AUSERA: Large-Scale Automated Security Risk Assessment of Global Mobile Banking Apps

Sen Chen*, Guozhu Meng‡, Ting Su†, Lingling Fan*, Minhui Xue§, Yinxing Xue¶, Yang Liu†, Lihua Xu¶
*East China Normal University, †Nanyang Technological University
‡Chinese Academy of Sciences, §Optus Macquarie University Cyber Security Hub and Data61-CSIRO
¶University of Science and Technology of China, ||New York University Shanghai

Abstract—Contemporary financial technology (FinTech) that enables cashless mobile payment has been widely adopted by financial institutions, such as banks, by virtue of its convenience and efficiency. However, FinTech makes massive and dynamic transactions susceptible to security risks. Given large financial losses caused by such vulnerabilities, regulatory technology (RegTech) has been developed, but more comprehensive security risk assessment is specifically desired to develop robust, scalable, and efficient financial activities. In this paper, we undertake the first automated security risk assessment and focus on global banking apps to examine FinTech. First, we analyze a large number of banking apps and propose a comprehensive set of security weaknesses widely present in these apps. Second, we design a three-phase automated security risk assessment system (AUSERA), which combines natural language processing and static program analysis of data and control flows, to efficiently identify security weaknesses of banking apps. We performed experiments on 693 real-world banking apps across over 80 countries and unveiled 2,157 weaknesses. To date, 21 banks have acknowledged the weaknesses that we reported. We find that outdated version of banking apps, pollution from third-party libraries, and weak hash functions are highly prone to being exploited by attackers. We also show that banking apps of different provenance exhibit various types of security weaknesses, mainly due to economies and regulations that take shape. Given the drastic change in the nature of intermediation, it behooves the RegTech companies and all stakeholders to understand the characteristics and consequences of security risks brought by contemporary FinTech.

Index Terms—Mobile Banking Apps, Security Risk Assessment, Security Weaknesses

1 INTRODUCTION

CONTEMPORARY financial technology (FinTech) that encourages cashless mobile payment has significantly fragmented the traditional financial services, beginning with the first ATM and culminating in e-banking. These massive dynamic transactions have paved a real path to global financial inclusion, stimulating economic growth and employment among both developed and developing countries. However, FinTech also presents new challenges for financial regulators, such as flaws and vulnerabilities that cause huge financial loss [1, 2, 3], thereby necessitating a parallel development of recent regulatory technology (RegTech) [4]. In particular, regulators should develop a robust, scalable, and quick solution that facilitates financial innovation and market confidence, aided by the use of comprehensive security risk assessment.

We, hereby, take large-scale banking apps as a peephole to examine contemporary FinTech which is deeply intertwined with the traditional financial infrastructure. Users often misconceive that banking apps provide secure transactions and provide 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 can hardly be discovered by existing industrial and open-source tools [5, 6, 7, 8]. For example, in a popular banking app from Google Play, a user will be asked to register with her personally identifiable 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 plaintext) 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 who impersonates that user to manipulate her legitimate banking account. Indeed, many other real-world banking-specific weaknesses and attacks have been witnessed globally [1, 9, 10, 11]. Therefore, developing security risk assessment is highly desirable for FinTech. Understanding at scale the banking entities that own these mobile banking services (e.g., how they operate, their relationships with subsidiaries, and what the regulations take shape) is critical to the sustainable growth of FinTech.

In this paper, we undertake the first automated security risk assessment of large-scale global banking apps, whereby regulators can locate the security risks of banking apps and inform security teams of patching the weaknesses expeditiously. However, existing state-of-the-art assessment approaches impose several limitations: (1) current studies lack a comprehensive baseline of security weaknesses to ensure an overall assessment of banking apps. If conclusions are drawn from small-scale manual analysis [12, 13, 14, 15], they are more likely to be biased and cannot represent the security status of the entire mobile banking ecosystem; (2) the current off-the-shelf services (e.g., QIHOO360 [5]) and open-source tools (e.g., ANDROBUGS [6]) usually focus on generic categories of apps, not directly applicable to financial apps. In addition, these industrial tools only use syntax-based scanning to perform a security check during app development, which will incur large numbers of false positives (e.g., the influence of dead code); (3) for recent cryptographic misuses [16] and inappropriate SSL/TLS implementations [17, 18, 19, 20] reported for years, it still appears unknown why so many security weaknesses of banking apps are not yet patched [15].

To address these limitations and challenges, our **au**tomated **se**curity **r**isk **a**ssessment system (AUSERA) combines natural language processing (NLP) techniques and static analysis (e.g., data flow analysis). The system has three major components: (1) *sensitive data tagging*, where we use word embeddings to identify sensitive data contained in banking apps, and assign the semantics to declared variables in code; (2) *function identification*, which concentrates on the very code related to key modules in banking apps and determines the behavior of a piece of code; (3) *weakness detection*, which tags sensitive data and functions, and performs static analysis on banking apps to check if any weakness is included. By leveraging these techniques, we cast our research in a comprehensive and automated fashion, and manage to explore the security impact and reduce the gap between academia and industry.

