Towards Understanding Android System Vulnerabilities: Techniques and Insights

Daoyuan Wu*
Singapore Management University
dywu.2015@smu.edu.sg

Debin Gao
Singapore Management University
dbgao@smu.edu.sg

Eric K. T. Cheng
The Hong Kong Polytechnic University
kam-to.cheng@connect.polyu.hk

Yichen Cao
SOBUG, ShenZhen, China
caoyichen@sobug.com

Jintao Jiang
SOBUG, ShenZhen, China
jiangjintao@sobug.com

Robert H. Deng
Singapore Management University
robertdeng@smu.edu.sg

ABSTRACT

As a common platform for pervasive devices, Android has been targeted by numerous attacks that exploit vulnerabilities in its apps and the operating system. Compared to app vulnerabilities, system-level vulnerabilities in Android, however, were much less explored in the literature. In this paper, we perform the first *systematic* study of Android system vulnerabilities by comprehensively analyzing all 2,179 vulnerabilities on the Android Security Bulletin program over about three years since its initiation in August 2015. To this end, we propose an automatic analysis framework, upon a hierarchical database structure, to crawl, parse, clean, and analyze vulnerability reports and their publicly available patches. This framework includes (i) a lightweight technique to pinpoint the affected modules of given vulnerabilities; (ii) a robust method to study the complexity of patch code; and most importantly, (iii) a similarity-based algorithm to cluster patch code patterns. Our clustering algorithm first extracts patch code's essential changes that not only concisely reflect syntactic changes but also keep important semantics, and then leverages affinity propagation to automatically generate clusters based on their pairwise similarity. It allows us to obtain 16 vulnerability patterns, including six new ones not known in the literature, and we further analyze their characteristics via case studies. Besides identifying these useful patterns, we also find that 92% Android vulnerabilities are located in the low-level modules (mostly in native libraries and the kernel), whereas the framework layer causes only 5% vulnerabilities, and that half of the vulnerabilities can be fixed in fewer than 10 lines of code each, with 110 out of 1,158 cases requiring only one single line of code change. We further discuss the implications of all these results. Overall, we provide a clear overview and new insights about Android system vulnerabilities.

*The idea was proposed by this author, and partial of his work was performed while at SOBUG (https://sobug.com/) during a research internship as a vulnerability analyst.

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329831

CCS CONCEPTS

• **Security and privacy → Mobile platform security**;

KEYWORDS

Android Security; System Vulnerability; Patch Code Clustering

ACM Reference Format:
Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. 2019. Towards Understanding Android System Vulnerabilities: Techniques and Insights. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3321705.3329831

1 INTRODUCTION

Android has become the most popular system for pervasive devices over years, with a global market share of smartphones at over 80% [8]. As more and more attacks are targeting at Android by exploiting vulnerabilities in its apps and the system [7, 23, 61, 75], detecting and analyzing Android vulnerabilities has been an emerging topic in Android security research. Compared to app vulnerabilities that have been extensively studied (e.g., [22, 24, 25, 27, 30, 34, 42, 48, 54, 58, 66–69, 72, 79, 80]), system-level vulnerabilities in Android, however, were much less explored in the literature (mainly about framework-layer vulnerabilities, e.g., [16, 37, 60, 64]). This could be due to the difficulty of understanding low-level system vulnerabilities and the lack of analysis resources.

The recent arise of bug bounty programs gives researchers a new source to systematically analyzing vulnerabilities. For example, Finifter et al. [29] performed the first empirical study of vulnerability rewards programs (VRP) using the Chrome and Firefox VRPs, and Zhao et al. [76] measured the vulnerability reports submitted by white-hats on the Hackerone and Wooyun vulnerability platforms. Android also has its own bug bounty program called the Android Security Bulletin program. A recent study [52] utilized the Bulletin resource to analyze Android system vulnerabilities; however, it relied on significant manual effort to measure 660 vulnerabilities only for metadata and statistical results. Moreover, only text information from corresponding CVE (Common Vulnerabilities and Exposures) reports was analyzed, while the patch was left not mined.

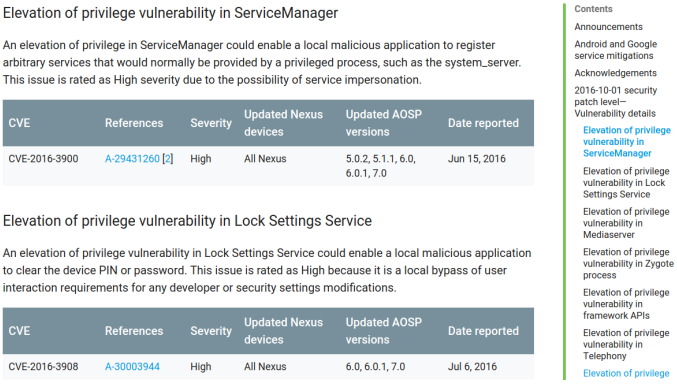
In this paper, we aim to fill the current gap in understanding Android system vulnerabilities by performing the first *systematic* study that covers all 2,179 vulnerabilities and their 1,349 publicly available patches on the Android Security Bulletin program from

its initiation in August 2015 to our analysis launched in June 2018. To make such scale a study and to easily adapt to larger datasets in the future, it is critical to adopt a systematic methodology with manual efforts involved only for configuring the analysis and interpreting the results. Fortunately, with structured Bulletin reports, we are able to propose such an automatic analysis framework that can crawl, parse, clean, and analyze vulnerability reports and their patches. Specifically, it builds upon a hierarchical database to store all the text and code information of each Android vulnerability in an organized and searchable structure, and the major novelty lies in its three analyzers for the analysis of vulnerable modules, patch code complexity, and vulnerability patterns. In particular, how to automatically cluster vulnerability patterns from a number of initially irrelevant code fragments (i.e., contiguous lines of code [40]) is the key challenge. We now elaborate these three analyzers and the corresponding analysis results.

In the first analyzer, we classify vulnerabilities by different Android modules to shed light on the system modules that are most susceptible and thus require more security attention. Unlike the prior work [52] that employs manual analysis, we propose a lightweight technique that leverages two useful features of Android Bulletin reports (see §3.2 for details) to effectively pinpoint the affected modules of given vulnerabilities. With this analyzer, we successfully obtain the layered map of vulnerable Android modules, and find that 92% of the Android vulnerabilities are located in low-level modules that are mainly coded in C/C++, especially native libraries and kernel drivers. In contrast, the framework and application layers contribute to only 5% and 2.5% vulnerabilities, respectively. Moreover, the media, Wi-Fi, and telephony related modules introduce hundreds of vulnerabilities across different layers, making them highly risky. We also perform more in-depth study on code with a large number of vulnerabilities, e.g., `MPEG4Extractor.cpp` in the `libstagefright` media library that appeared in 26 distinct patches.

Secondly, we present a robust method to study the complexity of patch code, in which we extract the "real" patch diff code by excluding not only the auxiliary code lines (e.g., the `include`/`import` and the comment statements) but also the test code that is associated with patches. We analyze the complexity of diff code extracted at both the file and the code line granularity. Results show that a significant portion of the Android vulnerabilities involve non-complex fixes, with 60% requiring only one file change and with 50% fixable in fewer than 10 lines of code. This indicates that many Android vulnerabilities are likely implementation bugs.

Lastly, we propose a similarity-based algorithm to automatically cluster Android patch code patterns, and reveal system developers' common coding mistakes that lead to vulnerabilities. Note that this task is different from the classic code clone detection problem [17, 39, 40, 43, 44, 49–51, 71] because our goal of clustering similar patches is about finding similar "changes" that involve four pieces of code per pair of patches, whereas code clone detection focuses only on two pieces of "original" code per comparison. Hence, we design a new algorithm specifically for similar patch clustering. We first extract diff code fragments' essential changes and express each such change into one code text. We then generate a similarity matrix by calculating these code texts' pairwise similarity, and further leverage affinity propagation [31] to automatically generate



Elevation of privilege vulnerability in ServiceManager

An elevation of privilege in ServiceManager could enable a local malicious application to register arbitrary services that would normally be provided by a privileged process, such as the system_server. This issue is rated as High severity due to the possibility of service impersonation.

