An Empirical Assessment of Security Risks of Global Android Banking Apps

Sen Chen[1], Lingling Fan[1], Guozhu Meng[2,3], Ting Su[4], Minhui Xue[5], Yinxing Xue[6]
Yang Liu[1,8], Lihua Xu[7]
[1]Nanyang Technological University, Singapore
[2]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[3]School of Cyber Security, University of Chinese Academy of Sciences, China [4]ETH Zurich, Switzerland
[5]The University of Adelaide, Australia [6]University of Science and Technology of China, China
[7]New York University Shanghai, China [8]Zhejiang Sci-Tech University, China
chensen@ntu.edu.sg

ABSTRACT

Mobile banking apps, belonging to the most security-critical app category, render massive and dynamic transactions susceptible to security risks. Given huge potential financial loss caused by vulnerabilities, existing research lacks a comprehensive empirical study on the security risks of global banking apps to provide useful insights and improve the security of banking apps.

Since data-related weaknesses in banking apps are critical and may directly cause serious financial loss, this paper first revisits the state-of-the-art available tools and finds that they have limited capability in identifying data-related security weaknesses of banking apps. To complement the capability of existing tools in data-related weakness detection, we propose a three-phase automated security risk assessment system, named Ausera, which leverages static program analysis techniques and sensitive keyword identification. By leveraging Ausera, we collect 2,157 weaknesses in 693 real-world banking apps across 83 countries, which we use as a basis to conduct a comprehensive empirical study from different aspects, such as global distribution and weakness evolution during version updates. We find that apps owned by subsidiary banks are always less secure than or equivalent to those owned by parent banks. In addition, we also track the patching of weaknesses and receive much positive feedback from banking entities so as to improve the security of banking apps in practice. We further find that weaknesses derived from outdated versions of banking apps or third-party libraries are highly prone to being exploited by attackers. To date, we highlight that 21 banks have confirmed the weaknesses we reported (including 126 weaknesses in total). We also exchange insights with 7 banks, such as HSBC in UK and OCBC in Singapore, via in-person or online meetings to help them improve their apps. We hope that the insights developed in this paper will inform the communities about the gaps among multiple stakeholders, including banks, academic researchers, and third-party security companies.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380417

CCS CONCEPTS

• **Security and privacy → Software and application security**;

KEYWORDS

Mobile Banking Apps, Vulnerability, Weakness, Empirical Study

ACM Reference Format:
Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, Lihua Xu. 2020. An Empirical Assessment of Security Risks of Global Android Banking Apps. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380417

1 INTRODUCTION

Banking apps belong to the most security-critical and data-sensitive app category. Cashless mobile payment has significantly fragmented the traditional financial services, beginning with the first ATM and culminating in e-banking. Users often misconceive that banking apps provide secure transactions and an easy-to-use interface, by assuming all communications are done between local banking apps and remote bank servers securely (e.g., over HTTPS). Unfortunately, this assumption does not always hold. After examining many real-world banking apps, we find new types of weaknesses that are hard to be detected by existing industrial and open-source tools, e.g., QiHoo360 [17], AndroBugs [2], MobSF [13], and QARK [16]. For example, in a popular banking app from Google Play, the user will be asked to register with her personal information, including first name, last name, password, and address. After the user clicks the "register" button, the app sends an SMS attached with the sensitive data (in plain text) to authenticate that user, but the data is stored in the SMS outbox unexpectedly. If an attacker registers a content observer to the SMS outbox on the mobile device with `READ_SMS` permission, the user's sensitive data can be easily intercepted by the attacker. Indeed, many other real-world banking-specific weaknesses and attacks have been witnessed globally [59, 60]. Nowadays, banking apps pose new challenges, such as flaws and vulnerabilities [9, 11] that cause huge financial loss [3].

To understand the weaknesses exhibited in banking apps and help to improve the security of these apps, several studies have been done manually on a small-scale banking apps [30, 61, 63, 64]. The conclusions drawn from manual analysis may be more likely to be biased and cannot represent the security status of the entire

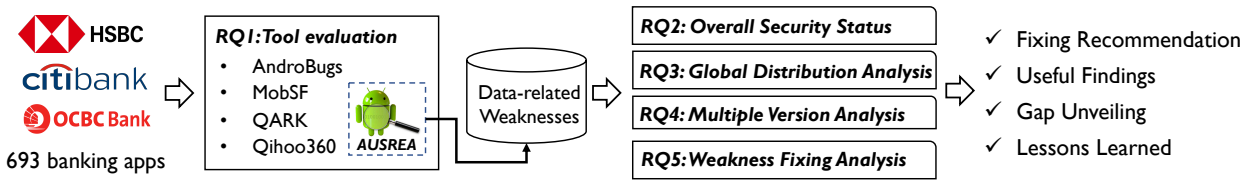

Figure 1: Overview of our study

banking ecosystem. Apart from manual analysis on only a small-scale apps, state-of-the-art assessment approaches also pose several other limitations: (1) current studies lack a baseline of sensitive data-related security weaknesses specific to the core functionality of banking apps to ensure an overall assessment of these apps; (2) the current off-the-shelf services (e.g., Qihoo360 [17]) and open-source tools (e.g., AndroBugs [2]) use syntax-based scanning to perform a security check during app development, which would incur a large number of false positives (e.g., non-sensitive data printed in the log file). Besides, these tools focus on generic categories of apps, not specific to banking apps. Even when the weaknesses, such as cryptographic misuses [42] and inappropriate SSL/TLS implementations [40, 45, 51, 65], have been reported for years, it still appears unknown why so many security weaknesses in banking apps are not yet patched [63]. Overall, the existing work cannot represent the security status of the entire banking ecosystem, and the state-of-the-art tools are ineffective in collecting a large number of weaknesses to conduct further in-depth analysis.

To explore the entire mobile banking ecosystem and help to ensure the user's financial security, this paper takes a large number of banking apps as subjects to conduct a comprehensive empirical study on the data-related weaknesses in global Android banking apps. As shown in Figure 1, our study contains three main steps: (1) we first collect 693 banking apps across 83 countries from various markets, to our knowledge, this is the largest banking app dataset taken into study to date; (2) to collect the weaknesses exhibited in banking apps and complement the capability of existing tools in data-related weakness detection, we first summarize a weakness baseline and propose an **au**tomated **se**curity **r**isk **a**ssessment system (Ausera). Ausera combines static program analysis techniques and sensitive keyword identification, to identify such weaknesses (cf. Section 2). (3) By applying Ausera, we collected 2,157 security weaknesses in the 693 banking apps, and further conduct a comprehensive empirical study (cf. Section 3) to investigate the ecosystem of banking apps in terms of security weaknesses, aiming to answer the following research questions:

- **RQ1:** What is the current status of existing tools towards collecting reliable data-related weaknesses in banking apps compared with Ausera?
- **RQ2:** What is the overall security status of banking apps in terms of data-related weaknesses?
- **RQ3:** What is the weakness status of banking apps globally w.r.t. economies and regulations?
- **RQ4:** How are weaknesses introduced during app evolution and fragmentation?
- **RQ5:** What is the gap between academic researchers and banks in understanding and fixing weaknesses?

Through an in-depth analysis of the weaknesses, we find that (1) banking apps across different regions exhibit various types of security status, mainly due to different economy status (e.g., small village banks) and financial regulations (e.g., GDPR [18]). Banking apps in Europe and North America have few security weaknesses, with only 0.27 weakness of data leakage per app. Asia is most flooded with security weaknesses, averaging out to 6.4 weaknesses per app. Banking apps from Africa have comparatively moderate security status with 4.6 weaknesses per app, primarily because of its high demand for cashless payment services. (2) Weaknesses of apps vary across different markets by countries and bring fragmentation problems among different versions of the same banking apps. Apps owned by subsidiary banks are always less secure than or equivalent to those owned by parent banks. This observation is evidenced by the South Korean version of the Citibank app and the Chinese version of the HSBC app. (3) Apart from the lessons learned from our study, we also track the weakness fixing process based on our reported weaknesses and set up 9 in-person or online meetings with 7 banks. These meetings help the communities understand the gaps between different parties, including banks, academic researchers, and third-party security companies.

In summary, we make the following contributions:

- To collect weaknesses in banking apps and complement the capability of existing tools in data-related weakness detection, we developed an automated security risk assessment system (Ausera), to efficiently identify security weaknesses in banking apps, outperforming 4 state-of-the-art industrial and open-source tools.
- To our knowledge, we conducted the first large-scale empirical study on 2,157 security weaknesses collected from 693 banking apps, the largest dataset taken into study to date. We attempt to investigate the ecosystem of global banking apps in terms of data-related weaknesses from four different aspects, such as global distribution analysis and evolution of multiple versions.
- We report the identified weaknesses to banks and provide simple-but-concrete fixing recommendations. To date, 21 banks have acknowledged our results, and 52 reported weaknesses have been patched by the corresponding banks. Some of them have actively collaborated with us to improve the security of their apps.