We conduct our experiments on a real-world dataset, comprising collectively 693 banking apps across over 80 countries. To our knowledge, this is the largest banking app dataset taken into study to date. We finally identify 2,157 weaknesses in total, including botched certification validation, weak authentication protocols, and privacy violations. We further study these apps to explore the demographic and longitudinal results and security impact which could interpret the drastic change in the nature of intermediation. Banking apps across different regions exhibit various types of security status, mainly due to different economies (e.g., small village banks) and financial regulations (e.g., GDPR [21]) that take shape. Weaknesses of apps vary across different markets by countries and bring version fragmentation problem. 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.

In summary, we make the following contributions:

- We develop and integrate a set of baseline security policies into the overall security assessment that corresponds to popular attack strategies.
- We develop the first automated security risk assessment system (AUSERA), to quickly identify security weaknesses of banking apps, with precision of over 98%, outperforming 4 state-of-the-art industrial and open-source tools [5, 6, 7, 8]. The detection approach also combines NLP techniques and static program analysis of data and control flows.
- We take a large-scale measurement analysis on 693 banking apps, the largest dataset taken into study to date, which was collected across over 80 countries. These in-depth studies include demographic and longitudinal analysis.

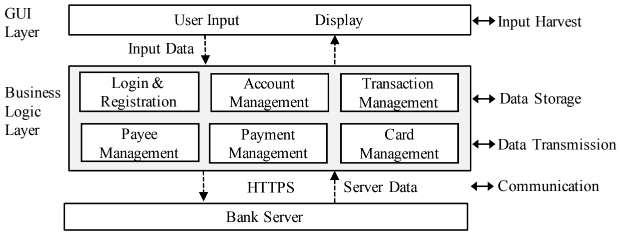


Fig. 1. Architecture of banking apps. The dotted lines show the flows of data.

- We contact banking entities to report our found weaknesses and provide simple-but-concrete recommendations. To date, 21 banks have acknowledged our results, and 52 reported weaknesses have been patched by the corresponding banks.

To the best of our knowledge, we are the first to provide systematic study to bridge the gap between banking entities and security researchers from academia toward automatically assessing security weaknesses of apps. We hope that our study can provide insights and experience for RegTech companies and therefore stakeholders can understand the characteristics and consequences of security risks brought by FinTech.

Ethical Considerations. In this paper, we carefully designed our experiments and conducted attacks to avoid ethical issues. We demonstrated proof-of-concept exploits in a restricted local environment, in which we did not eavesdrop on a public network or intercept messages. We ensure that these exploits did not send any forged message to bank servers to cause any unexpected damage or do any harm to ordinary users. Furthermore, we have anonymized all the names of banking entities with specific security weaknesses, so that only the authors know their identities. Finally, we have reported all our findings and detailed security risk reports to the related banks, expecting them to enhance the security of their apps.

2 Taxonomy of Security Weaknesses within Banking Apps

In this section, we first introduce banking apps, then develop and integrate a comprehensive baseline of security weaknesses. We finally propose our threat model.

2.1 Banking Apps

Banking apps, different from ordinary apps, have their own architecture paradigms to achieve financial transactions [22]. Fig. 1 shows the basic architecture, distilled by our investigation on the customer guidelines and their corresponding implementations. A banking app is generally composed of two layers, i.e., the GUI layer (interact with users) and the business logic layer (perform financial transactions). The app communicates with the bank server to commit changes via HTTPS. During the operation of a banking app, *sensitive data* (e.g., identity, credentials) may be manipulated by one single module, flowing from one module to another, or sent to the server via communication channels. Here, the data are considered *sensitive*, unless being prevented from leaking or hijacking by third parties, free from any loss

TABLE 1
Taxonomy of security weaknesses in our study and comparisons with existing tools and research works.

Weakness Category	Security Weakness Type		AUSERA	QIHOO360	ANDROBUGS	MOBSF [7]	QARK [8]	Reaves [12]	Castle [13]	Parasa [14]
(C1) Input Harvest	Sensitive data (e.g., credentials) harvested via screenshots		●		◐					
(C2) Data Storage	Sensitive data (e.g., PIN) stored in shared preferences		●					○		
	Password stored in webview.db		●							
	Sensitive data (e.g., firstname) logged		●			◐		○	○	
	Sensitive data (e.g., transfer history) stored on SD Card		●							
	Sensitive data (e.g., testing user info) written in text file		●							
(C3) Data Transmission	Sensitive data (e.g., pincode) transmitted via SMS		●					○	○	
	ICC leaked	via dynamically registered Receiver	●	◐						
		via implicit Intent								
		via Component export			◐		◐			
(C4) 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 DES/Blowfish encryption	●					○		○
		uses improper function (e.g., ECB mode)	●	◐		◐	◐			
	Uses improper RSA encryption	no RSA	●							
		uses improper function (e.g., NoPadding)	●							
	Uses insecure SecureRandom (i.e., setSeed)		●			◐	◐			
	Uses insecure hash function (i.e., MD5, SHA-1)		●			◐				

●: Automated flow analysis ◐: Syntax-based scanning ○: Manual analysis ◉: Automated analysis by MALLODROID [17]

of confidentiality, integrity, and availability (CIA [23]) in financial services.