CVE	References	Severity	Updated Nexus devices	Updated AOSP versions	Date reported
CVE-2016-3900	A-29431260 [2]	High	All Nexus	5.0.2, 5.1.1, 6.0, 6.0.1, 7.0	Jun 15, 2016

Elevation of privilege vulnerability in Lock Settings Service

An elevation of privilege vulnerability in Lock Settings Service could enable a local malicious application to clear the device PIN or password. This issue is rated as High because it is a local bypass of user interaction requirements for any developer or security settings modifications.

CVE	References	Severity	Updated Nexus devices	Updated AOSP versions	Date reported
CVE-2016-3908	A-30003944	High	All Nexus	6.0, 6.0.1, 7.0	Jul 6, 2016

Contents

Announcements
Android and Google service mitigations
Acknowledgements
2016-10-01 security patch level—Vulnerability details
Elevation of privilege vulnerability in ServiceManager
Elevation of privilege vulnerability in Lock Settings Service
Elevation of privilege vulnerability in Mediaserver
Elevation of privilege vulnerability in Zygote process
Elevation of privilege vulnerability in framework APIs
Elevation of privilege vulnerability in Telephony
Elevation of privilege

Figure 1: A sample webpage of Android Security Bulletin website.

clusters according to the matrix. Finally, patterns are abstracted from top similar cases within clusters.

By running this algorithm, we obtain 83 initial clusters of which we quickly filter out 50 small-size ones as they contain only fewer than 10 code fragments each and actually do not exhibit evident security-oriented patterns. Out of the remaining 33 clusters, 28 (84.8%) are associated with certain patterns, with 19 clusters for security-oriented patterns and 9 clusters for non-security-related patterns. We eventually extract 16 vulnerability patterns from 19 security-oriented clusters. They include not only traditional patterns, e.g., overflow and uninitialized data, but also six new ones not known in the literature, such as mis-retrieving Android service by reference and inconsistent Android Parcelable serialization. We then analyze their characteristics by performing case studies.

Furthermore, we discuss four implications of our analysis results. Besides quantitatively pointing out the seriousness of Android system vulnerabilities and the necessity of adopting them into future threat models, our results can help system developers avoid making similar mistakes in the same module and guide program analysis techniques for automatic vulnerability detection.

2 ANDROID SECURITY BULLETIN PROGRAM

Android Security Bulletin program (https://source.android.com/security/bulletin/) started in August 2015 and is updated monthly. Figure 1 shows a sample page (October 2016) of its website. It lists all vulnerabilities that were fixed and made public in a calendar month. As shown on the right-hand side of Figure 1, it first gives an outline of the vulnerabilities in different modules, such as the service manager, the lock setting service, and the media server. For each module, it further lists the detailed vulnerability information, including CVE, the Android vulnerability ID (AID), the vulnerability severity, and the updated Android versions. In particular, the URL of AID actually points to the webpage of the corresponding patch code, and we call such URL "the patch URL".

3 METHODOLOGY

Our goal is to conduct a systematic study of Android system vulnerabilities by comprehensively analyzing all vulnerabilities on the Android Security Bulletin program from its initiation in August 2015 to our analysis launched in June 2018. To minimize manual

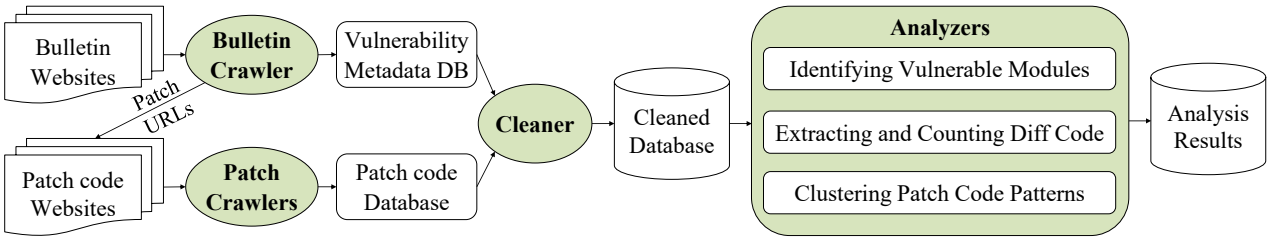


Figure 2: The workflow of our automatic analysis framework for Android system vulnerability reports and their patches.

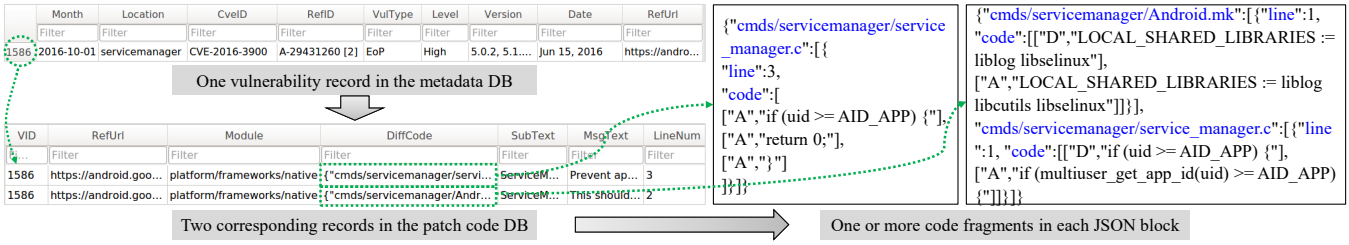


Figure 3: An example illustrating our hierarchical database structure for storing all the text and code information of each Android vulnerability in an organized and searchable structure.

analysis as in previous work [41, 52], we propose the first analysis framework that can automatically crawl, parse, clean, and analyze Android bulletin reports and their publicly available patches. With such a framework, manual efforts are required only for configuring the analysis and interpreting the results (e.g., abstracting patterns from automatically generated clusters). It can also easily adapt to larger datasets in the future with evolving analysis results.

Overview. Figure 2 shows the overall workflow of our analysis framework. It consists of a bulletin crawler, a patch crawler, a cleaner, and three analyzers. All these components are written in Python, with 1,230 lines of code excluding the library support, e.g., Selenium [14] for crawlers and Jellyfish [9] for string similarity metrics. We summarize the functionality of each component as follows:

- *Bulletin crawler* is responsible for crawling the basic information of every vulnerability on Android Bulletin website. The information crawled includes CVE (Common Vulnerability Entry) id, vulnerability type, vulnerability severity, and several other meta information. One important meta information is the URLs of each vulnerability's patch code, which will be further used by the patch crawler. All this information is parsed directly from the bulletin website's HTML files and saved into a vulnerability metadata database.
- *Patch crawler* takes patch URLs as input to crawl the patch code websites and then builds a patch code database. Since there are several types of patch code websites for Android bulletin vulnerabilities, we build all corresponding patch crawlers. The HTML parsing here is more complicated than that in the bulletin crawler, because extracting diff code of patches into organized structures is difficult; see details in §3.3.
- *Cleaner* is designed for cleaning the raw database, especially the text information in the vulnerability metadata. This is because Android bulletin reports are still manually created and thus could come with disorganized text. For example, the "EoP" vulnerability type could be represented as "elevation-of-privilege-vulnerability", "elevation_of_privilege", and even "eopv". Moreover, although the majority of patch URLs are correct, a few of them are outdated (e.g., "commit/?id=" instead of "patch/?id="), or contain unescaped characters (e.g., "%2F") and redundant characters (e.g., redundant "/" in "la//"). Cleaner cleans all this misconfigured information in a one-time manner.
- *Analyzers* take the cleaned database as input and output analysis results. Besides the vulnerability metadata analysis, we have designed three analyzers (as shown in Figure 2) to support vulnerable module analysis, patch code complexity analysis, and patch code pattern analysis. We will illustrate them in subsequent subsections.