2 TOOL EVALUATION

In this section, we propose an automated weakness detection tool (named Ausera), guided by our constructed security weakness baseline in order to collect security weaknesses in banking apps. We also evaluate its effectiveness compared with the state-of-the-practice

Table 1: Taxonomy of security weaknesses in our study

Category	Security Weakness Type	
Input Harvest	Sensitive data harvested by screenshots	
Sensitive Data Storage	Stored in shared preferences	
	Stored in webview.db	
	Logged locally	
	Stored on SD Card	
	Written in text files	
Sensitive Data Transmission	Transmitted via SMS	
	ICC leaked	by dynamically registered Receiver
		by implicit Intent
		by component export
Communication Infrastructure	Only uses HTTP protocol	
	Uses invalid certificates (i.e., expiration, SHA-1 used)	
	Uses invalid certificate authentication	allows all hostname request
		uses invalid hostname verification
		uses invalid server verification
	Uses hard-coded encryption key	
	Uses improper AES encryption	uses insecure encryption
		uses improper function
	Uses improper RSA encryption	no RSA
		uses improper function
	Uses insecure SecureRandom (i.e., setSeed)	
	Uses insecure hash function (i.e., MD5 and SHA-1)	

tools to observe the current status of detection ability towards data-related weaknesses in banking apps. We then introduce the data collection process of banking apps and security weaknesses in these apps as the basis to conduct a large-scale analysis. Before proposing Ausera, we first revisit the state-of-the-art available tools or online services for weakness detection.

AndroBugs [2], QARK [16], and MobSF [13] are all open-source tools for detecting vulnerabilities in general Android apps. Specifically, AndroBugs is a framework to find potential vulnerabilities in Android apps by pattern-matching, and it also records some meta data in the database such as permissions used in the current app. QARK is designed to look for vulnerabilities related to Android apps, either in source code or packaged APKs. MobSF is a pen-testing framework, which is able to detect app vulnerabilities, and the results can be displayed on webpages. Apart from the open-source detection tools, Qihoo360 is a popular security company in China, which maintains an app scan engine, named appscan [17]. It is a free online application for security risk scanning service.

However, the current off-the-shelf services and tools have the following limitations in banking specific weakness collection according to our investigation: (1) They usually use syntax-based scanning, thus cannot verify the actual data flow, which would incur a large number of false positives that are not related to sensitive data leakage. (2) They usually aim to detect weaknesses in general apps, not specific to banking apps. Thus the patterns they use to detect weakness are difficult to detect data-related weaknesses in banking apps. The detection ability of the state-of-the-art weakness detection tools are demonstrated in Section 2.2. Considering the aforementioned situations, to complement the capability of existing tools in data-related weakness detection, we propose a tool, Ausera, for automating the detection and collection of sensitive-date related weaknesses specific to banking apps.

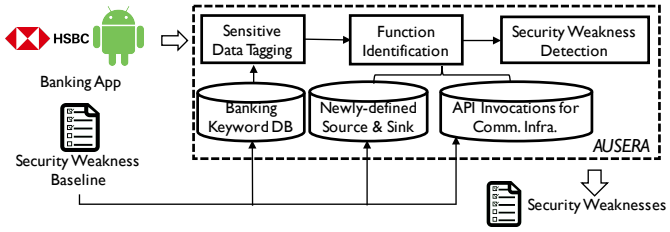

Figure 2: Overview of Ausera

2.1 Ausera

In order to collect a data-related weakness dataset specific to banking apps, we first propose a taxonomy of sensitive data-related security weaknesses in banking apps. Guided by the baseline, Ausera is proposed to identify weaknesses in banking apps.

2.1.1 ***Taxonomy of Security Weaknesses within Banking Apps.*** We propose and integrate security weaknesses of mobile banking apps from prior research [30, 61, 63, 64], best industrial practice guidelines and reports (e.g., OWASP [14], Google Android Documentation [10], and AppKnox security reports [59, 60]), NowSecure reports [70], and security weakness and vulnerability databases (e.g., CWE [22], CVE [21]). We take an in-depth look at the weaknesses w.r.t. ***sensitive data***, since the biggest threat to banking apps comes from manipulation of digital assets and routine financial activities. As shown in Table 1, sensitive data may be exposed to attackers through various ways as follows:

- *Input Harvest,* confidential inputs and user relevant sensitive data (e.g., transaction details) can be harvested via UI screenshot by malicious apps on rooted devices, or even adb-enabled devices without root access [55].
- *Data Storage,* an adversary is able to obtain data stored in local storage (e.g., shared preference, webview.db) on rooted devices or external storage (e.g., SD Card), and also from the output of the Android logging system without root access.
- *Data Transmission,* sensitive data transmission via SMS can be easily intercepted by malware observing the outbox of Android SMS service. Moreover, data leakage via inter-component communication (ICC) is another potential threat, allowing third parties to obtain data from banking apps by making implicit intent calls, or dynamic registration of a broadcast Receiver.
- *Communication Infrastructure,* MITM attack can obtain sensitive data through sniffing network traffic between client and server, thereby sending fake data to either party. This kind of attack is generally achieved due to improper authentication protocols, insecure cryptography, lack of certificate verification, etc.

Our baseline contains data-related weaknesses of multiple categories and builds a solid foundation for analyzing weaknessesin banking apps.

2.1.2 ***Methodology of Ausera.*** To collect a large dataset of security weaknesses, Ausera takes as input each banking app, guided by the weakness baseline, and ultimately outputs security weaknesses of the app. Figure 2 shows the overview of Ausera, including three phases: (1) *Sensitive data tagging*, which identifies sensitive data in banking apps, including user inputs and the data

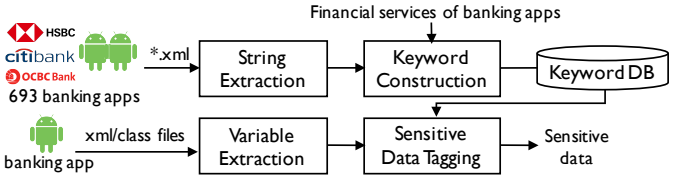

Figure 3: Identification of sensitive data

from server displayed on the UI pages, and then attaches semantics to the sensitive data-related variables in .xml/.java files according to our constructed sensitive keyword database. (2) *Function identification*, which identifies the functions related to data leakage such as preference storage, SMS transmission, and determines the behavior of a piece of code based on API invocations (or their call sequence patterns). (3) *Weakness detection*, which performs taint analysis based on the tagged sensitive data and functions to check the existence of weaknesses in the proposed baseline.

• **Sensitive data tagging.** Since we are concerned about the sensitive data in banking apps that may incur security risks, we manually extract typical data-related keywords in banking apps.

Figure 3 shows the process of sensitive keyword database construction and sensitive data tagging. (1) *Sensitive keyword DB construction.* To construct the keyword database, we first extract all strings (i.e., component ID name of EditText, the hint text of EditText, and the text of TextView) from the layout files of 693 banking apps by reverse engineering. We then filter the strings according to the core financial services of banking apps such as login, payment, etc. Note that, to avoid missing variants of the keywords, we further employ Word2vec [57] to supplement the corpus of the keyword database. Specifically, we load the trained model based on the *.bin* word vector, using the sentences extracted by Supor [52] from 54,371 general apps. For example, we further find the string "passwd" is a variant of the sensitive keyword of "password." These sensitive keywords are able to indicate the semantics of the components (i.e., EditText and TextView). Eventually, we build a sensitive keyword database containing 70 keywords, which can be classified into 4 categories as shown in Table 2. Currently, we only consider two languages (i.e., English and Chinese). In the future work, we may extend the language types. The full list of keywords is publicly available online.[1] (2) *Sensitive data tagging.* Based on the keyword database, we can identify the sensitive data-related variables in the code and attach semantics to them. Specifically, we first extract variables related to two kinds of components: EditText for user input and TextView for data display. For each component, as shown in Figure 4, there may be several variables declaring different aspects of the component such as the component ID, component hint, and component text. Therefore, we extract all the variables related to each component, and then tag the variable as sensitive if it matches with any keyword in the keyword database. Note that, the semantic tagging method in previous work [28, 52] relies on the component relation in layouts, which may lose some user inputs. As a result, sensitive data is tagged with its semantics in the format ⟨*variable*, *keyword*⟩.