2.2 Taxonomy of Security Weaknesses

We develop and integrate security weaknesses for Android apps from prior research [12, 13, 14, 15], best industrial practice guidelines and reports (e.g., OWASP [24], Google Android Documentation [25], and AppKnox security reports [2, 3]), NowSecure reports [26], and security weakness and vulnerability databases (e.g., CWE [27], CVE [28]). We take an in-depth look at the specific key businesses of banking apps, mainly focusing on weaknesses w.r.t. sensitive data, since the biggest threat to banking apps comes from manipulation of digital assets and routine financial activities. As such, we elicit sensitive data associated with contained businesses in banking apps, and identify potential attack surfaces (i.e., exploitable points) in terms of sensitive data's existence states. As shown in Table 1, sensitive data may be exposed to attackers at the input end (i.e., *Input Harvest*), saved into persistent storage (i.e., *Data Storage*), transmitted within one app (i.e., *Data Transmission*), and sent to the remote (i.e., *Communication Infrastructure*). Based on these attack surfaces, we develop and integrate a baseline to assess security risks of banking apps (Column 1 of Table 1).

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 [29].

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.

Data Transmission. Sensitive data transmission via SMS can be easily intercepted by malware observing the outbox of Android SMS service. So if banking apps resort to SMS as a vehicle of sensitive data, an attacker can easily harvest them, and further compromise financial accounts. 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, dynamic registration of a broadcast `Receiver`, or component export.

Communication Infrastructure. MITM attack, as the main threat to communication infrastructure, 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.

Columns 3∼7 show the weakness points addressed by our proposed system, AUSERA, and the state-of-the-practice tools (i.e., QIHOO360, ANDROBUGS, QARK [8] and MOBSF [7]). Our baseline is comprehensive by incorporating multiple categories and sets up a solid foundation for analyzing weaknesses for banking apps.

2.3 Threat Model

On top of the security risk baseline, we propose a threat model. We assume two types of adversaries as follows:

- **App Adversary.** We assume that, due to both the security weakness presented in C1∼C3 of Table 1 and a crafted malicious app pre-installed by an app adversary, all sensitive data that leak to these sinks can be ultimately obtained.
- **Communication Adversary.** We assume that adversaries are able to obtain all communication data be-

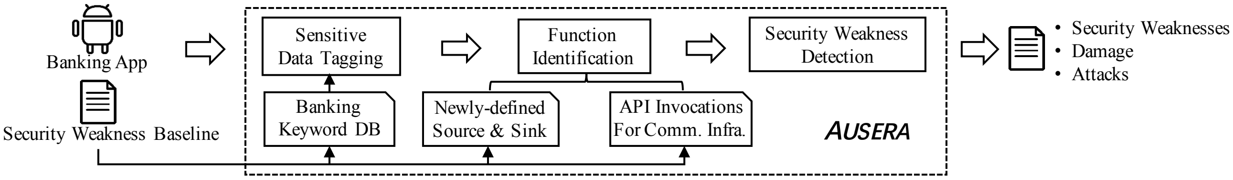


Fig. 2. The overview of AUSERA

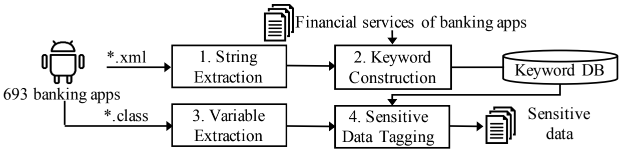


Fig. 3. Identification of sensitive data

tween banking apps and the corresponding bank servers. If the communication data are plain text without encryption, adversaries can immediately read them and subsequently subvert banking apps. If the communication data are encrypted with weak authentication or vulnerable cryptographic algorithms, it is still possible for adversaries to crack communication in practice. All these aforementioned scenarios lead to the security weakness of C4 in Table 1.

3 AUTOMATED SECURITY RISK ASSESSMENT

In this section, we propose AUSERA, a three-phase detection system, to automatically assess security risk of banking apps.