Challenges. Since we are the first to study Android system vulnerabilities in an automatic fashion, there are some unique challenges. Notably, our three analyzers face the challenges on *effectively* pinpointing vulnerability modules (§3.2), *robustly* measuring patch code complexity (§3.3), and *automatically* clustering vulnerability patterns (§3.4), respectively. Before explaining these challenges in detail and our methods of overcoming them, we first show in §3.1 how we store all the text and code information of each Android vulnerability in an organized and searchable structure.

3.1 Designing a Hierarchical Database Structure

The first challenge is in representing vulnerabilities' text and code information in a way that analysts can directly make SQL queries to retrieve the desired vulnerability information without writing additional scripts. This is challenging because we notice that 1) an

Android vulnerability might be associated with several patches; 2) one patch may include several affected code files; 3) one code file may contain multiple patched code fragments; and 4) one code fragment usually covers several code lines.

We propose to build a hierarchical database structure and use a carefully designed nested JSON [1] format to represent patched code in a hierarchical way. Figure 3 shows the high-level picture of our hierarchical database structure using a specific vulnerability example (CVE-2016-3900). We first use a database table to record all the metadata of this vulnerability, as mentioned earlier in the bulletin crawler component. Since CVE-2016-3900 involves two patches, we then save the information of both patches in the patch code database and point them to the corresponding row id (1586 in this example) in the metadata database. Finally, we design a nested JSON format to represent all diff code of each patch. In this way, we use only one database field ("DiffCode" in Figure 3) to cover the patch code and avoid having to dynamically extend the database. In each JSON, we use code name as the JSON key and use nested arrays to record each code and their code fragments. Figure 3 shows the two JSON examples of CVE-2016-3900, one with one code file and the other with two. Here all the three pieces list only one code fragment each, but it is possible that multiple fragments occur in a single patch code.

With this hierarchical database structure, we are able to compose complex search of the vulnerability database directly in SQL queries. Listing 1 demonstrates one query example that counts the median number of code fragments in each patched code file. We use SQLite's JSON1 extension [10] to handle nested JSON. For example, in Listing 1, we use the `json_each()` API to decompose each "DiffCode" field into a key-value row, where the key refers to the code file name and the value is a nested array of code fragments. We thus can use the key field to exclude assembly code files and use `json_array_length(value)` to further count code fragments. In this way, we can obtain the vulnerability search results (e.g., the median number of per-file code fragments is 2) without writing dedicated scripts.

Listing 1: A SQL query example of searching the database.

```
select median(json_array_length(value))
from PatchTable, json_each(PatchTable.DiffCode)
where PatchTable.DiffCode like '{%}'
and key not like '%.s';
```

3.2 Identifying Vulnerable Modules

Classifying vulnerabilities by different Android modules can shed light on the system modules that are most susceptible and thus require more security attention. Therefore, we include a dedicated analyzer in our analysis framework to identify vulnerable Android modules. However, it is challenging because there is no clear module information in CVE reports. As a result, previous work [52] employed two manpower to manually inspect the 660 Android vulnerabilities in their dataset.

We propose a lightweight technique that leverages two useful features of Android Bulletin reports to locate the affected modules of given vulnerabilities. The first is the patch code paths for those with publicly available patches, which could imply the module information. However, the full code paths are often too detailed, e.g., `platform/system/bt/bta/dm/bta_dm_act.cc` in CVE-2018-9355. Fortunately, we found that the Android Security team has embedded the high-level module path information in patch URLs. For example, the patch URL of CVE-2018-9355 is `https://android.googlesource.com/platform/system/bt/+/99a263`, in which we can extract the path `platform/system/bt` (meaning the Bluetooth stack according to the patch code website [13]) as the affected module.

Since around half of the vulnerabilities in our dataset have no publicly available patches, we still need to find another way to identify vulnerable modules. Moreover, the module path information of some URLs are coarse-grained, e.g., the aforementioned CVE-2016-3900 only shows `platform/frameworks/native` in its patch URL. Our technique thus leverages the second feature: the Android Bulletin webpage itself contains certain pattern that records the module information input by the Android Security team. For instance, in the bulletin webpage shown in Figure 1, we can locate the HTML field `<h3 id="eopv-in-servicemanager">` for CVE-2016-3900, where "eopv" is the vulnerability type and "servicemanager" pinpoints the module.

3.3 Extracting and Counting Diff Code

Our second analysis objective is to study the complexity of Android patch code, and it requires a *robust* method to extract the "real" patch diff code and count their line change. This is because not all modified code lines in a patch are for the vulnerability fix and some are only auxiliary, e.g., the `#include` statements in C/C++, the `import` statements in Java, and also many comment statements.

Before dealing with those auxiliary code lines, we need to extract patch code fragments (i.e., contiguous lines of code [40]) from the raw HTML files and organize them in the format shown in Figure 3. We first use Selenium [14] to locate the code diff fields [6], i.e., "add", "del", "ctx", and "hunk", in the patch HTML files. We then use "add"/"del" as the indicators to count contiguous code lines and use "ctx"/"hunk" as the stop words. In the meantime, we count the number of line changes for each code fragment as

$$countFrag = max(countAdd, countDel)$$

where *countAdd* and *countDel* are the total numbers of lines added or deleted (note that the auxiliary code lines have been excluded when performing the counting). The number of line changes in a code fragment, *countFrag*, is the maximum of the two since any single line change could contribute to both addition and deletion. With each individual *countFrag* counted in a code fragment, the line change of a file, *countFile*, is then the corresponding sum.

When counting the number of code line changes, we exclude the auxiliary code lines as follows. First, the blank lines, after striping the "+"/"-" symbols and various whitespaces, are eliminated. Second, the `include` and `import` statements in C/C++ and Java are not taken into consideration when studying the complexity of a patch. Third, we remove comment statements, some of which are not easy to be recognized. For example, we need to track forward across multiple lines to pinpoint the end of a comment block that uses /* ...*/ in C/C++/Java or <!- - ...- -!> in XML. Moreover, some comment blocks are only partially shown in the diff files (e.g., not

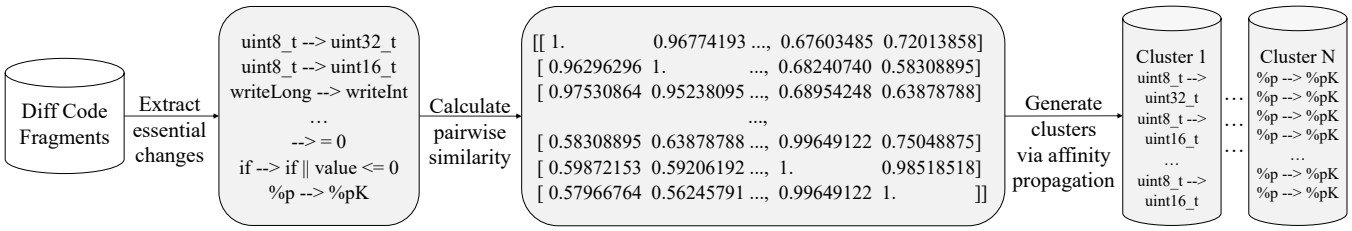


Figure 4: A high-level overview of our similarity-based algorithm to automatically generate patch code clusters.

Table 1: Examples illustrating how we extract patch code's essential changes.

ID	Diff Code	Change
C1	- dest.writeLong(mSubId); + dest.writeInt(mSubId);	writeLong --> writeInt
C2	- uint8_t len = 0; + uint32_t len = 0;	uint8_t --> uint32_t
C3	- void lim_compute_crc32(uint8_t *pDest, uint8_t *pSrc, uint8_t len); + void lim_compute_crc32(uint8_t *pDest, uint8_t *pSrc, uint16_t len);	uint8_t --> uint16_t
C4	- const sp<ICameraService>& cs = getCameraService(); + const sp<ICameraService> cs = getCameraService();	sp<ICameraService>& --> sp<ICameraService>
C5	- pr_debug('%s: ndx=%d base=%p', __func__, ctrl->ndx, ctrl->base); - pr_debug('%s: ndx=%d base=%pK', __func__, ctrl->ndx, ctrl->base);	%p --> %pK
C6	- uint64_t slotMask; + uint64_t slotMask = 0;	--> = 0
C7	- runtime.start("com.android.internal.os.RuntimeInit", args); + runtime.start("com.android.internal.os.RuntimeInit", args, zygote);	--> , zygote
C8	- return true; + return false;	return true --> return false
C9	- if(value >= ps_sps->i4_pic_size_in_ctb) + if(value >= ps_sps->i4_pic_size_in_ctb || value <= 0)	if --> if || value <= 0
C10	- #define MAX_NUM_INPUT_OUTPUT_BUFFERS 32 + #define MAX_NUM_INPUT_OUTPUT_BUFFERS 64	#define 32 --> #define 64

starting with "/*" but with "*"), which require us to track the subsequent lines for determining whether the current line is a comment statement or not.