[1] www.sites.google.com/view/ausera/

Table 2: Keyword examples

Category	Keyword Examples	Number
Identity	username, userid, byname, user-agent	13
Credential	password, passcode, pwd, pin	11
Personal Info	name, phone, email, birthday	24
Financial Info	credit card, amount, payment, payee	22

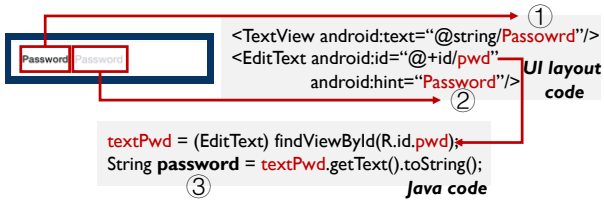

Figure 4: Code relation between Java and UI layout code

For the example described in the introduction, the sensitive data is tagged as ⟨edit_PIN, pin⟩, ⟨edit_firstName, firstname⟩, ⟨edit_lastName, lastname⟩, ⟨edit_addr, addr⟩, so the app in the example is confirmed to send sensitive data out via SMS.

• **Function identification.** The sensitive data extracted above are defined as *sources*, and be far apart from the access of unauthorized users. We use our newly-defined sinks (i.e., specific-APIs) to identify function code that is associated with weaknesses for banking services. However, as discussed in Section 2.1.1, these sensitive data may be divulged during the storage or transmission process. To achieve confidentiality, the sensitive data should not flow into a code point where unauthorized users can access via local storage, external storage, logging output, SMS, and component transition in Table 1 (a.k.a., *sinks* of sensitive data). It is worth mentioning that the sinks here are different from the sinks defined in SuSi [62]. SuSi's sinks are all potential method calls with 12 categories that leak sources out of mobile devices, while our newly-defined sinks are leaking sensitive data through specific leakage channels (e.g., shared preferences, logging output, and SMS). According to the leakage channels, we manually define 106 vulnerable sinks [20] in total that are likely to be exploited in banking apps.

Communication infrastructure, which is indispensable to banking apps [42, 64]. It establishes a channel to communicate with remote bank servers. However, communication infrastructure is likely to be attacked, and hence it can undermine the security of these apps. The core functionalities in communication infrastructure include certificate verification, cryptographic operation, and host authentication. To accurately identify the functional code for communication infrastructure, we summarize all invocation patterns of multiple Android APIs for each functionality. Taking hostname verification as an example, if there is an invocation sequence {`new X509HostnameVerifier`, `setHostnameVerifier` of class `HttpURLConnection`}, we consider that the app uses hostname verification during communication. We further check its implementation to determine whether it implements correctly. We have 12 groups of API invocation patterns in total for function identification in communication infrastructure. We reverse-engineer banking apps, locate the invocations of these relevant Android APIs, and

use call graphs and component transitions to determine their call relation in between. Finally, we can identify the functional code for communication infrastructure of banking apps.

• **Security weakness detection.** Given a banking app, we attempt to find whether it contains any weaknesses listed in Table 1 and reduce false positives by employing the two strategies: *(1) a forward data-flow analysis to determine whether there exists sensitive data flowing into insecure sinks by leveraging sensitive data tagging and taint analysis; (2) a backward control-flow analysis to check whether the vulnerable functional code identified by API invocation patterns in communication infrastructure is feasible based on call graphs and Activity transitions.*

We carry on a forward taint analysis on top of Soot [71] to support intra- and inter-component communication analysis based on the tagged sensitive data. These data are regarded as ***sources***, and the sinks are the Android API list we defined. During the process of functional code identification, we can obtain all vulnerable code (i.e., ***incorrect implementation***) that exists in communication infrastructure. However, noise may arise because the dead code for testing purpose cannot be executed during runtime. Reaves et al. [63, 64] found that the dead code may bring false positives to the detection results. We perform a backward control-flow analysis, and extract all reachable call sequences according to call graphs and Activity transitions. If the vulnerable code is reachable, we determine it is a valid weakness, otherwise, it is a false alarm.

We highlight the following three strategies to reduce false positives. (1) Ausera reduces the size of our extracted keywords from 124 to 70, which effectively reduces ambiguity of the keywords (e.g., "info" and "status"), and hence can identify sensitive data more accurately. (2) Ausera utilizes newly-defined sources and sinks, which are relevant to weaknesses of sensitive data leakage. (3) Ausera identifies the vulnerable code and checks its reachability to eliminate weaknesses in dead code by call graphs and Activity transitions.

2.1.3 ***Implementation of Ausera.*** To implement Ausera, we combine static program analysis and sensitive data tagging to identify sensitive data in banking apps, and associate them with the corresponding variables in XML/Java code. Ausera relies on Apktool [5] to extract resource files from apks. It then uses parts-of-speech (POS) tagger of OpenNLP-1.8.3 [19] to parse the text labels in TextView and EditText, thereby identifying keywords included. We manually check on these keywords to retain the ones that are sensitive and relevant to the core functionalities of banking apps. After that, we employ Word2vec to supplement the keyword database.

To accomplish the detection, we summarize 12 groups of patterns (e.g., AES/ECB/NoPanding) to depict the communication weaknesses. Then we employ pattern-based static analysis to find the possible vulnerable patterns in code. We check three aspects for certificate authentication: whether the client side 1) allows all hostname requests; 2) bypasses hostname verification; 3) fails to implement anything in the server verification method (`checkServerTrusted`). The weakness "hard-coded encryption key" is determined by first checking whether an encryption key is embedded in code, and examining whether it is used to encrypt sensitive data to reduce false positives. The banking sensitive data are encrypted with the DES or Blowfish algorithm. Using either of the encryption mechanisms is viewed as a weakness [63, 64]. The AES forbids ECB mode because it does not provide a general notion of privacy [42]. The padding of AES and RSA is always improper, such as `NoPadding` and `PKCS1`, though AES/ECB/NoPadding is very frequently used. The function SecureRandom should not be seeded with a constant. The hash functions MD5 and SHA-1 are insecure [72, 73].

Table 3: Distribution of the collected banking apps

Continent	#Developed	#Developing	Total	Percentage
Europe	102	0	102	21.7%
America	53	24	77	16.4%
Asia	16	210	226	48.1%
Oceania	16	0	16	3.4%
Africa	0	49	49	10.4%
Total	187	283	470	-

2.1.4 ***Evaluation of Ausera.*** We randomly selected 60 banking apps (12.8%) in our dataset and manually checked the detection results to evaluate Ausera's precision. Note that, we cannot evaluate the false negatives when assessing banking apps due to lack of weakness benchmarks of banking apps. False positive (FP) refers to weakness that are detected during static analysis but actually infeasible at runtime or detected by mistake. As a result, we only found 6 false positives (corresponding to five weakness types, i.e., Shared Preference Leakage, Logging Leakage, SD Card Leakage, Text File Leakage, and Hard-coded Key) from the identified 341 weaknesses of these 60 banking apps, achieving an average precision of 98.24%.

Consequently, 5 out of 6 false positives belong to sensitive data leakage. The reason is that Ausera matches variables (e.g., "pkgname.txt," "login_fragement," "loginpager," and "spinnerGender") inaccurately with the keywords in our database. The remaining one FP belongs to Hard-coded Key type, because the extracted variable is relevant to the exception parameters (i.e., "KeyPermanentlyInvalidateException").

2.2 RQ1: Tool Evaluation and Data Collection

Banking app collection. As shown in Figure 1, we collected 693 banking apps[2] in total from various Android markets such as Google Play store and APKMonk [24]. Note that we only collect multiple versions of some apps from APKMonk to conduct the longitudinal anlaysis (cf. Section 3.3) since APKMonk maintains the full versions of apps, while Google Play store only maintain the latest version. The collected apps range across 470 unique banks, where some apps have multiple versions. They originate from both developed and developing countries across five continents (see breakdowns in Table 3). Table 3 indicates that 48.1% of the banking apps are from Asia, considering the largest population proportion all over the world. Only 3.4% of apps are from Oceania, considering its smallest population proportion all over the world. The 24 banking apps of American developing countries all originate from South America, while 16 apps of Oceanian developed countries originate from Australia and New Zealand. 16 apps of Asian developed countries

[2] Apart from the 693 apps, we have filtered out apps with packer techniques (49 apps in total) and with decompilation failure since they are out of scope in this paper.

Table 4: Detection result comparisons

Tools	#Types	Precision	Time/App (mins)
Ausera	341	98.24%	1.6
Qihoo360	80	87.50%	8.5
AndroBugs	76	81.58%	1.8
QARK	93	87.10%	16.1
MobSF	213	48.36%	2.4

originate from Singapore, Japan, and South Korea. To our knowledge, this is the largest banking app dataset taken into study to date.