3.1 Approach Overview

AUSERA takes as input each app $\boldsymbol{a} \in \boldsymbol{A}$ (i.e., whole dataset of banking apps), and the set of sensitive data $\boldsymbol{D}$, the set of sinks $\boldsymbol{S}$, the set of vulnerable code $\boldsymbol{C}$, and the set of entry points $\boldsymbol{E}$, guided by the baseline weaknesses $\boldsymbol{B}$, and ultimately outputs the set of security weaknesses, damage, and potential attacks $\boldsymbol{l}_a \in \boldsymbol{L}$ of the app $\boldsymbol{a}$. However, this problem is challenging: (1) we need to detect security weaknesses for *domain-specific apps* (i.e., banking apps) instead of generic ones; and (2) we need to provide *precise, developer-actionable weakness reports* — reduce as many false positives as possible and provide context-aware information for weakness confirmation. To this end, AUSERA proceeds in three phases (see Fig. 2):

- **Sensitive data tagging**, which identifies sensitive data contained in banking apps, and assigns the semantics to declared variables in code w.r.t. the business logic.
- **Function identification**, which concentrates on the very code w.r.t. key modules in banking apps. Based on API invocations (or their call sequence patterns), AUSERA determines the behavior of a piece of code.
- **Weakness detection**, which takes as input the tagged sensitive data and functions, and performs static analysis on the banking apps to check if any weaknesses presented in the proposed baseline are included.

3.2 Sensitive Data Tagging

As discussed in Section 2, we are concerned with the sensitive data enclosed in the bank's key businesses that may incur security risks. Therefore, in this section, we tag data (in the format of variable) in code with sensitivity according to its indicated text [30]. As a result, we derive a list of sensitive data that is relevant to banking apps' core financial services. "Financial services of banking apps" are services listed in Fig. 1 (e.g. Login, Register, and Payment). We view these services as guidance to manually extract keywords that are relevant to banking apps' core functionalities. The technique is detailed as follows.

Fig. 3 shows the four steps: (1) *string extraction* to extract all strings in xml files and `R.class` by reverse engineering. Generally, the string presented in a `TextView` describes the semantics of the content in its subsequent element, e.g., an `EditText` or another `TextView`. To extract both inputs and transfer data from bank server, we consider the mapping of $\{\langle$`TextView`, `EditText`$\rangle$, $\langle$`TextView`, `TextView`$\rangle\}$. For example, account balance should be labeled as $\langle$"`text_balance`," data from bank server$\rangle$. Thus, we extract such a kind of strings as *corpus* to infer the semantics of displayed data in context; (2) *keyword construction* to extract sensitive keywords from this corpus with NLP techniques. To avoid missing similar keywords and to remain the latent semantics among keywords, we further employ WORD2VEC [31] to supplement the corpus of keywords by loading `.bin` word vector trained model, utilizing the sentences extracted by SUPOR [30] from 54,371 apps. These keywords are able to tag the semantics of the view elements in xml files. After that, we construct a keyword database for sensitive data tagging; (3) *variable extraction*, which elicits variables defined in Java classes; (4) *sensitive data tagging*, which glues the variables and the tagged view elements together. One variable in code binds to a specific view element by invoking method `findViewById()`. As a result, sensitive data are tagged with its semantics in the format of $\langle$`variable, keyword`$\rangle$. AUSERA extracts 70 keywords in total from 693 banking apps to test, and these keywords can be categorized into four types as shown in Table 2. The full list of keywords is publicly available online.[1]

In 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 that the example is confirmed to send sensitive data via SMS.

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

TABLE 2
Keyword examples

Category	Keyword Examples	#
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

3.3 Function Identification

In this section, we utilize static analysis to identify function code that is associated with weaknesses for banking services. Such function code is categorized into two types:

Sink of Sensitive Data. The sensitive data extracted in Section 3.2 includes user inputs and data from the bank server, which can be obtained via `getText()` or `findViewById()`. We define them as *sources*. These sensitive data could be far apart from the access of unauthorized users. However, as discussed in Section 2, this 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 based on C2 and C3 in Table 1 (a.k.a., sink of sensitive data). It is worth mentioning that the sinks here are different from the sinks defined in SuSi [32]. SuSi's sinks are calls into any resource method (e.g., `getDeviceId()` and `getLatitude()`) that leaks sources out of mobile devices, such as `URLConnection#openConnection` and `HttpResponse#execute`, while our sinks are leaking sensitive data to other malicious apps or physically connected users, by breaking the sandbox policy on Android [33].

We identify 106 vulnerable sinks [34] in total that are likely to be exploited. The process of functional code identification is performed by using static code analysis. For example, `putString` of class `SharedPreferences.Editor` object saves data to local `shared_prefs` and could potentially be accessed by other unauthorized users. In this paper, we consider two types of sinks (i.e., storage and transmission) as insecure weaknesses, covering the 106 vulnerable ones.

Communication Infrastructure. Communication infrastructure is indispensable to banking apps [12, 16]. It establishes a channel to communicate with remote bank servers. However, communication infrastructure is likely to be attacked, and hence it can subvert banking 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 has applied hostname verification during communication. Then, we will check its implementation. We have in total 12 groups of API invocation patterns for communication infrastructure. We reverse-engineer banking apps, and locate the invocations of these relevant Android APIs. We use call graphs and component transitions to determine their call relationship in between. After all these steps, we can identify the functional code for communication infrastructure of banking apps.