Besides the auxiliary code lines, we found that some patches also include test code. For example, CVE-2017-13176 includes `core/tests/coretests/src/android/net/UriTest.java` [4] which should not be counted when calculating patch complexity. To remove the impact of such test code, we simply use the keyword "Test" or "test" to exclude the test code without affecting the normal one.

3.4 Clustering Patch Code Patterns

In the last and most important analyzer, we aim to automatically cluster Android patch code to reveal system developers' common coding mistake patterns. Specifically, our objective is to cluster (patch) code-level patterns, such as changes in an integer type from `uint8_t` to `uint32_t` and changes in a character of printing kernel addresses from `%p` to `%pK`. These patterns, after interpreting with security knowledge, can reflect the root causes of corresponding vulnerabilities, e.g., inappropriate usage of pointer `%p` in kernel address printing could signal information leakage. Figure 4 depicts a high-level overview of our similarity-based algorithm, which is comprised of three major steps as follows.

First, different from code clone detection approaches [17, 39, 40, 43, 44, 49–51, 71] that typically compare multiple versions of the same code piece or code from multiple software, we need to extract "diff of the diff" from code fragments. More specifically, we extract patch code' *essential changes* that not only concisely reflect syntax-level changes but also maintain important semantic information by keeping change-related tokens. Table 1 shows various examples to illustrate how we extract essential changes of patch code. For example, in code C1, we not only extract syntax-level change (i.e., `Long` to `Int`) but also keep the full token (i.e., function name in this example) to capture change-related semantic. Similarly, in code C2 and C3, the essential change we extracted is `uint8_t` to `uint16|32_t`, which is much more concise than the original diff code and also more meaningful than the syntax-only change (i.e., `8` to `16|32`). The only special handling is that we add keywords for `return`, `if`, and `define` statements (see code C8 to C10) to better maintain their semantic changes.

To express each code change into one code text, we employ a special character "-->" to represent the change process. For code fragments that are fully added or deleted, we simply use their original JSON format shown in Figure 3, which clearly marks the added or deleted code lines. Note that we currently focus only

on short code fragments, since our objective is to reveal *common* coding mistake patterns that are usually not complicated. We leave as our future work to cluster complex vulnerability patterns that involve long code fragments.

Second, with extracted code change texts, we then calculate their pairwise similarity to generate a large similarity matrix, as shown in Figure 4. Each row of this matrix is a vector of similarity scores between one code change text and all the others. The similarity score is represented as a string distance. There are multiple string distance metrics, and we tested four common ones including Jaro distance, Jaro-Winkler distance, Levenshtein distance, and Damerau-Levenshtein distance. We found that Jaro-Winkler distance is the most suitable string distance metric in our problem context — clustering using Levenshtein or Damerau-Levenshtein distance can generate only one cluster, and Jaro distance does not perform well in some situations (e.g., clustering `memset()` usages).

Third, we automatically generate patch code clusters according to the matrix. We choose affinity propagation [31] as our clustering algorithm because it does not require pre-estimation of the number of clusters as in k-means or k-medoids clustering algorithms. To obtain good clustering results, we first did tests to find affinity propagation's optimal damping factor [15] at 0.9 in our problem context. Note that such parameter tuning is simple and performed only once.

4 ANALYSIS RESULTS

In this section, we present our analysis results of Android system vulnerabilities. We first introduce the dataset and vulnerability metadata in §4.1, then describe our analysis results of vulnerable modules, patch code complexity, and patch code patterns from §4.2 to §4.4, and finally discuss their implications in §4.5.

4.1 Dataset and Vulnerability Metadata

Till we initiated the analysis in June 2018, we have collected the information of 2,179 vulnerabilities on the Android Security Bulletin program and their 1,349 publicly available patches (from 1,158 distinct vulnerabilities). These vulnerabilities include all Android vulnerabilities reported over around three years (from August 2015 to June 2018). For vulnerability clustering, we extract a total of 940 short code fragments from these 1,349 patches.

Table 2 shows four major vulnerability types and four levels of vulnerability severity that are defined by the Android Security team. Among all the vulnerability types, we can see that the EoP (elevation of privilege) is the most common one with a total of 954 (43.8%) vulnerabilities. ID (information disclosure) and the most dangerous RCE (remote code execution) rank second and third with 313 and 254 vulnerabilities, respectively. DoS (denial of service), unsurprisingly, is the least affected vulnerability type with 160 vulnerabilities. Additionally, there are 498 vulnerabilities marked as "N/A" by Google, which are because of the closed-source driver components of which the vulnerability details are not ready to be made public at the time of our crawling. Despite this, we estimate that the "N/A" type of vulnerabilities would mainly cause EoP and RCE issues by correlating with the severity (i.e., via the 402 high and 82 critical vulnerabilities in the "N/A" category).

Table 2: Vulnerability metadata: type and severity

	RCE	EoP	ID	DoS	N/A	
Critical	200	156	0	0	82	438
High	47	641	112	133	402	1,335
Moderate	6	156	197	19	14	392
Low	1	1	4	8	0	14
	254	954	313	160	498	

Notes:
RCE = Remote Code Execution; EoP = Elevation of Privilege;
ID = Information Disclosure; DoS = Denial of Service.
N/A = Not Available, due to closed-source driver components.

Regarding the vulnerability severity, high-level severity accounts for the largest proportion with 1,335 (61.3%) vulnerabilities. The critical- and the moderate-level severity hold the remaining 38% portions, with only 14 vulnerabilities being rated as at low-level severity. By correlating the severity with the vulnerability type, we further find that the critical-level severity is mostly related to RCE issues, and similarly, high severity appears the most significantly in EoP, the moderate and low severity are for ID and DoS, respectively. Such a one-to-one relationship is also almost true when correlating the vulnerability type with the severity except that most of the DoS vulnerabilities result in high-level severity.

Key Takeaway: Most of Android vulnerabilities are dangerous, with 81% (1,773/2,179) rated as the high severity or above. Moreover, most of them could lead to a serious elevation of privilege and remote code execution. These suggest that Android vulnerabilities could make severe security impacts and require better understanding.

4.2 Analysis of Vulnerable Modules

In this subsection, we present our analysis of vulnerable modules. We first depict the layered map of vulnerable Android modules in Figure 5, with the percentage counted for each Android layer and with the vulnerability number marked behind each module name. By inspecting the vulnerability percentage of different Android layers, we can observe that layers with modules mainly coded in Java (i.e., the Application Framework and the System Applications layers) have significantly fewer vulnerabilities than those mainly coded in C/C++ at 7.25% v.s. 92.75%. In particular, the Linux Kernel layer itself already accounts for 65.7% of all the 2,179 Android vulnerabilities studied, and the Native Libraries layer also holds 23.9%. Both layers introduce many third-party drivers or libraries, the code quality of which might be worse than Android's own code. Generally, there are more vulnerabilities in C/C++ code than Java due to the potential memory corruption issues (e.g., buffer overflow). This could be supported by the evidence that in our dataset of 1,349 patches, only 154 patches involve Java code while that number for C/C++ is 1,164.