Comparison with the state of the practice. We compare the detection results of Ausera with 4 industrial and open-source tools, including Qihoo360, AndroBugs, MobSF, and QARK. We randomly select 60 banking apps in our dataset for comparison and run each tool 3 times to stabilize the detection accuracy. Table 4 shows the results. Ausera outperforms other tools in both precision and time cost, achieving 98.24% precision in 1.6 minutes per app. The precisions for these tools are obtained by manual validation through filtering out all false positives. We also conduct a cross-validation of detection results across different authors. We can see that all comparisons of the detection results comply with the weaknesses baseline. Ausera outperforms other tools with higher precision and less time. Ausera manages to scan each app within 1.6 minutes on average, much faster than the other tools.

In particular, we show several specific cases to explain how false positives are incurred. Sensitive data disclosure through logging can be detected by MobSF, however, MobSF just matches the following APIs if used (e.g., `Log.e()`, `Log.d()`, and `Log.v()`), without further determining whether the output data is sensitive or not. There is no doubt that it would incur a large number of false positives. If the data is not sensitive, such as "`menu_title`," it is very normal for developers to output it in the terminal or write messages to understand the state of their application. The risk is that some credentials (e.g., PIN and password) are also leaked by logging outputs. A syntax-based scanning tool may provide an incomplete and incorrect analysis result due to the influence of dead code. For example, Qihoo360 detected three code blocks violating server verification, e.g., do nothing in `checkServerTrusted`. In contrast, Ausera aims to to minimize the influence of dead code. Two key strategies to eliminate such false positives are: (i) checking whether invalid authentication is in a feasible path in call graphs; (ii) checking whether the `Class` has been instantiated in Activity transitions.

Apart from the comparison with the above 4 tools, we also discuss the comparison between Ausera and two taint analysis tools (i.e., FlowDroid [29] and IccTA [54]). Ausera aims to identify weaknesses specifically in banking apps, while FlowDroid and IccTA, which largely rely on sources and sinks defined in SuSi, aim to identify the data leakage in general apps. (1) The sources and sinks considered by FlowDroid and IccTA are specified by SuSi, which contains 12 different source categories and 15 different sink categories. However, among them, we only use taint analysis on 4 types of weaknesses (i.e., Shared preference leakage, logging leakage, SD card leakage, and SMS leakage). In other words, FlowDroid and IccTA cannot detect most of security weakness types in our proposed data-related baseline specific to banking apps. (2) In fact, we have deployed FlowDroid and IccTA on our defined sources and sinks, and find that they cannot identify the concrete data types (i.e., sensitive or non-sensitive) when tracking the 4 types of weaknesses. For example, developers usually output debug information such as string length via logging channel, however, tracking such non-sensitive data causes many false positives. While Ausera only tracks the labeled sensitive data that are most relevant to the core financial services of banking apps. (3) Most of the sources defined in SuSi are not sensitive in banking apps, such as the API invocations of Bluetooth, Calendar, and Settings. More comparison results can be found on our website [20].

Table 5: Weaknesses in 470 banking apps

Weakness Category	Weakness Type	#Affected Apps
Input Harvest	Screenshot	415 (88.3%)
Data Storage	Shared preference	44
	WebView DB	64
	Logging	66
	SD Card	14
	Text File	10
Data Transmission	SMS Leakage	18
	ICC Leakage	324 (68.9%)
Communication Infrastructure	HTTP Protocol	84
	Invalid Certificate	31
	Invalid Authentication	222
	Hard-coded Key	30
	Improper AES	131
	Improper RSA	231
	Insecure SecureRandom	133
	Insecure Hash Function	340 (72.3%)

Answer to RQ1. In summary, existing state-of-the-practice tools are less effective (i.e., lower precision, more false positives, and cost more time) in identifying data-related weaknesses in banking apps, compared with Ausera. Therefore, Ausera can be used to collect a large number of security weaknesses for further in-depth analysis.

Weakness collection. Ausera is demonstrated as the most effective tool to collect banking specific security weaknesses, we thus apply it on the collected 693 banking apps across 83 countries. Finally, we collect 2,157 security weaknesses for further large-scale empirical study.

3 A LARGE-SCALE COMPREHENSIVE EMPIRICAL STUDY

In this section, we conduct a large-scale empirical study from different aspects based on the collected weaknesses by Ausera.

3.1 RQ2: Security Status of Banking Apps

Since the multiple versions of a banking app may have overlapped weaknesses, we select the latest version of the 693 apps if they have multiple versions, and apply Ausera to these 470 unique banking apps to conduct the following study. Table 5 shows the results of weaknesses corresponding to the security baseline defined in Section 2.1.1.

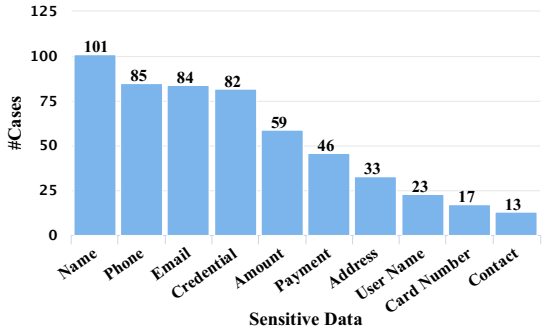

Figure 5: Top 10 sensitive data types leaked in 470 banking apps

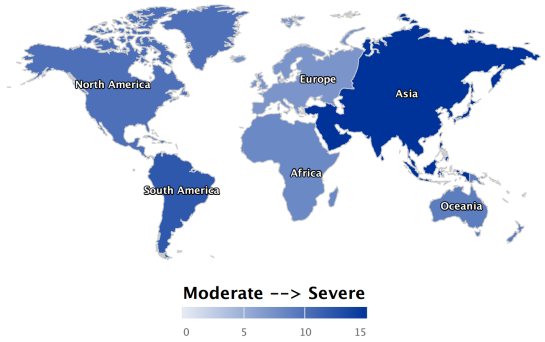

Figure 6: Number of weaknesses in global banking apps

Input harvest. Screenshot (88.3%), as an easy-to-use way to harvest users' credentials, is most likely to be neglected by developers. Only 55 apps (e.g., Bank of Communications of China) are protected from screenshots in our investigation.

Data storage. Only a small portion of apps store sensitive data on SD Card (2.98%) and Text File (2.13%), which are globally accessible and thereby susceptible to privacy leakage. We show that Shared preference, Logging, and WebView DB are the main channels that leak sensitive data. As shown in Figure 5, Ausera identifies 592 cases of private data leakage across 470 unique banking apps. *Credentials* (e.g., PIN), as the most dangerous leakage in banking apps, appear in 82 cases and affect 64 banking apps. Note that *banking-specific data* (e.g., transaction password and card number) accounts for 22.47%, and the other data leakage includes *personal info* (e.g., Name, Phone, and Email).

Data transmission. We show that ICC Leakage (68.9%) is also among the most popular weaknesses. Despite the small portion of SMS Leakage, SMS could directly forward credentials, thwarting confidentiality. For example, the real banking app mentioned in the introduction leaks sensitive data such as pin, first name, last name, and address via SMS.

Communication infrastructure. The protection of communication infrastructure in banking apps is far away from satisfactory. More specifically, many apps are still using HTTP to exchange sensitive data with the remote bank server, or do not validate the certificates of the connected servers. We find 222 banking apps with invalid authentication, including 13 banking apps that have both invalid and correct SSL/TLS implementations in source code. They establish communications with servers using different strategies (i.e., invalid and correct SSL/TLS implementation). Insecure Hash Function (72.3%) is also frequently misused.

Answer to RQ2. Overall, the security status of banking apps is severe according to the results. In summary, Screenshot (88.3%), Insecure Hash Function (72.3%), and ICC Leakage (68.9%) are the most popular weaknesses of banking apps. Meanwhile, Invalid Authentication (222 apps) also has severe damage.

