We find that apps owned by subsidiary banks are always less secure. 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 We further find that weaknesses derived from outdated versions of banking apps or third-party libraries are highly prone to being exploited by attackers.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.

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

ACM Reference Format:
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 automated security risk assessment 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...
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 MOSSF [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. MOSSF 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, QHOO360 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.

### 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 can 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 outbound 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 weaknesses in 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...
Financial services of banking apps

693 banking apps

String Extraction

Keyword Construction

Variable Extraction

Sensitive Data Tagging

Sensitive 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 “password” 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 further extracted 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).

Table 2: Keyword examples

<table>
<thead>
<tr>
<th>Category</th>
<th>Keyword Examples</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>username, userid, byname, user-agent</td>
<td>13</td>
</tr>
<tr>
<td>Credential</td>
<td>password, passcode, pwd, pin</td>
<td>11</td>
</tr>
<tr>
<td>Personal Info</td>
<td>name, phone, email, birthday</td>
<td>24</td>
</tr>
<tr>
<td>Financial Info</td>
<td>credit card, amount, payment, payee</td>
<td>22</td>
</tr>
</tbody>
</table>

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

1www.sites.google.com/view/ausera/
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/NoPadding) 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 (checkServerTrusteed). 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>
<thead>
<tr>
<th>Continent</th>
<th>#Developed</th>
<th>#Developing</th>
<th>Total</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Europe</td>
<td>102</td>
<td>0</td>
<td>102</td>
<td>21.7%</td>
</tr>
<tr>
<td>America</td>
<td>53</td>
<td>24</td>
<td>77</td>
<td>16.4%</td>
</tr>
<tr>
<td>Asia</td>
<td>16</td>
<td>210</td>
<td>226</td>
<td>48.1%</td>
</tr>
<tr>
<td>Oceania</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>3.4%</td>
</tr>
<tr>
<td>Africa</td>
<td>0</td>
<td>49</td>
<td>49</td>
<td>10.4%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>187</strong></td>
<td><strong>283</strong></td>
<td><strong>470</strong></td>
<td><strong>-</strong></td>
</tr>
</tbody>
</table>

### 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 weaknesses 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, Test 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., “pkg-name.txt,” “login_framgement,” “loginipager,” 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 analysis (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.
which contains 12 different source categories and 15 different sink weaknesses specifically in banking apps, while

**FlowDroid**

weaknesses baseline. Ausera

A syntax-based scanning tool may provide an incomplete and incor-

(i.e., **FlowDroid** [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, compared with **Ausera**. Therefore, **Ausera** can be used to collect a large number of security weaknesses for further in-depth analysis.

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 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.

#### Table 5: Weaknesses in 470 banking apps

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

#### Table 4: Detection result comparisons

<table>
<thead>
<tr>
<th>Tools</th>
<th>#Types</th>
<th>Precision</th>
<th>Time/App (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ausera</td>
<td>341</td>
<td>98.24%</td>
<td>1.6</td>
</tr>
<tr>
<td>Qihoo360</td>
<td>80</td>
<td>87.50%</td>
<td>8.5</td>
</tr>
<tr>
<td>AndroBugs</td>
<td>76</td>
<td>81.58%</td>
<td>1.8</td>
</tr>
<tr>
<td>QARK</td>
<td>93</td>
<td>87.16%</td>
<td>16.1</td>
</tr>
<tr>
<td>MonSF</td>
<td>213</td>
<td>48.36%</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Comparison with the state of the practice. We compare the detection results of **Ausera** with 4 industrial and open-source tools, including Qihoo360, AndroBugs, MonSF, 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 MonSF; however, MonSF 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 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, **Ausera** has been instantiated in Activity transitions.

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.

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].

**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.
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, AUSELLA 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.
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 hardcoded 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.

3.3 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 hardcoded 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.

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. 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

---

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

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., Apkmirror [24]). On average, these apps have 7.7 different functions despite being aware of the insecure, as they assume that outdated vulnerabilities are unlikely to be attacked if the login page is not protected (without setting the flag WindowManager.LayoutParams.FLAG_SECURE to prohibit taking.

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.
screenshots), 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 (line 4). To exploit this, we use 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.

**Version update weakness.** In 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-NC005-0001) and the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE004-0001.

REFERENCES

An Empirical Assessment of Security Risks of Global Android Banking Apps

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