We then study the vulnerable modules across different layers and obtain the following observations. First, the media-related modules are the high-risk modules from the Native Libraries layer down to the Linux Kernel, including the media framework (code in the `frameworks/av` [11]), the media libraries (e.g., `libstagefright` and `libmpeg2`), the media components in the hardware abstract layer (e.g., `hardware/qcom/media` [12]), and the sound and video

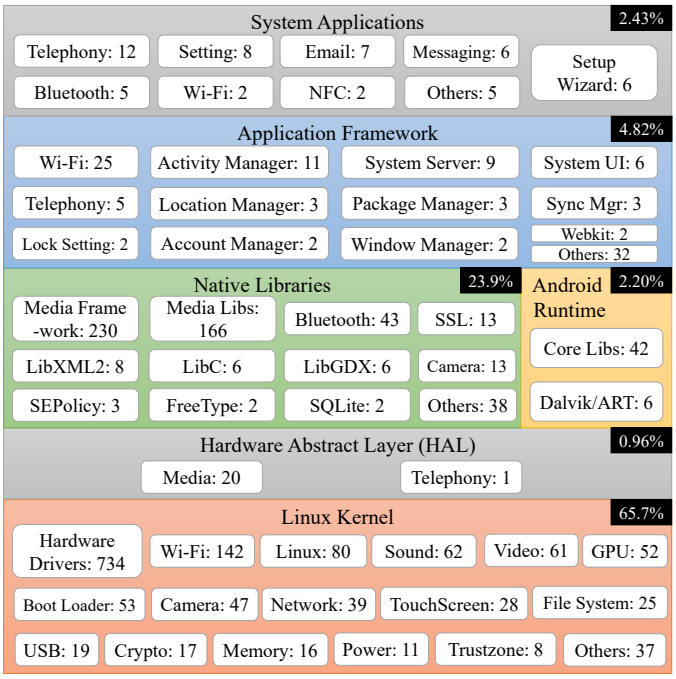


Figure 5: The layered map of vulnerable Android modules.

Table 3: The code that was frequently reported as vulnerable in terms of appearing in at least ten vulnerabilities.

Code	#
media/libstagefright/MPEG4Extractor.cpp	26
decoder/ih264d_parse_pslice.c	23
decoder/ih264d_api.c	20
decoder/ih264d_parse_slice.c	17
drivers/misc/qseecom.c	17
media/libeffects/lvm/wrapper/Bundle/EffectBundle.cpp	17
CORE/HDD/src/wlan_hdd_cfg80211.c	15
app/aboot/aboot.c	14
decoder/ihevcd_parse_headers.c	14
services/audioflinger/Effects.cpp	14
decoder/impeg2d_dec_hdr.c	13
decoder/ih264d_parse_headers.c	11
com/android/server/am/ActivityManagerService.java	11
post_proc/equalizer.c	10

drivers in the kernel. Second, the vulnerable Wi-Fi modules appear in not only the kernel layer but also the framework and application layers. In particular, the Wi-Fi driver and the Wi-Fi framework are the mostly affected module in the corresponding layers, with 142 and 25 vulnerabilities, respectively. Third, the telephony-related modules are also of high risk with 12 vulnerabilities in the application layer, five vulnerabilities in the framework, and one vulnerability in the hardware abstract layer. Additionally, some other hardware-related modules, e.g., camera, also appear in both the native libraries and the kernel.

We further take a close look at the code that was frequently reported as vulnerable. Table 3 lists the top one that appears in at least ten vulnerabilities. We have the following observations. First, a third of the 14 high-risk code is located in the "decoder" directory with six pieces of frequently vulnerable code. In particular, the "ih264d" related decoder code was affected the most, which deserves more security attention. Moreover, several media libraries were also frequently reported, e.g., the `libstagefright` and `libeffects`. In particular, the file `MPEG4Extractor.cpp` in the `libstagefright` even appears in 26 patches — the riskiest code. Besides that related to media, we find that code for WLAN (`wlan_hdd_cfg80211.c`), bootloader (`aboot.c`), and Activity Manager is also in the high-risk list. Finally, among all the listed code, only `ActivityManagerService.java` is written in Java. This provides another evidence that C/C++ code in Android could be less secure than Java code.

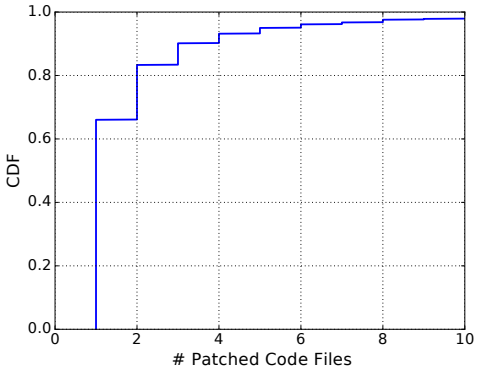


Figure 6: CDF plot of # patched code files per vulnerability.

> **Key Takeaway:** 92% of the Android vulnerabilities are located in low-level modules that are mainly coded in C/C++, especially native libraries and kernel drivers. Moreover, media, Wi-Fi, and telephony related modules are at high risk as they introduce hundreds of vulnerabilities across different layers. We also study code frequently reported as vulnerable. Overall, our analysis sheds light on susceptible Android system modules.

4.3 Analysis of Patch Code Complexity

In this subsection, we study the complexity of patch code by measuring the number of code changes required to fix each vulnerability. We use all the 1,349 patches (from 1,158 unique vulnerabilities) for analysis and draw the CDF (cumulative distribution function) plots of their patch code complexity.

We first analyze the number of code files needs to be patched for each vulnerability. Figure 6 shows the CDF plot of the number of patched code files per vulnerability. We can see that over 60% Android vulnerabilities require a code change in only one file, and this percentage goes up to over 80% if we count the vulnerabilities with no more than two files changed. This suggests that most of the Android vulnerabilities are quite dedicated and involve minimal code files to be patched. However, there are also a few vulnerabilities requiring an exceptional number of files changed, which are due to either system library updates or fixing common root causes in different files. For example, CVE-2014-9675 upgrades the FreeType library from 2.6.0 to 2.6.2, and thus adjusts a total of 112 files [2], the

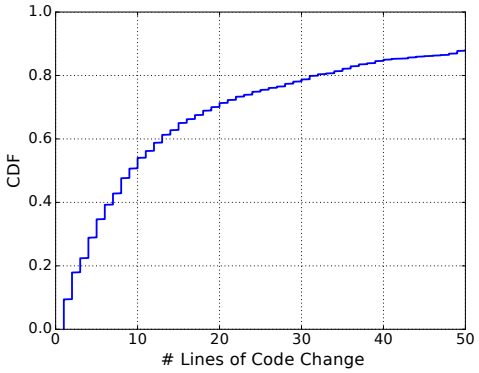


Figure 7: CDF plot of # code lines changed per vulnerability.

largest number of code files patched among all vulnerabilities. In another instance, CVE-2017-13177 adds the push-pop instructions in around 60 different ARM Neon 32-bit functions [3].

We further study at the granularity of code lines and draw the CDF of the number of code lines changed per vulnerability in Figure 7. We find that half of the vulnerabilities can be fixed in fewer than 10 lines of code, with the median being nine. Moreover, a third of the vulnerabilities are patchable with no more than five lines, and around one fifth require no more than two lines of changes. In particular, 110 out of 1,158 vulnerabilities can be patched with only one line code change. All these indicate that many Android vulnerabilities are likely implementation bugs.

> **Key Takeaway:** A significant portion of the Android vulnerabilities involves non-complex patch fixes, with 60% requiring only one file change and with 50% fixable in fewer than 10 lines of code. This indicates that many Android issues are likely implementation bugs.

4.4 Analysis of Patch Code Patterns

In this subsection, we first give a statistical overview of our clustering results to demonstrate their good quality. We then describe and analyze clustered patch code patterns in detail.

By running our clustering algorithm over a set of 940 short code fragments, we obtain 83 initial clusters, out of which we can quickly filter out 50 small-size clusters as they contain only fewer than 10 code fragments each and actually do not exhibit evident security-oriented patterns. The remaining 33 clusters contain code fragments ranging from 10 to 56 fragments each, with an average of 21 fragments. We found that these clusters are in good quality, with only five clusters not exhibiting clear patterns. In other words, 84.8% (28/33) clusters are associated with certain patterns, with 19 clusters for security-oriented patterns and 9 clusters for non-security-related patterns (e.g., declaring variables and using `#ifdef`). Out of the 19 security-oriented clusters, we obtain a total of 16 patterns with the majority corresponding to distinct ones.