3.2 RQ3: Global Distribution of Weaknesses

Figure 6 shows the number of weaknesses discovered among the banking apps by continents. The intensity scale encodes the number of weaknesses the apps have, scaled from light blue (least) to dark blue (most). We observe the following findings: (1) Weaknesses in banking apps of Asia outnumber those of Europe (resp. North America) by 1.56 (resp. 1.31) to 1, where each banking app of Asia has 6.4 weaknesses on average, indicating that the banking apps of developed countries (i.e., Europe and North America) have fewer weaknesses than those of developing countries. Ironically, to our surprise, we find that weaknesses in apps of Asian developed countries slightly outnumber (with 6.7 weaknesses per app) those of Asian developing countries. (2) Banking apps from Africa exhibit satisfactory security status, having only 4.6 weaknesses on average, some are even more secure than those of developed countries. Possible reasons why the security of banking apps varies across regions can be interpreted as follows:

- The financial regulations and development guidelines are different across regions, which may affect the implementation of banking apps. For example, both Europe (GDPR [18]) and USA (PCI DSS [15]) adopt very strict security and privacy regulations. The GDPR poses a regulatory framework that is unique to the financial service industry. Failure to meet its requirements will come with potentially hefty penalties [44]. This is also reflected by the 143 banking apps from Europe and USA, where data leakage rarely exists, with only 0.27 data leakage weakness reported per app.
- The development budget and developers' expertise may affect the security of products. During our investigation, we find that a number of local banking apps of China have many more weaknesses than international or nationwide ones. We speculate that due to inadequate budget for app development, those released apps are prone to being less secure.
- Cashless payment systems have been bootstrapped in areas where traditional banking is uneconomical and expensive, removing large investments on the massively deployed financial infrastructure. This is evidenced by the fact that Kenya, a country in Africa, is a world leader of money transfers by mobile [1], and 68% of people in Kenya report the use of phones for a financial service [12].

Answer to RQ3. We conclude that apps across different countries exhibit various types of security status, mainly because of different economies and regulations that take shape. We find that apps from Africa have comparatively moderate security status, primarily because of its high demand for cashless services.

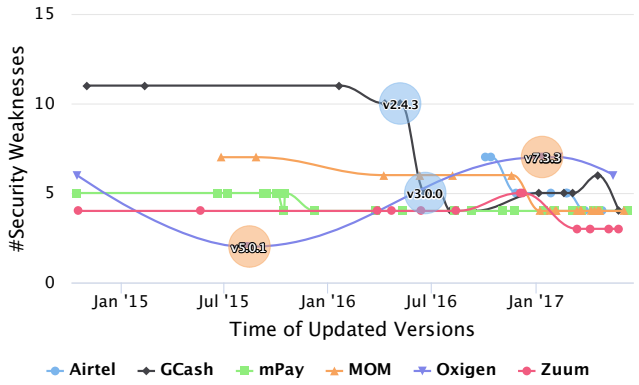

Figure 7: Number of weaknesses in each update version.

3.3 RQ4: Longitudinal Analysis of Version Updates and Fragmentation

We attempt to perform a longitudinal study on security risks by revisiting the 7 apps (GCash, mPay, MOM, Zuum, Oxigen Wallet, Airtel Money, and mCoin) which have been systematically studied by Reaves et al. [64], with confirmed weaknesses. We downloaded all available versions of 6 apps (mCoin is excluded since history versions are not publicly available.), and obtained 88 different versions in total, i.e., GCash (6 versions), mPay (20 versions), MOM (22 versions), Zuum (12 versions), Oxigen Wallet (12 versions), and Airtel Money (8 versions). All versions span more than two years.

Figure 7 shows the number of detected weaknesses across all versions of each app. We can see most of the version updates (90%) fail to bring at least two successful patches for weaknesses in their history versions, which echoes the findings of paper [63] that apps have not repaired critical vulnerabilities in their new versions. After an in-depth manual analysis, we find input harvest via screenshots, MITM attacks, AES/RSA misuses, and insecure hash functions are the most common weaknesses that remain unfixed. Furthermore, developers usually neglect hostname verification or server authentication, which may enable the MITM attack. These apps are also not aware of AES/RSA misuses and insecure hash functions, indicating that developers are still not aware of these weaknesses perpetually.

GCash has a sharp decline from v2.4.3 to v3.0.0 in terms of the number of weaknesses. Three weaknesses are patched, the hard-coded encryption key, insecure SecureRandom, and privacy leakage to SD Card. Reaves et al. [63, 64] found that the vulnerabilities still remain in the updated version in 2016. However, according to our security reports, GCash fixed most of the vulnerabilities in their latest version. In contrast, the weaknesses of Oxigen Wallet significantly increase from v5.01 to v7.3.3 due to the changes of app features. More specifically, many new weaknesses (i.e., `WebView DB` Leakage, ICC Leakage, MITM Attacks, and Insecure SecureRandom) were introduced, which had not been discovered by Reaves et al. [64]. They compared the code similarity between the 2015 and 2016 versions of each app, and found some apps have significant amounts of new code [63]. This aligns with our study that many banking apps do not perform systematic security checks before delivery.

Furthermore, we find banks encounter the version fragmentation problem especially when they release versions to different markets by countries. We selected the top 5 banking apps based on the S&P Global Market Intelligence report [7] across their 30 different versions, i.e., Citibank (10 versions), HSBC (3 versions), Deutsche Bank (3 versions), Banco Santander (8 versions), and ICBC (6 versions). By comparing the differences of weaknesses between these versions, we observe the following: (1) A subsidiary bank, incorporated in the host country but owned by a foreign parent bank, usually launches its original financial services with most of its products, such as banking apps, into the host market. As a result, a subsidiary bank inherits the weaknesses from the original version of its parent bank. This observation is evidenced by the South Korean version of Citibank app and the Macau version of ICBC app (see Figure 8). (2) Due to the business difference, culture difference, and expertise of security teams, weaknesses of apps vary across different markets by countries. This is also evidenced by the fact that the official app of HSBC (China) v2.7.1 has more weaknesses than that of HSBC (UK) and HSBC (Hong Kong). A possible reason might be that HSBC (China) is independent of the parent bank in terms of its app development outsourcing procedures and security teams, while in Hong Kong, as the former UK colony, HSBC (Hong Kong) largely follows the convention of HSBC (UK). Nevertheless, we find that not all subsidiary banks operate under the host country's regulations in terms of the number of banking app security risks (Figure 8 shows the source and host countries of flows containing security weaknesses.).

Answer to RQ4. By revisiting apps studied by previous research and further examining them across all their publicly available versions that have not been scrutinized before, we conclude that app developers are still not aware of these weaknesses perpetually. Furthermore, apps owned by subsidiary banks are always less secure than or equivalent to those owned by parent banks, for which the assumption that subsidiary banks operate under the host country's regulations does not always hold true.

3.4 RQ5: Weakness Fixing and Feedback

Our study has uncovered 2,157 weaknesses in total from 693 banking apps, most of which have been reported to the corresponding banks. As shown in Table 7, 21 banks have replied and confirmed these weaknesses, and **16 apps** have been patched.[3] Furthermore, we approached the major stakeholders across the global, such as HSBC (UK/Hong Kong/Shanghai), OCBC (Singapore), DBS (Singapore), and BHIM (India), to understand their security practice and policies. Through in-depth discussions with 7 banks, we find they hold different mindsets toward assessing severity of weaknesses and setting security goals. Note that, on average, the 7 banks take 41 days to fix their security weaknesses we reported. We elaborate this gap and provide our insights on how to close it.

Lack of effective criteria for rating security weaknesses. An effective severity criterion of weaknesses is crucial for banks to prioritize security patching. However, such a criterion is still missing for banking apps. As a result, some banks use CVSS [23] to determine the severity of the identified weaknesses. However, this

[3]We do not disclose any concrete weakness types or details in these banking apps to avoid security threats.

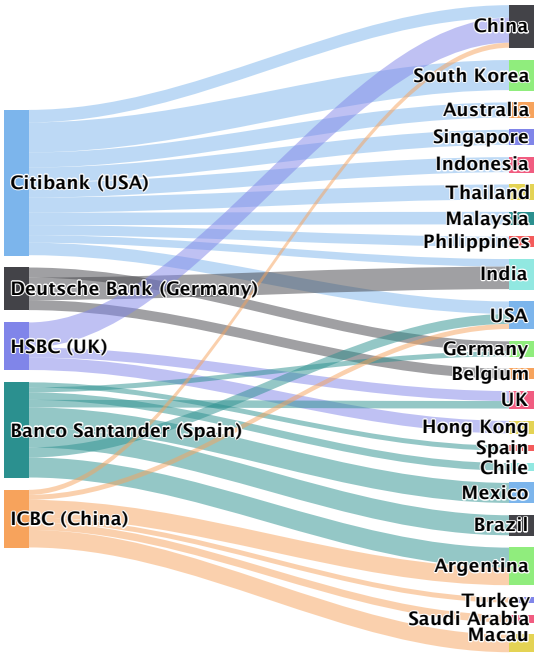

Figure 8: Flow of security weaknesses from parent banks to subsidiary banks across the world. The flow width indicates the number of weaknesses originating and terminating between two corresponding banks. Different types of banks are encoded by different colors.

standard is not perfect in practice [25–27, 58], and provides few principled ways to characterize security risks and potential impact. Moreover, we find these banks hold subjective attitudes toward fixing different types of weaknesses. For example, most banks are concerned about obvious privacy leakage (e.g., leakage from `SharedPreference`, Logging, SMS, SD Card, Text File, and WebView DB), while they are only aware of and somehow reluctant to fix the weaknesses, such as ICC Leakage, Invalid Certificate, and Insecure Hash Function. Table 6 summarizes our observations on various banks' attitudes towards different weaknesses, which are classified by "Concerned" (high priority) and "Aware" (low priority).