ALGORITHM 1: Security Weakness Detection

```
Input: Each mobile banking app under analysis $a \in A$; The set of
       sensitive data $D$; The set of sinks $S$; The set of vulnerable code
       $C$; The set of entry points $E$
Output: The set of weaknesses $l_a \in L$ of the app $a$
1  Let $G$ be the control-flow graph of the app $a$ and $l_a$ be $\emptyset$
2  foreach Sensitive data $d \in D$ do
       // Identify all paths that $d$ flows in
3      $paths \leftarrow ForwardDataFlowAnalysis(d, G)$
4      foreach Path $p \in paths$ do
5          Let $s$ be the last statement in $p$
6          if $s \in S$ then
7              $l_a \leftarrow l_a \cup \{p\}$
8  foreach Vulnerable code $c \in C$ do
       // Identify all paths that reach $c$
9      $paths \leftarrow BackwardControlFlowAnalysis(c, G)$
10     foreach Path $p \in paths$ do
11         Let $s$ be the first statement in $p$
12         if $s \in E$ then
13             $l_a \leftarrow l_a \cup \{p\}$
14 return $l_a$
```

3.4 Security Weakness Detection

Given a banking app, we attempt to find whether it contains any weaknesses listed in Table 1. We employ two strategies to detect weaknesses: a forward data flow analysis to determine whether there exist sensitive data flowing into insecure sinks; a backward control flow analysis to check whether the vulnerable functional code (API invocation patterns) in communication infrastructure is truly reachable.

We carry on a forward data-flow analysis on top of Soot [35] by supporting intra- and inter-component communication analysis. The sources are tagged sensitive data in Section 3.2, and the sinks are the list of Android APIs presented in Section 3.3. In the process of functional code identification, we can obtain all vulnerable code that exists in communication infrastructure. However, the noise may rise because the dead code for testing purpose de facto cannot be executed during runtime. Reaves et al. [12, 15] 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 component transitions. If the vulnerable code is reachable, we determine it is a valid weakness, or otherwise.

Algorithm 1 shows the process of weakness detection. The input of the algorithm is a set of banking apps, containing sensitive data, sinks, vulnerable code, all possible entry points (there are basically two types of entry points: *user input* [36] and *system event* [37]). Algorithm 1 first proceeds with data leakage detection (lines 2-7). It figures out that all data-flow paths start by accessing sensitive data. If this path ends up with an insecure sink, we regard it as a leakage. Second, it conducts a reachability analysis (lines 8-13). Conversely, we backtrack the vulnerable code and identify its entry point. If the entry point is a feasible entrance leading to the vulnerable code, we regard it as a true weakness.

TABLE 3
Distribution of the evaluated banking apps

Continent	#Developed	#Developing	Total
Europe	102	0	102
America	53	24	77
Asia	16	210	226
Oceania	16	0	16
Africa	0	49	49
Total	187	283	470

3.5 Implementation of AUSERA

To implement AUSERA, we combine NLP techniques and static analysis to identify sensitive data of banking apps, and associate them with the corresponding variables in code. AUSERA relies on APKTOOL [38] to extract resource files (e.g., layout files and `R.class`) from apks. It then uses parts-of-speech (POS) tagger of APACHE OPENNLP-1.8.3 [39] to parse the text labels in `TextView`, thereby identifying keywords included. We manually check on these keywords to retain ones that are sensitive and relevant to core functionalities of banking apps. After that, we employ WORD2VEC to supplement the keyword database by: (1) utilizing the sentences extracted by SUPOR from 54,371 Google Play apps to train a `Bin` model; (2) extending sensitive keywords by the model; (3) filtering out as many non-sensitive keywords as possible, to raise the accuracy of tagging.

To accomplish the detection, we summarize 12 groups of semantic patterns (e.g., AES/ECB/NoPanding) to depict the communication weaknesses. Then we employ a lightweight static analysis to find the possible vulnerable patterns in code. We obtain server certificates from package, and utilize `X509Certificate` to parse their internal information (e.g., sign algorithm and term of validity). We check three aspects for certificate authentication: whether the client side allows all hostname requests; bypasses hostname verification; or fails to implement anything in 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. We also check whether an encryption key is embedded in apk package. The banking sensitive data are encrypted with the DES or Blowfish algorithm. Using either of the encryption mechanisms is viewed as a weakness [12, 15]. The AES forbids ECB mode because it does not provide a general notion of privacy [16]. 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 [40, 41].

4 EXPERIMENTS AND EVALUATION

We evaluate AUSERA's effectiveness on the banking apps collected. Based on the experimental results, we intend to answer the following research questions:

RQ1. How effective is AUSERA to detect weaknesses per se as well as in comparison with other state-of-the-practice tools?

TABLE 4
Weaknesses in 470 banking apps

Weakness Category	Weakness Type	#Affected Apps
Input Harvest	Screenshot	415 (88.3%)
Data Storage	`SharedPreference`	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%)

RQ2. What insights can be gained from those weaknesses exposed to FinTech, such as production process, root causes, and regulatory compliance?