Table 4 lists the detailed pattern results, sorted according to the size of the clusters. As shown in the last column of detailed cluster ID, only two clusters (cluster 21 and 1) are mapped to multiple patterns, which indicate that we can easily abstract patterns from each automatically generated cluster. Moreover, most of the patterns correspond to only one cluster, with just five patterns merged from multiple clusters each. In the second last column, we further determine whether these patterns were previously known in the literature and identify six new ones (marked with ✗). Additionally, there are three patterns not fully covered by the literature (marked with ◑).

We now explain all the 16 patterns in details. We first analyze six new patterns one by one, and then introduce two more Android-specific patterns and eight traditional patterns.

P1 (new): Kernel address leakage due to %p. This vulnerability originates from a type of security bugs exposed recently in the Linux kernel (Kernel 4.14 and earlier), where printing kernel addresses to user space using `%p` can leak sensitive information about kernel memory layout. Timely mitigation is to replace `%p` with `%pK` to print only Zeros as address (see code example C5 in Table 4). One year later in late 2017, a fundamental fix [5] was released by printing only hashed addresses via `%p`. With the pattern `%p --> %pK`, we identify a total of 28 such vulnerabilities in our entire patch code dataset of 1,158 distinct vulnerabilities.

P2 (new): Mis-retrieving Android service by reference. This vulnerability is quite specific to Android, where system processes need to retrieve various Android system services, e.g., camera service as shown in code C4. Android system developers previously obtained these services by reference (i.e., `sp<>&`); however, such service pointers can be cleared out by another thread or system Binder death callback. Therefore, a safe way is to retrieve these services by value (i.e., `sp<>`). In our patch code dataset, 10 vulnerabilities suffered from this issue.

P3 (new): Inconsistent Android Parcelable serialization. This vulnerability is also specific to Android, where structured data sharing across different processes requires serializing and deserializing custom Parcelable objects. Inconsistency happens if data types in `writeToParcel()` and `readFromParcel()` are not symmetric, and an adversary could exploit such inconsistency to elevate privileges. For example, in code C1, a long integer was written but a normal integer was read. Other data types, e.g., byte and string, could also be misused. Moreover, different control-flow branches could make it easier for developers to make mistakes, and therefore we also see fixes like adding `writeInt()` in the `else` branch. Seven inconsistent serialization bugs were identified.

P9 (new): Incomplete C++ destruction. This type of vulnerabilities appears in some Android media encoders, where the C++ destruction is not fully finished and some memory buffers could still be controlled by attackers. To make the destruction more focused and clearer, a standalone `onReset()` is added to destruct all relevant member variables.

P12 (new): Missing certain parameter, causing logic flaws. Mitigating this type of vulnerabilities requires adding certain parameters and their handling logic. For example, in code C7 (CVE-2015-3865), a new parameter called `zygote` was added. Code was also added to check fir this parameter to enable debugging only for apps forked from zygote. Detailed logic flaws in this vulnerability pattern could be different, but they all relate to the missing of certain parameters and the corresponding handling logic.

P14 (new): Forgetting to set certain variable const/transient. The last new pattern is about the use of `const` and `transient`

Table 4: Clustered 16 patch code patterns for Android system vulnerabilities (some examples can be referred to Table 1).

ID	Description	Pattern (using diff code's essential change format)	Example	Known?	Cluster ID
P1	Kernel address leakage due to using %p	%p --> %pK	C5	✗	73
P2	Mis-retrieving Android service by reference	sp<XXXService>& --> sp<XXXService>	C4	✗	21
P3	Inconsistent Android Parcelable serialization	writeLong --> writeInt OR + writeInt();	C1	✗	21, 81
P4	Mis-exported component in system apps	exported='true' --> exported='false'	–	✔ [70]	21
P5	Missing or mis-setting IF check condition	if [OLD_CONDITION] --> if NEW_CONDITION	C9	◑ [19]	63, 2
P6	Use-after-free and double free issues	+/- XXX_free();	–	✔ [33]	1
P7	Missing Android permission/UID checking	+ checkXXXPermission()/checkCallerXXX();	–	◑ [60]	1
P8	Overflow due to inappropriate #define value	#define INT1 --> #define INT2	C10	✔ [19]	74
P9	Incomplete C++ destruction	+ virtual void onReset();	–	✗	14
P10	Uninitialized data due to missing memset()	+ memset();	–	✔ [53]	50, 36
P11	Uninitialized data due to unassigned variable	VARIABLE --> VARIABLE = INIT_VALUE	C6	✔ [53]	32, 4, 15
P12	Missing certain parameter, causing logic flaws	--> , PARAMETER	C7	✗	52
P13	Overflow due to missing error case checking	+ if (CONDITION) + { return ERROR; }	–	◑ [19]	31
P14	Forgetting to set certain variable const/transient	--> const / transient	–	✗	58
P15	Integer overflow due to inappropriate INT type	uint8_t|int --> uint16|32_t|long|size_t	C2, C3	✔ [65]	7, 12
P16	Data race due to missing lock/unlock	+ XXX_lock(); + XXX_unlock();	–	✔ [26]	69

type qualifiers, where marking a variable `const` or `transient` can prevent it from being modified or initialized, respectively. For example, in CVE-2015-8967, `const` is needed to stop the system-call table being modified. In CVE-2015-3837, `transient` is used to hide the OpenSSLX509Certificate context variable and prevents it from participating in the serialization process.

P4 & P7: Two more Android-specific patterns. Besides P2 and P3, pattern P4 and P7 are also Android specific. P4 is a common vulnerability pattern in Android apps which also appears in system apps. It mistakenly exports sensitive Android components to other (potentially malicious) apps. On the other hand, P7 is about missing permission or UID (i.e., app user ID) checking, and this pattern appears in 26 vulnerabilities of our dataset, demonstrating its pervasiveness. A prior work, Kratos [60], was designed for this problem, but it can detect only inconsistent permission checking and only at framework layer.

P5 & P8 & P13 & P15: Overflow-related patterns. Now we present some traditional patterns. Hundreds of vulnerabilities in our dataset are covered by overflow-related patterns, as in pattern P5, P8, P13, and P15. The first three are about buffer or stack overflow, while P15 is on integer overflow. Most of buffer overflows are due to missing appropriate bounds checking, which could either miss or mis-set a check condition in the IF statements (see code C9 in pattern P5) or forget to handle a certain error branch (i.e., pattern P13). In the case of pattern P8, the buffer itself needs to be enlarged. Regarding integer overflow, the root cause is that inappropriate integer types are used and the fix is to replace a smaller integer type (e.g., `uint8_t`) with a larger one (e.g., `uint32_t`).

P10 & P11: Vulnerabilities due to uninitialized data. Uninitialized data is another traditional vulnerability [19], and pattern P10 and P11 cover its two scenarios. The first scenario misses using `memset()` to initialize memory buffer, and the second forgets to assign an initial value (e.g., `0` and `NULL`) to a certain data variable. These uninitialized data might be exploited to leak information about memory layout.

P6: Use-after-free and double free issues. This type of vulnerabilities is due to incorrect use of memory free functions (e.g., `osi_free()` and `kfree()`). In some vulnerabilities, such as CVE-2017-13257, a memory free function was placed at a location where the data was still in use, causing an use-after-free issue. In other cases, such as CVE-2018-9356, a memory buffer was freed two times under a certain control-flow branch, resulting in a double free issue.

P16: Data race due to missing lock/unlock. The last traditional vulnerability pattern we clustered is about data race, where lock/unlock functions (e.g., `spin_lock/unlock()` and `mutex_lock/unlock()`) are not placed to prevent race conditions in a multi-thread system such as Android.

> **Key Takeaway:** Our clustering algorithm automatically generates good-quality clusters of patch code fragments, with 84.8% clusters associated with certain patterns. We thus can extract 16 vulnerability patterns from 19 security-oriented clusters, including six new ones not known in the literature and four specific to Android. We further analyze the characteristics of these patterns via case studies.