Lack of systematic security checks and validation tools. Many banking apps do not undergo a systematic security check and validation before delivery — Ausera discovers a large number of high-severity weaknesses, e.g., sensitive data leakage, hard-coded key and invalid authentication. With the assistance of Ausera, many banks, e.g., OCBC and Zijin Bank, expeditiously patched the weaknesses in their new versions. However, ironically, some banks patched the weaknesses but introduced new ones at the same time. For example, C∗ patched two weaknesses (i.e., Logging Leakage and HTTP Protocol) by employing SSL over HTTPS communication. However, new weaknesses are introduced in the updated version, i.e., the app fails to verify the identity of the bank server (`checkServerTrusted`), which echoes the finding of [63, 64] that 4 apps have new vulnerabilities. Due to lack of systematic security checks and validation tools, many security weaknesses still reside in these apps.

Table 6: Different concerns from banks

	Security Weaknesses
Concerned	Screenshot, `SharedPreference` Leakage, Logging Leakage, SMS Leakage, SD Card Leakage, Text File Leakage, WebView DB Leakage, Invalid Authentication, Hard-coded Key, Insecure SecureRandom
Aware	ICC Leakage, HTTP Protocol, Invalid Certificate, Improper AES/RSA, Insecure Hash Function

Outdated versions remain in effect in the wild. Banks usually hold the assumption that customers always keep their apps updated, and thus concentrate more on the weaknesses of latest versions than those of outdated versions. However, this assumption is *not true*, considering the device fragmentation problem — Android apps have to be compatible with more than 10 major versions of Android OS running on over 24,000 distinct device models; and it is also *dangerous*, considering attackers can leverage the weaknesses of outdated versions to mount specific attacks. We find that most banking apps across multiple versions still remain in effect in the wild (e.g., Apkmonk [24]). On average, these apps have 7.7 different versions, and the most fragmented app has 25 versions. Thus, we strongly recommend banks push compulsory app updates to the customers or block access to outdated apps, especially when high-severity weaknesses were patched.

Risks from third-party libraries. Our study finds the third-party libraries, e.g., `com.google.android.gms.*` and `com.facebook.*`, are widely used in banking apps. Ausera detects BHIM (v2.3.6) and MyAadhar (v1.9.3) use insecure third-party hash functions, such as MD5 and SHA-1, to produce message digests, which have already been accepted as insecure [66, 73]. Banks still use these insecure functions despite being aware of the insecure, as they assume that ordinary attackers are not capable of breaking them. However, it is still possible for experienced attackers to mount a large-scale attacks by exploiting these weaknesses. Banks are liable if they use security-weakened or poisoned third-party libraries without careful inspection. To avoid an "amplification effect" caused by the weaknesses in third-party libraries [41], we strongly recommend banks to carefully inspect third-party libraries in use.

Answer to RQ5. Incomplete security criterion provides banks wide leeway to use one-sided judgment about specific security practices. We also observe that outdated versions and weaknesses from third-party libraries are all likely to be exploited. They remain unfixed for weeks to months post-disclosure. This gap provides opportunities for attackers to strike. Understanding the gap between industrial practitioners and academic researchers help illuminate the nature of patching process.

3.5 Case Studies of Weaknesses

To showcase the exploitability of these weaknesses, we introduce 4 vulnerable apps reported by Ausera.

Screenshot weakness. A∗ Bank (v3.3.1.0038) employs two-factor authentication, i.e., the user first inputs the username and password, and then enters verification code sent by the bank server. It can be attacked if the login page is not protected (without setting the flag `WindowManager.LayoutParams.FLAG_SECURE` to prohibit taking

```
1 // Save Username and Password to Preference
2 private void SaveCredentials() {
3   if (this.rememberMe.isChecked()) {
4     Editor editor = UnamePrefs.edit();
5     Editor editor1 = PasswordPrefs.edit();
6     // Save Username
7     editor.putString("Uname", etUsername
      .getText());
8     // Save Password
9     editor1.putString("Password",etPassword
      .getText());
10    editor.commit();
11    editor1.commit();
12    return;
13 }
```

Figure 9: Simplified code of Preference weakness in G∗

```
1 // Update new banking app version
2 public static void update(Context context) {
3   if(checkNewVersion()){
4     getApk("AndroidBankingApp.apk", SDCard);
5 }
6 // Check banking app version
7 Private boolean checkNewVersion(){
8   Connection con = new Connection();
9   con.checkServerTrusted(); return true;
10 }
11 // Check bank server
12 public final void checkServerTrusted(X509Cert[]
13 x509CertificateArr, String str) {
14   // do nothing
15 }
```

Figure 10: Simplified code of update weakness in I∗ SMS

```
1 private String IV = "fedcba9876543210";
2 private String KEY = "tQna25tR89d6af1a";
3 // Encrypt Data
4 public static String encryptStr(String text){
5   Cipher cipher = Cipher.getIns("AES/CBC/NoPadding");
6   cipher.init(1, KEY, IV);
7   plainText = bytesToHex(cipher.doFinal());
8   return text;
9 }
10 // Decrypt Data
11 public static String decryptStr(String encStr){
12   Cipher cipher = Cipher.getIns("AES/CBC/NoPadding");
13   cipher.init(2, KEY, IV);
14   encStr = cipher.doFinal(hexToBytes(encStr));
15   return encryptedStr;
16 }
```

Figure 11: Simplified code of en/decryption weakness in N∗

Table 7: Weaknesses tracking of 21 banking apps. 16 banks have already patched their banking apps, and the rest have confirmed the weaknesses in their replies and will fix them soon in new versions.

No.	Banking Apps	#W	#Patched	#New	Country	Downloads
1	HSBC*	5	2	0	UK	5M - 10M
2	PSD Bank	3	2	0	Germany	50K - 100K
3	BBBank	3	2	0	Germany	50K - 100K
4	Intesa Sanpaolo Mobile	5	2	0	Italy	1M - 5M
5	AIB Mobile	8	1	0	Ireland	5M - 10M
6	Alma Bank	6	3	0	Russia	5K - 10K
7	Discover Mobile	8	4	0	USA	10M - 50M
8	Citizens Bank of Lafayette	2	1	2	USA	5K - 10K
9	CDB	6	2	2	China	5K - 10K
10	Zijin Bank	8	7	0	China	10K - 50K
11	DBS	10	0	0	Singapore	5M - 10M
12	OCBC	9	8	0	Singapore	5M - 10M
13	MyAadhar	4	0	0	India	50M - 100M
14	BHIM	3	2	0	India	10M - 50M
15	ICICI Netbanking	7	0	0	India	100M - 500M
16	ICICI Pockets	7	0	0	India	50M - 100M
17	GCash	11	8	0	Philippines	10M - 50M
18	Bank Australia	7	2	0	Australia	10K - 50K
19	CaixaBank	5	2	0	Brazil	1M - 5M
20	BMCE Bank	5	2	0	Morocco	100K - 500K
21	NMB Mobile Bank	4	0	1	Zimbabwe	10K - 50K

"#W": The number of detected weaknesses. "#Patched": the number of patched weaknesses in update versions. "#New": The number of newly-introduced weaknesses in update versions. "Country" means the country of bank headquarters.
"∗": The HSBC Cybersecurity team has reviewed and responded that the remaining three reported "weaknesses" in HSBC China version (v2.7.1) are not vulnerabilities, but are features purposely retained to support market specific customer requirements.

screenshot), and the verification code can be accessed with granted permissions. As such, we generate a malicious app [33, 68] that runs a service which can take screenshot of the screen and read the verification code from SMS during the process of login. As a result, the remote attacker can steal the credentials and bypass the login authentication. Note that the crafted malware [38, 49, 50] has bypassed the security vetting of Google Play and is successfully put on the shelf, which makes this attack more practical [36, 37, 39].

