62] Lucas Davi, Alexandra Dmitrienko, Ahmad Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Proc. Springer ISC*, 2010.

[63] Adrienne Felt, Helen Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proc. Usenix Security*, 2011.

[64] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. USENIX Security*, 2011.

[65] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *Proc. ISOC NDSS*, 2012.

[66] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing attack surfaces for intra-application communication in Android. In *Proc. ACM SPSM*, 2012.

[67] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing flexible mac to Android. In *Proc. ISOC NDSS*, 2013.

[68] Sven Bugiel, Stephan Heuser, and Ahmad Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. Usenix Security*, 2013.

[69] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proc. ACM OOPSLA*, 2013.

[70] Wontae Choi, George Necula, and Koushik Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proc. ACM OOPSLA*, 2013.

[71] Shuai Hao, Bin Liu, Suman Nath, William Halfond, and Ramesh Govindan. PUMA: Programmable UI-automation for large scale dynamic analysis of mobile apps. In *Proc. ACM MobiSys*, 2014.

[72] Thomas Blasing, Leonid Batyuk, Aubrey Schmidt, Seyit Camtepe, and Sahin Albayrak. An Android application sandbox system for suspicious software detection. In *Proc. IEEE MALWARE*, 2010.

[73] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent Freeh. Taming information-stealing smartphone applications (on Android). In *Proc. Springer TRUST*, 2011.

[74] William Enck, Peter Gilbert, Byung Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. Usenix OSDI*, 2010.

[75] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. ISOC NDSS*, 2005.

[76] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren't the Droids You're Looking For: Retroffiting Android to protect data from imperious applications. In *Proc. ACM CCS*, 2011.

[77] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proc. ISOC NDSS*, 2011.

[78] Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. Springer TRUST*, 2012.

[79] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. ACM WiSec*, 2012.

[80] Daoyuan Wu and Rocky Chang. Indirect file leaks in mobile applications. In *Proc. IEEE Mobile Security Technologies (MoST)*, 2015.

[81] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *Proc. Usenix Security*, 2014.

[82] Li Li, Alexandre Bartel, Jacques Klein, and Yves Traon. Automatically exploiting potential component leaks in Android applications. In *Proc. IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2014.

[83] Daoyuan Wu. On the feasibility of automatically generating Android component hijacking exploits. https://github.com/daoyuan14/ComponentHijackingExploit.