4.5 Implications of Our Analysis Results

In this subsection, we further discuss four implications of our analysis results presented earlier.

Implication 1: *Our analysis quantitatively points out the seriousness of system-level vulnerabilities in Android.* By analyzing the severity of all 2,179 vulnerabilities in §4.1, we found that 81% of them are rated as high or critical severity. This suggests that detecting system-level issues is equally, if not more, important than app-level vulnerabilities. Indeed, a considerable portion of Android malware in the wild leveraged system vulnerabilities for root exploits [32, 55, 78]. Therefore, it is especially important for security researchers to detect and patch zero-day Android vulnerabilities ahead of hackers.

Implication 2: *The results of vulnerable modules can help system developers avoid making similar mistakes in the same module or code.* This is a further usage of our vulnerable module results beyond the statistical data presented in §4.2. Specifically, when an Android system developer or a third-party ROM maker starts to work on a particular Android module, he/she can first go through previously

reported vulnerable code examples in the same module. In particular, our module results contain detailed code file paths (e.g., Table 3 in §4.2) and their associated patches. To help developers easily retrieve such information, we are on the way of implementing a web portal to make our results browsable and searchable.

Implication 3: *Since implementation bugs are an important source of Android system vulnerabilities, it is necessary for future defense systems to adopt them into threat models.* Existing research efforts on securing Android OS have proposed mandatory access control (e.g., SEAndroid [62] and ASM [36]) and information flow control (e.g., Weir [56] and Aquifer [57]). These defense systems typically assume no implementation vulnerabilities in Android platform components. For example, SEAndroid [62] admits that it cannot mitigate kernel vulnerabilities or address threats from other platform components, while Weir [56] explicitly includes Android OS as its trusted computing base. However, as revealed by our analysis of patch code complexity in §4.3, a significant portion of Android vulnerabilities are likely implementation bugs. These implementation weaknesses could then turn down an originally secure system design.

Implication 4: *Our patch code patterns can be leveraged for automatic vulnerability detection using program analysis techniques.* A key problem in using static program analysis for vulnerability detection is to determine patterns, and our analysis in §4.4 can serve for this purpose. Specifically, extracted vulnerability patterns can be utilized in two ways. First, some patterns are context-independent (e.g., P1 and P2) or can be tracked using data/control flows (e.g., P3, P6, P10, P11, and P15), and thus can be directly inputted to a static analysis tool. For other patterns that are fully related to program contexts, learning-based methods (e.g., VulDeePecker [51]) can be further employed to distinguish different contexts.

5 RELATED WORK

In this section, we present the research related to Android system vulnerabilities, vulnerability report analysis, and similar or cloned code detection.

Research on Android system vulnerabilities. While most prior work was concerned about app-level vulnerabilities (e.g., [22, 24, 25, 27, 30, 34, 42, 48, 54, 58, 66–69, 72, 79, 80]), there are some recent studies specialized for Android system vulnerability detection. Notably, ADDICTED [77] made a first attempt in analyzing the (in)security of Android device drivers and they found that a large number of device drivers customized by vendors are underprotected with downgraded permissions. Following this direction, several studies of mobile device drivers were further performed, on a new dynamic analysis [63], on the ION driver insecurity [74], and on an Android-specific kernel driver called Binder [18, 28]. Compared to drivers, Android framework received more security research. For example, Kratos [60], Kywe et al. [46], Gu et al. [35], AceDroid [16], and ACMiner [38] discovered inconsistent security policy enforcement in the Android framework, while ASVHunter [37] and KMHunter [64] examined denial-of-service attack issues. Different from these studies on detecting unknown vulnerability instances, we aim to obtain insights from reported vulnerabilities.

Analysis of vulnerability reports. Our paper belongs to the general research category of analyzing vulnerability reports. The most related are two works [41, 52] that also analyzed Android vulnerability reports. Compared with our large-scale study via an automatic analysis framework, these two studies relied on significant manual efforts and used only a small set of vulnerabilities for analysis. Moreover, they did not present in-depth analysis, e.g., clustering patch code patterns as we did in this paper.

In the research of other vulnerability reports, Chen et al. [20] made a pioneer work on using a finite-state machine (FSM) to model and analyze memory corruption vulnerabilities in 2006. In 2011, Chen et al. [19] performed a high-impact study on analyzing 141 Linux kernel vulnerability reports. In 2017, Li and Paxson [47] conducted a generic measurement study of all kinds of security patches. There were also some studies surveying vulnerability reports as part of their research. For example, UniSan [53] surveyed the root causes of kernel information leaks reported after 2013, and InstaGuard [21] measured the patch delays of 12 vulnerabilities in Android system programs and also evaluated their patch solution in 30 selected Android vulnerabilities. Furthermore, a recent work, SemFuzz [73], leveraged vulnerability-related text from CVE reports and Linux git logs to guide automatic generation of proof-of-concept exploits.

Detection of similar or cloned codes. Code clone detection is a long-lasting problem in the software engineering and security areas. Back to 1998, Baxter et. al. [17] had proposed to use abstract syntax tree (AST) for clone detection. To improve the scalability, CCFinder [43], CP-Miner [49], and ReDeBug [39] splitted code into token sequences for a multilinguistic clone detection in large-scale source code, while Deckard [40] computed characteristic vectors for approximating ASTs and thus can cluster similar vectors only. VulPecker [50] and VUDDY [44] further abstracted vulnerability-related features specifically for vulnerable code clone detection. More recently, deep learning is also exploited for clone detection in source code [51] and binary code [71]. However, these clone detection works are not designed for finding similar code "changes", thus not suitable for our patch code clustering problem. Only two recent works, Kreutzer et al. [45] and Paletov et al. [59], also worked on clustering code changes. Our clustering algorithm differs these two by extracting patch code's essential changes and leveraging affinity propagation for automatic clustering without assuming any pattern template or structure.

6 CONCLUSION

In this paper, we conducted the first systematic study of Android system vulnerabilities by comprehensively analyzing all 2,179 vulnerabilities and their 1,349 publicly available patches on the Android Security Bulletin program over around three years. To support such analysis, we proposed an automatic analysis framework and its three analyzers for the analysis of vulnerable modules, patch code complexity, and vulnerability patterns. In particular, we designed a similarity-based algorithm that extracts patch code's essential changes and leverages affinity propagation to automatically cluster patch code patterns. With this analysis framework, we pinpointed the distribution of vulnerabilities in different Android modules, studied the complexity of Android patch code, and successfully obtained 16 vulnerability patterns that include six new ones not known in the literature. In the future, we plan to further improve our clustering algorithm by supporting long code fragments, and also evolve our analysis results over time.

ACKNOWLEDGEMENTS

We thank all the reviewers of this paper for their valuable comments. We especially thank Prof. Lingxiao Jiang for his helpful discussion on clustering diff code. This work is partially supported by the Singapore National Research Foundation under NCR Award Number NRF2014NCR-NCR001-012.