Preference weakness. Figure 9 shows the vulnerable code of a `Preference` weakness in G∗ Bank (v1.1) from Algeria. This app stores the credentials (i.e., username and password) into `Preference` named `UnamePrefs` and `PasswordPrefs` (lines 6-9). To steal these credentials, we can either (1) create a malicious app signed with the same key, so that it can run in the same sandbox as the victim app on a non-rooted device; or (2) create a malicious app that modifies the original file permission from "660" to "777" by running `Runtime.getRuntime().exec` on rooted devices [63, 64]. In either way, the malware can access the victim's sensitive data stored in the `Preference`. Even worse, we find that several apps use insecure permissions `MODE_WORLD_READABLE/WRITEABLE` rather than `MODE_PRIVATE`, which eases such attacks.

Version update weakness. I∗ SMS Bank (v5.0) is detected as having a MITM risk during version updates, the vulnerable code is shown in Figure 10. The app checks new versions with the bank server once started (line 3), but does not verify the `X.509` certificates from SSL servers (lines 11-15). It allows MITM attackers to spoof the server by crafting an arbitrary certificate. As a result, the new version can be downloaded to SD Card from an attack server (line 4). To exploit this, we use Burp Suite [6] and Fiddler [8] to fool the banking app, by sending a malicious app to impersonate the most recent version [4]. After this malicious app is installed, it serves as a phishing app to steal user credentials and other data.

Encryption/Decryption attack. Ausera detects an encryption weakness in N∗ Bank (v1.8) as shown in Figure 11. It leaves the hard-coded AES keys (`IV` and `KEY`) as plain text (lines 1-2), and uses them to encrypt and decrypt the communication between the app and the bank server. By leveraging these keys, we successfully decrypt all sensitive data during communication. Moreover, AES uses block cipher modes. If we set with `NoPadding` (lines 5 and 12), it is easier for attackers to subvert encryption because they only need to decrypt one of the blocks.

4 LESSONS LEARNED AND LIMITATIONS

Lessons learned. (1) According to the security assessment of global banking apps in Table 5, banking apps are not as secure as we expected in the real world. Meanwhile, the results of the global status and longitudinal studies unveil many security threats and unreasonable phenomena. Stockholders such as security teams in banks should pay more attention on these security issues. (2) The processes of weakness reporting and patches tracking reveal the gaps between academic researchers, banks, and third-party security companies. (3) The processes of meeting and discussions between corresponding banks bring useful recommendations, and some of

them have been used to improve the banking app security. (4) From the perspective of banks, they should pay more attention to security issues compared with functional bugs. Meanwhile, they should provide various channels to respond to the reported vulnerabilities, to make the patching process more efficient. (5) Fortunately, some of banks have accepted our reported vulnerabilities and actively collaborated with us to improve their app security by using Ausera before releasing new app versions.

Limitations. (1) The proposed data-related baseline is integrated by many channels based on our depth understanding and knowledge, thus might be incomplete. However, we can investigate the global ecosystem of banking apps based on the baseline. Meanwhile, according to the communications with real banks, they are highly concerned about the security weaknesses we proposed in Table 1. (2) The keyword database is constructed first with manual selection of keywords, and then extended with the help of NLP techniques. However, some of keywords may be ignored in the manual analysis process. Actually, the database can be further extended with the increasing banking apps. (3) Ausera is built on the top of the static analysis framework (i.e. Soot), thus inherits the limitation of Soot that it may fail and lose some data flows, creating false negatives.

5 RELATED WORK

Security assessment of banking apps. In 2015, Reaves et al. [64] realized the severe weaknesses of branchless banking apps. They reverse engineered and then manually analyzed 7 apps from developing countries, and last found 28 significant weaknesses. Most of these weaknesses remained unresolved after one year [63]. Chanajitt et al. [31] also manually analyzed 7 banking apps, and investigated three types of weaknesses, including how much sensitive data is stored on device, whether the original apps can be substituted, and whether communication with the remote server can be intercepted. Our study differs from [31, 63, 64] with regards to the scope of the study. Whereas [31, 63, 64] mainly leverage case studies to study banking apps, the focus of our paper is to conduct a large-scale empirical study on security weaknesses of banking apps. Furthermore, we also incorporate multidisciplinary expertise (e.g., code comprehension, regulations, economics) to interpret the potential causes of occurrence of security weaknesses. Our work also differs from alternative topics, such as functional bugs [47, 48, 67], performance [56] and fragmentation [74]. For the concrete security weaknesses, for example, SSL issues have been widely discussed in [46], which suggests revisiting the SSL handling in applied platforms (e.g., iOS and Android). Followed by recent reports [53, 61] and our observation, we find that many banking apps have fairly weak or even no authentication and encryption mechanisms. Sounthiraraj et al. [65] proposed to combine static and dynamic analysis to identify security problems in SSL/TLS for Android apps. Georgiev et al. [51] focused on SSL connection authentication of non-browser software, indicating that SSL certificate validation is defective and vulnerabilities are logical errors, due to the poor design of APIs to SSL libraries and misuse of such APIs. Egele et al. [42] checked for violations of 6 cryptographic rules (using cryptographic APIs) in real-world Android apps. They applied static analysis to extract necessary information to evaluate the properties and showed that about 88% of the apps violate the security rules. For our research, we also integrate these aforementioned weaknesses as vulnerable security points, and examine whether banking apps contain these vulnerabilities.

Global analysis of banking apps. Castle et al. [30] conducted a manual analysis of 197 Android apps and interviewed 7 app developers across developing countries (Africa and South America). They divided 13 hypothetical attacks into 5 categories and concluded that realistic concerns are on SMS interceptions, server attacks, MITM attacks, unauthorized access, etc. Lebeck et al. [53] summarized weaknesses of mobile money apps in developing economies, and combined existing techniques (e.g., cryptocurrencies) to achieve security and functionality goals. Parasa et al. [61] studied 9 mostly-used mobile money apps across 9 Australasian countries, and reported the security weaknesses in authentication, data integrity, poor protocol implementation, malfunction, and overlooked attack vectors. They reported that the apps from comparatively developed countries (e.g., AliPay, Osaifu-Keitai) also have weaknesses. Besides, Taylor et al. [69] adopted two off-the-shelf tools to roughly scan the apps that are labeled as finance from Google Play Store. All these prior work adopts small-scale analysis or is taken by survey, while our results are obtained in an automated and largest-scale fashion, which have not been systematically scrutinized before. Besides, Chen et al. [35] focused on studying the details of issue-reporting and issue-patching lifecycle based on the results of weakness detection tools like Ausera [34]. It unveils gaps between the industry and academia regarding the inconsistent understanding of reported issues and responsibilities. However, in this paper, we propose a comprehensive taxonomy of data-related security weaknesses for banking apps, and propose a detection approach based on the taxonomy. Using Ausera, we conducted experiments to identify security weaknesses and investigate the overall ecosystem of global banking apps from multiple aspects.

Security analysis of Android apps. Taint analysis is a commonly-used method to reveal potential privacy leakage in Android apps. For example, TaintDroid [43] is a dynamic taint-tracing tool which tracks flows of private data by modifying Dalvik virtual machine; FlowDroid and IccTA [29, 54] are both static taint analysis tools that accept the *source* and *sink* configurations for privacy leaks. However, these tools target on general apps [32], and thus may not be able to unveil specific security weaknesses (summarized in Table 1) when applied for banking apps. We also detail the differences in Section 2.2.

6 CONCLUSION

In this paper, we conduct a large-scale comprehensive empirical study on the collected 2,157 security weaknesses of 693 banking apps across more than 80 countries from various aspects. To collect the dataset, we also propose a three-phase system, Ausera, to automatically identify data-related weaknesses in banking apps. Our detected security weaknesses (i.e., 52 security weaknesses) have been confirmed and patched by the 21 corresponding banks and some of them have actively collaborated with us to improve the security of their banking apps. The study also narrows down the gaps between academic research and industrial banks, and helps both banks and third-party companies to better tackle security weaknesses.

ACKNOWLEDGMENTS

We appreciate all the reviewers for their valuable comments. We would like to acknowledge HSBC Cybersecurity team for their conscientious response to our responsible disclosure. This work is partially supported by the National Satellite of Excellence in Trustworthy Software System (Award No. NRF2018NCR-NSOE003-0001), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001) and the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE004-0001.