RQ3. How do industrial representatives respond to weaknesses, and what are the gaps of understanding weaknesses between industry and academia?

In this paper, we answer **RQ1** in Section 4 by elaborating the found weaknesses on a large-scale banking apps, comparing with 4 state-of-the-practice tools, and illustrating the exploitation of 4 typical weaknesses. **RQ2** is answered in Section 5.1 and 5.2 with demographic and longitudinal analysis. To answer **RQ3**, we study common practices of banking app production from industry, and then uncover the gaps and propose recommendations in Section 5.3.

Dataset. We collected totally 693 banking apps (without using packer techniques) across 470 unique banking entities, where some apps have multiple versions. These apps originate from both developed and developing countries across five continents (see breakdowns in Table 3). If banking apps have packer and decompilation failure when using SOOT, we thereby leave them out of our study. Table 3 indicates that 48.1% of the banking apps are from Asia, considering the largest population proportion (59.1%) all over the world, and only 3.4% of apps are from Oceania, considering its smallest population proportion (0.5%) 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 originate from Singapore, Japan, and South Korea. To our knowledge, this is the largest banking app dataset taken into study to date.

4.1 Evaluation on the Security Weakness Baseline

We execute AUSERA on these 470 unique banking apps to evaluate its effectiveness and efficiency, because the multiple versions of a banking app have many overlaps of weaknesses. Table 4 shows results of weaknesses that correspond to the security baseline defined in Section 2.

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 `Preference`, `Logging`, and `WebView DB` are the main channels that leak sensitive data. 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 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.

Communication Infrastructure. The protection of communication infrastructure in banking apps is far 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 connected servers. We find 222 banking appswith 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. Insecure Hash Function (72.3%) is also frequently misused.

In summary, we show that 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.

4.2 Precision of AUSERA

We randomly selected 60 banking apps (12.8%) in our dataset and manually checked the detection results to evaluate AUSERA's precision, which always equals accuracy in our case. False positive (FP) is referred to as any weakness that are detected during static analysis but actually unreachable at runtime or completely mistaken. As a result, we only found 6 false positives (corresponding to five weakness types, i.e., `Preference` Leakage, Logging Leakage, SD Card Leakage, Text File Leakage, and Hard-coded Key) from the identified 341 weaknesses of these 60 banking apps, leading to an average precision of 98.24% for AUSERA.

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

False Positive Analysis. 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 dead code.

TABLE 5
Detection result comparisons

Tools	#Type	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

In particular, we show several specific cases to explain how to incur false positives. Sensitive data disclosure through logging is always detected by MOBSF as shown in Table 1, but MOBSF just matches the following APIs if used (e.g., `Log.e()`, `Log.d()`, and `Log.v()`). There is no doubt that it has incurred plenty of false positives. If the data are not sensitive, such as "`menu_title`," it is very normal for developers to log 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 provides an incomplete and incorrect analysis result, due to the influence of dead code. For example, the tools detected three code blocks violating server verification, i.e., do nothing in `checkServerTrusted`. In contrast, AUSERA aims to reduce as minimal dead code influence as possible. 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 component transitions.

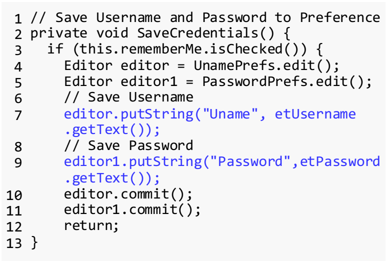
Comparisons with the State of the Practice. We compare the detection results with 4 industrial and open-source tools, including QIHOO360, ANDROBUGS, MOBSF, and QARK. We randomly select 60 banking apps in our dataset for comparisons. All comparisons of the detection results comply with the baseline of weaknesses in Section 2. The precisions for these tools are obtained by manual validation through filtering out all false positives. In addition, we run each tool 3 times to stabilize the detection accuracy.

In summary, AUSERA identifies the largest number of weaknesses, significantly outperforming the other tools in terms of precision. AUSERA manages to scan each app within 1.6 mins on average, much faster than the other tools (Since QIHOO360 is an online scanning service, we eliminate the upload time for a fair comparison.). We conclude that AUSERA is the most efficient system for security risk assessment for banking apps.

4.3 Case Studies of Weaknesses

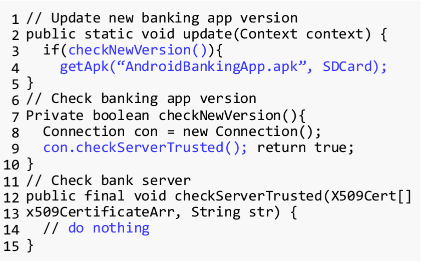
AUSERA has found a number of severe weaknesses, which affect 431 banking apps. To showcase the exploitability of these weaknesses, we selected four vulnerable apps and constructed corresponding proof-of-concept attacks.