REFERENCES

[1] 2010. Nested JSON objects. https://stackoverflow.com/a/2098294/197165. (2010).
[2] 2016. Android Bug: 24296662. https://android.googlesource.com/platform/external/freetype/+/f720f0db. (2016).
[3] 2017. Android Bug: 68320413. https://android.googlesource.com/platform/external/libhevc/+/b686bb2d. (2017).
[4] 2017. Android Bug: 68341964. https://android.googlesource.com/platform/frameworks/base/+/4afa0352. (2017).
[5] 2017. printk: hash addresses printed with %p. https://lwn.net/Articles/737451/. (2017).
[6] 2018. diff Output Formats. https://tinyurl.com/diffOutput. (2018).
[7] 2018. Exploit Database for Android. https://www.exploit-db.com/?platform=android. (2018).
[8] 2018. Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. (2018).
[9] 2018. Jellyfish: a Python library for doing approximate and phonetic matching of strings. https://github.com/jamesturk/jellyfish. (2018).
[10] 2018. The JSON1 Extension in SQLite. https://www.sqlite.org/json1.html. (2018).
[11] 2018. platform/frameworks/av - Git at Google. https://android.googlesource.com/platform/frameworks/av/. (2018).
[12] 2018. platform/hardware/qcom/media - Git at Google. https://android.googlesource.com/platform/hardware/qcom/media/. (2018).
[13] 2018. platform/system/bt - Git at Google. https://android.googlesource.com/platform/system/bt/. (2018).
[14] 2018. Selenium: Web Browser Automation. https://www.seleniumhq.org/. (2018).
[15] 2018. sklearn.cluster.AffinityPropagation. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AffinityPropagation.html. (2018).
[16] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *Proc. ISOC NDSS*.
[17] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *Proc. IEEE ICSM*.
[18] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. 2015. Towards Analyzing the Input Validation Vulnerabilities associated with Android System Services. In *Proc. ACM ACSAC*.
[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. ACM APSys*.
[20] Shuo Chen, Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2006. Security Vulnerabilities: From Analysis to Detection and Masking Techniques. *Proc. IEEE* vol. 94, no. 2 (2006).
[21] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. 2018. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *Proc. ISOC NDSS*.
[22] Erika Chin, Adrienne Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-Application Communication in Android. In *Proc. ACM MobiSys*.
[23] Daniel Dieterle. 2014. Android Webview Exploit Tutorial (70% of Devices Vulnerable!). In *https://cyberarms.wordpress.com/2014/02/26/android-webview-exploit-tutorial-70-of-devices-vulnerable/*.
[24] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proc. ACM CCS*.
[25] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *Proc. USENIX Security*.
[26] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proc. ACM SOSP*.
[27] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proc. ACM CCS*.
[28] Huan Feng and Kang G. Shin. 2016. Understanding and Defending the Binder Attack Surface In Android. In *Proc. ACM ACSAC*.
[29] Matthew Finifter, Devdatta Akhawe, and David Wagner. 2013. An Empirical Study of Vulnerability Rewards Programs. In *Proc. USENIX Security*.
[30] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Proc. IEEE Symposium on Security and Privacy*.
[31] Brendan J. Frey and Delbert Dueck. 2007. Clustering by Passing Messages Between Data Points. *Science* 315, 5814 (2007).
[32] Ioannis Gasparis, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. 2017. Detecting Android Root Exploits by Learning from Root Providers. In *Proc. USENIX Security*.
[33] David Gens, Simon Schmitt, Simon Schmitt, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *Proc. ISOC NDSS*.
[34] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. 2014. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks. In *Proc. ISOC NDSS*.
[35] Yacong Gu, Yao Cheng, Lingyun Ying, Yemian Lu, Qi Li, and Purui Su. 2016. Exploiting Android System Services Through Bypassing Service Helpers. In *Proc. Springer SecureComm*.
[36] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad Sadeghi. 2014. ASM: A Programmable Interface for Extending Android Security. In *Proc. Usenix Security*.
[37] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In *Proc. ACM CCS*.
[38] Sigmund Albert Gorski III, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. 2019. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proc. ACM CODASPY*.
[39] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proc. IEEE Symposium on Security and Privacy*.
[40] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proc. ACM ICSE*.
[41] Matthieu Jimenez, Mike Papadakis, Tegawende F. Bissyande, and Jacques Klein. 2016. Profiling Android Vulnerabilities. In *Proc. IEEE International Conference on Software Quality, Reliability and Security*.
[42] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Peri. 2014. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proc. ACM CCS*.
[43] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering* 28, 7 (2002).
[44] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proc. IEEE Symposium on Security and Privacy*.
[45] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M. Eskofier, and Michael Philippsen1. 2016. Automatic Clustering of Code Changes. In *Proc. ACM MSR*.
[46] Su Mon Kywe, Yingjiu Li, Kunal Petal, and Michael Grace. 2016. Attacking Android Smartphone Systems without Permissions. In *Proc. 14th Annual Conference on Privacy, Security and Trust (PST)*.
[47] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proc. ACM CCS*.
[48] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. 2017. Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews. In *Proc. ACM CCS*.
[49] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proc. USENIX OSDI*.
[50] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proc. ACM ACSAC*.
[51] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proc. ISOC NDSS*.
[52] Mario Linares-Vasquez, Gabriele Bavota, and Camilo Escobar-Velasquez. 2017. An Empirical Study on Android-related Vulnerabilities. In *Proc. ACM MSR*.
[53] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proc. ACM CCS*.
[54] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proc. ACM CCS*.
[55] Huasong Meng, Vrizlynn L.L. Thing, Yao Cheng, and Zhongmin Dai. 2018. A survey of Android exploits in the wild. *Computers & Security* vol. 76 (2018).
[56] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android. In *Proc. USENIX Security*.
[57] Adwait Nadkarni and William Enck. 2013. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proc. ACM CCS*.
[58] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes. 2018. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators. In *Proc. IEEE Symposium on Security*

and Privacy.

[59] Rumen Atanasov Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Inferring Crypto API Rules from Code Changes. In *ACM PLDI*.

[60] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proc. ISOC NDSS*.

[61] Seven Shen. 2015. Setting the Record Straight on Moplus SDK and the Wormhole Vulnerability. In *https://blog.trendmicro.com/trendlabs-security-intelligence/setting-the-record-straight-on-moplus-sdk-and-the-wormhole-vulnerability/*.

[62] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. ISOC NDSS*.

[63] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *Proc. Usenix Security*.

[64] Kai Wang, Yuqing Zhang, and Peng Liu. 2016. Call Me Back!: Attacks on System Server and System Apps in Android through Synchronous Callback. In *Proc. ACM CCS*.

[65] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability In X86 Binary Using Symbolic Execution. In *Proc. ISOC NDSS*.

[66] Daoyuan Wu and Rocky K. C. Chang. 2014. Analyzing Android Browser Apps for file:// Vulnerabilities. In *Proc. Springer Information Security Conference (ISC)*.

[67] Daoyuan Wu and Rocky K. C. Chang. 2015. Indirect File Leaks in Mobile Applications. In *Proc. IEEE Mobile Security Technologies (MoST)*.

[68] Daoyuan Wu, Yao Cheng, Debin Gao, Yingjiu Li, and Robert H. Deng. 2018. SCLib: A Practical and Lightweight Defense against Component Hijacking in Android Applications. In *Proc. ACM CODASPY*.

[69] Daoyuan Wu, Debin Gao, Rocky K. C. Chang, En He, Eric K. T. Cheng, and Robert H. Deng. 2019. Understanding Open Ports in Android Applications: Discovery, Diagnosis, and Security Assessment. In *Proc. ISOC NDSS*.

[70] Daoyuan Wu, Xiapu Luo, and Rocky K. C. Chang. 2014. A Sink-driven Approach to Detecting Exposed Component Vulnerabilities in Android Apps. *CoRR* abs/1405.6282 (2014). http://arxiv.org/abs/1405.6282

[71] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proc. ACM CCS*.

[72] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. 2018. Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications. In *Proc. IEEE Symposium on Security and Privacy*.

[73] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proc. ACM CCS*.

[74] Hang Zhang, Dongdong She, and Zhiyun Qian. 2016. Android ION Hazard: the Curse of Customizable Memory Management System. In *Proc. ACM CCS*.

[75] Veo Zhang, Jason Gu, and Seven Shen. 2018. New AndroRAT Exploits Dated Privilege Escalation Vulnerability, Allows Permanent Rooting. In *https://blog.trendmicro.com/trendlabs-security-intelligence/new-androrat-exploits/*.

[76] Mingyi Zhao, Jens Grossklags, and Peng Liu. 2015. An Empirical Study of Web Vulnerability Discovery Ecosystems. In *Proc. ACM CCS*.

[77] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proc. IEEE Symposium on Security and Privacy*.

[78] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proc. IEEE Symposium on Security and Privacy*.

[79] Yajin Zhou and Xuxian Jiang. 2013. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proc. ISOC NDSS*.

[80] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *Proc. IEEE Symposium on Security and Privacy*.