REFERENCES

[1] 2007. Kenya sets world first with money transfers by mobile. https://www.theguardian.com/money/2007/mar/20/kenya.mobilephones. (2007).
[2] 2015. AndroBugs. https://github.com/AndroBugs/. (2015).
[3] 2015. Over $7,000 lost in malware attack at fake banking portal. http://www.straitstimes.com/singapore/over-7000-lost-in-malware-attack-at-fake-banking-portal/. (2015).
[4] 2017. Android vulnerability allows attackers to modify apps without affecting their signatures. https://www.helpnetsecurity.com/2017/12/11/android-modify-apps-without-affecting-signatures/. (2017).
[5] 2017. Apktool: A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/. (2017).
[6] 2017. Burp Suite. https://portswigger.net/burp. (2017).
[7] 2017. Data Dispatch: The world's 100 largest banks. http://www.snl.com/web/client?auth=inherit#news/article?id=40223698&cdid=A-40223698-11568. (2017).
[8] 2017. Fiddler: Free Web Debugging Proxy - Telerik. http://www.telerik.com/fiddler. (2017).
[9] 2017. Flaw discovered in banking apps leaving millions vulnerable to hack. http://www.telegraph.co.uk/science/2017/12/06/flaw-discovered-banking-apps-leaving-millions-vulnerable-hack/. (2017).
[10] 2017. Google Best Practices for Security & Privacy. https://developer.android.com/training/best-security.html. (2 2017).
[11] 2017. Hackers' Delight: Mobile bank app security flaw could have smacked millions. https://www.theregister.co.uk/2017/12/11/mobile_banking_security_research/. (2017).
[12] 2017. Kenya tops Africa in use of mobile financial services. http://kenyanwallstreet.com/kenya-tops-africa-use-mobile-financial-services-report. (2017).
[13] 2017. Mobile-Security-Framework-MobSF. https://github.com/MobSF/Mobile-Security-Framework-MobSF. (2017).
[14] 2017. OWASP: OWASP Mobile Security Project. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10. (2 2017).
[15] 2017. PCI: Security Standards Council. https://www.pcisecuritystandards.org/. (2017).
[16] 2017. QARK: Tool to look for several security related Android application vulnerabilities. https://github.com/linkedin/qark. (2017).
[17] 2017. Qihoo360 (Appscan). http://appscan.360.cn/. (2017).
[18] 2017. The EU General Data Protection Regulation. https://www.eugdpr.org/. (2017).
[19] 2018. Apache OpenNLP 1.8.3. https://opennlp.apache.org/news/release-183.html/. (2018).
[20] 2018. AUSERA. https://sites.google.com/view/ausera/. (2018).
[21] 2018. CVE: Common Vulnerabilities and Exposures. https://cve.mitre.org/. (2018).
[22] 2018. CWE: Common Weakness Enumeration. https://cwe.mitre.org/. (2018).
[23] 2018. The Common Vulnerability Scoring System. https://www.first.org/cvss/. (2018).
[24] 2019. Apkmonk. (2019). https://www.apkmonk.com
[25] 2019. Scoring security vulnerabilities 101: Introducing CVSS for CVEs. (2019). https://snyk.io/blog/scoring-security-vulnerabilities-101-introducing-cvss-for-cve/
[26] 2019. Towards Improving CVSS. (2019). https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=538368
[27] Luca Allodi, Sebastian Banescu, Henning Femmer, and Kristian Beckers. 2018. Identifying relevant information cues for vulnerability assessment using CVSS. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 119–126.
[28] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. 2017. UiRef: analysis of sensitive user inputs in Android applications. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 23–34.
[29] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
[30] Sam Castle, Fahad Pervaiz, Galen Weld, Franziska Roesner, and Richard Anderson. 2016. Let's talk money: Evaluating the security challenges of mobile money in the developing world. In *Proceedings of the 7th Annual Symposium on Computing for Development*. ACM, 4.
[31] Rajchada Chanajitt, Wantanee Viriyasitavat, and Kim-Kwang Raymond Choo. 2018. Forensic analysis and security assessment of Android m-banking apps. *Australian Journal of Forensic Sciences* 50, 1 (2018), 3–19.
[32] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 596–607.
[33] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. 2019. GUI-Squatting Attack: Automated Generation of Android Phishing Apps. *IEEE Transactions on Dependable and Secure Computing* (2019).
[34] Sen Chen, Guozhu Meng, Ting Su, Lingling Fan, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2018. Ausera: Large-scale automated security risk assessment of global mobile banking apps. *arXiv preprint arXiv:1805.05236* (2018).
[35] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 2018. Are mobile banking apps secure? what can be improved?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 797–802.
[36] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *Computers & Security* 73 (2018), 326–344.
[37] Sen Chen, Minhui Xue, Lingling Fan, Lei Ma, Yang Liu, and Lihua Xu. 2019. How can we craft large-scale Android malware? An automated poisoning attack. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. IEEE, 21–24.
[38] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. 2016. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 377–388.
[39] Sen Chen, Minhui Xue, and Lihua Xu. 2016. Towards adversarial detection of mobile malware: poster. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*. ACM, 415–416.
[40] Tom Chothia, Flavio D Garcia, Chris Heppel, and Chris McMahon Stone. 2017. Why banker Bob (still) can't get TLS right: A Security Analysis of TLS in Leading UK Banking Apps. In *International Conference on Financial Cryptography and Data Security*. Springer, 579–597.
[41] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2187–2200.
[42] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 73–84.
[43] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems* (2014).
[44] European Parliament and Council of the European Union. 1995. Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal L* 281 (1995), 0031–0050.
[45] 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 *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 50–61.
[46] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL development in an appified world. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 49–60.
[47] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 486–497.
[48] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale Analysis of Framework-specific Exceptions in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 408–419.

[49] Lingling Fan, Minhui Xue, Sen Chen, Lihua Xu, and Haojin Zhu. 2016. Poster: Accuracy vs. time cost: Detecting Android malware through pareto ensemble pruning. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 1748–1750.

[50] Ruitao Feng, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, and Shang-Wei Lin. 2019. MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 61–70.

[51] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 38–49.

[52] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *USENIX Security Symposium*. 977–992.

[53] Kiron Lebeck, Temitope Oluwafemi, Tadayoshi Kohno, and Franziska Roesner. 2015. *Rethinking Mobile Money Security for Developing Regions*. Technical Report. University of Washington.

[54] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 280–291.

[55] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets.. In *NDSS*.

[56] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 1013–1024. http://doi.acm.org/10.1145/2568225.2568229

[57] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[58] Nuthan Munaiah and Andrew Meneely. 2016. Vulnerability severity scoring and bounties: Why the disconnect?. In *Proceedings of the 2nd International Workshop on Software Analytics*. ACM, 8–14.

[59] Prateek Panda. 2015. *Security Report of Top 100 Mobile Banking Apps-APAC*. Technical Report. Appknox.

[60] Prateek Panda. 2016. *A Security Analysis of The Top 500 Global E-commerce Mobile Apps in USA, UK, Australia, Singapore and India*. Technical Report. Appknox.

[61] Swathi Parasa and Lynn Margaret Batten. 2016. Mobile Money in the Australasian Region-A Technical Security Perspective. In *International Conference on Applications and Techniques in Information Security*. Springer, 154–162.

[62] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.. In *NDSS*.

[63] Bradley Reaves, Jasmine Bowers, Nolen Scaife, Adam Bates, Arnav Bhartiya, Patrick Traynor, and Kevin RB Butler. 2017. Mo (bile) money, Mo (bile) Problems: Analysis of branchless banking applications. *ACM Transactions on Privacy and Security (TOPS)* 20, 3 (2017), 11.

[64] Bradley Reaves, Nolen Scaife, Adam M Bates, Patrick Traynor, and Kevin RB Butler. 2015. Mo (bile) Money, Mo (bile) Problems: Analysis of Branchless Banking Applications in the Developing World. In *USENIX Security*. 17–32.

[65] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. Smv-hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 14)*. Citeseer.

[66] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The First Collision for Full SHA-1. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings*. 570–596.

[67] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[68] Chongbin Tang, Sen Chen, Lingling Fan, Lihua Xu, Yang Liu, Zhushou Tang, and Liang Dou. 2019. A large-scale empirical study on industrial fake apps. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 183–192.

[69] VF Taylor and I Martinovic. 2017. A Longitudinal Study of Financial Apps in the Google Play Store. In *Financial Cryptography and Data Security, Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg*.

[70] Chris Thompson, Ryan Leininger, and Roshani Bhatt. 2017. *Mobile Banking Applications: Security Challenges for Banks*. Technical Report. Accenture & NowSecure Inc.

[71] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 13.

[72] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. 2005. Finding collisions in the full SHA-1. In *Crypto*, Vol. 3621. Springer, 17–36.

[73] Xiaoyun Wang and Hongbo Yu. 2005. How to break MD5 and other hash functions. In *Eurocrypt*, Vol. 3494. Springer, 19–35.

[74] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 226–237.