Screenshot Weakness. A∗ Bank (v3.3.1.0038) employs two-factor authentication, i.e., a 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 *screenshot* feature), and the verification code can be



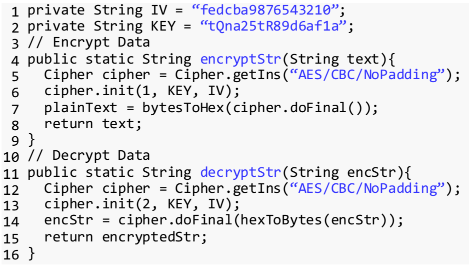
```
1 // Save Username and Password to Preference
2 private void SaveCredentials() {
3   if (this.rememberMe.isChecked()) {
4     Editor editor = UnamePrefs.edit();
5     Editor editor1 = PasswordPrefs.edit();
6     // Save Username
7     editor.putString("Uname", etUsername
      .getText());
8    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

Fig. 4. The 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 }
```

Fig. 5. The 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 }
```

Fig. 6. The simplified code of en/decryption weakness in N∗

accessed with granted permissions. As such, we generate a malicious app that runs a service which will snapshot 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 has bypassed the security vetting of Google Play and is successfully put on the shelf, which makes this attack more practical [42].

Preference Weakness. Fig. 4 shows the vulnerable code of `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 [12, 15]. In either way, the malware can access the victim's sensitive data from 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 MITM risk during version updating of which the code is shown in Fig. 5. The app checks new versions with 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 attacker server (line 4). To exploit this, we use BURP SUITE [43] and FIDDLER [44] to fool the banking app, by sending a malicious app to impersonate the most recent version [45]. After this malicious app is installed, it serves as a phishing app to steal user credentials and other sensitive data.

En/Decryption Attack. AUSERA detects an encryption weakness in N∗ Bank (v1.8) as shown in Fig. 6. 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 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.

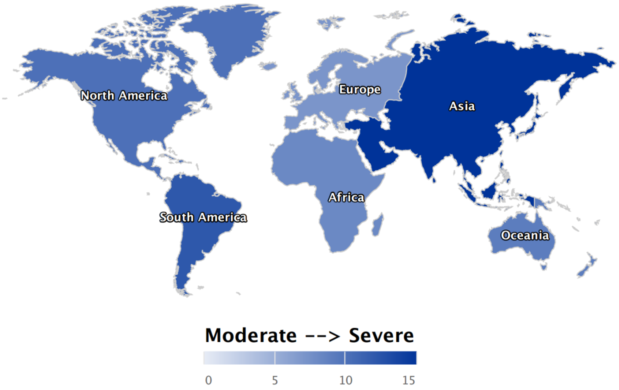


Fig. 7. Number of weaknesses of banking apps across the world

5 MEASUREMENT AND DISCOVERIES

In this section, we take a measurement analysis on top of our detection results to attempt to interpret the nature of security weaknesses of banking apps. We begin by taking a demographic study of banking apps w.r.t. economies and regulations. We then take a deep dive into how app version updates across markets incur weaknesses. We finally try to bridge the gap toward assessing severity of weaknesses among all stakeholders, to propel the mobile banking ecosystem.

5.1 Demographic Study of Apps

Fig. 7 shows a global map of the number of weaknesses discovered among the banking apps by continents. The heat bar encodes the number of weaknesses the apps have, scaled from light blue (least) to dark blue (most). In concrete, we observe the following: (1) Weaknesses of 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) largely have fewer weaknesses than those of developing countries. Ironically, to our surprise, we find that weaknesses of apps of Asian developed countries slightly outnumber (with 6.7 weaknesses per app) than those of Asian developing countries. (2) Strikingly, banking apps of Africa exhibit satisfactory security status, having only 4.6 weaknesses on average, some are even more secure than those of developed countries.

The possible reasons that the security of banking apps varies across regions can be interpreted as follows:

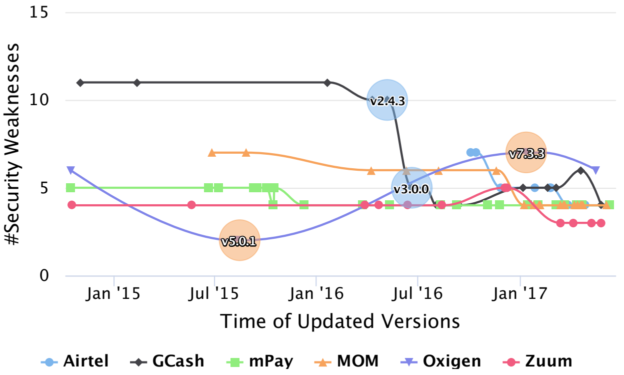


Fig. 8. Number of weaknesses in each update version. The x-axis represents the date of each release version, and the y-axis represents the total number of weaknesses detected in current release version.

- The financial regulations and development guidelines are different across regions, which may affect the implementation. For example, both Europe (GDPR [21]) and USA (PCI DSS [46]) 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 [47]. 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. Perhaps because of inadequate budget for app development, those apps released are prone to being less secure.
- Cashless payment system has been bootstrapped in areas where traditional banking is uneconomical and expensive, ripping out large investment on the massively deployed financial infrastructure. This is evidenced by the fact that Kenya, a country in Africa, sets the world first with money transfers by mobile [48], and 68% people in Kenya report use of phones for a financial service [49].

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

5.2 Longitudinal Analysis of Version Updates & Fragmentation

We try 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. [12], with confirmed weaknesses. We downloaded all available versions of 6 apps (mCoin is excluded since history versions are not publicly available.), with totally 88 different versions, 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.

Fig. 8 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 paper [15] 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 popular weaknesses that remain unfixed. Furthermore, developers usually neglect hostname verification or server authentication, which may bring 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. [12, 15] found that all six vulnerabilities still remain in 2016 updated GCash. However, according to our security reports, GCash has revised most vulnerabilities. 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) are introduced, which have not been discovered by [12]. They compared the code similarity between the 2015 and 2016 versions of each app, and found some apps have significant new code [15]. This is aligned with our study that many banking apps have not received systematic security check before delivery.

Furthermore, we find banking entities 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 [50] 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 Fig. 9). (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) has more weaknesses (e.g., `SharedPreference` Leakage, SD Card Leakage) than that of HSBC (UK) and HSBC (Hong Kong). This is largely because 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

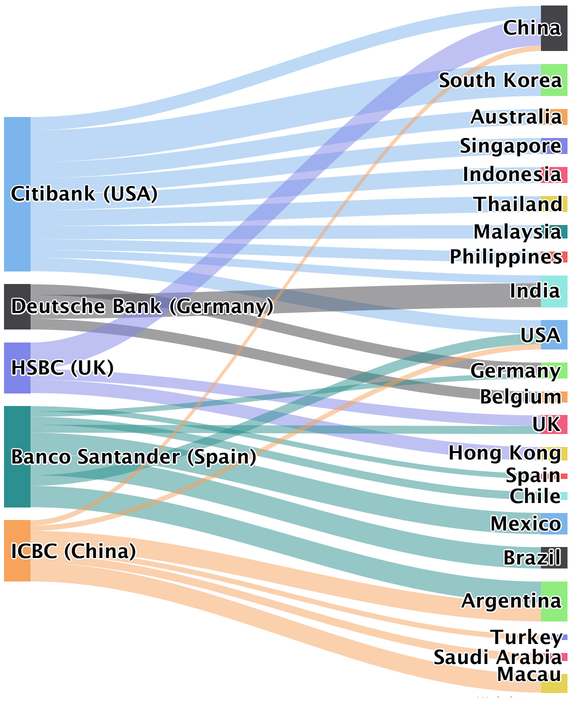


Fig. 9. Flow of security weaknesses from parent banks to subsidiary banks across the world. The flow width indicates the number of weaknesses originating and terminating between two corresponding banks. Different types of banking entities are encoded by different colors.

regulations in terms of the number of banking app security risks (Fig. 9 shows the source and host countries of flows containing security weaknesses.).

> **Remark.** 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.

5.3 Feedback and Recommendations

Our study has uncovered totally 2,157 weaknesses from 693 banking apps, most of which have been reported to the corresponding banking entities. 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 globe, such as HSBC (UK/Hong Kong/Shanghai), OCBC (Singapore), DBS (Singapore), and BHIM (India), to understand their security practice and policies. Through in-depth discussion, we find they hold different mindsets toward assessing severity of weaknesses and setting security goals. 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 [51] to determine the severity of found weaknesses. However, this standard is not perfect in practice [52], 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 concern 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 toward different weaknesses, which are classified by "Concerned" (high priority) and "Aware" (low priority).

TABLE 6
Different concerns from banking entities

	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

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 update version, i.e., the app fails to verify the identity of bank server (`checkServerTrusted`).

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 [53]). On average, these apps have 7.7 different versions, and the most fragmented app has 25 versions. Therefore, we strongly recommend banks push compulsory app updates to the customers, 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-

TABLE 7
Weaknesses tracking of 21 banking apps. "#W" is the number of detected weaknesses. "#Patched" is the number of patched weaknesses in update versions. "#New" is the number of newly-introduced weaknesses in update versions. 16 banks have already patched their banking apps, and the rest have confirmed the weaknesses in their replies and will fix them soon by 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

1, to produce message digests, which have already been accepted as insecure [40, 54]. Banks still use them although they are aware of this fact. They assume that ordinary attackers are not capable of breaking them. However, it is still possible for experienced attackers to mount a massive-scale attack by exploiting these weaknesses. It is banks' liability if they use security-weakened or poisoned third-party libraries without careful inspection. To avoid "amplification effect" caused by the weaknesses in third-party libraries [55], we strongly recommend banks to carefully inspect third-party libraries in use.

Remark. The contemporary incomplete security criteria provide banks wide leeway to use their 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 are unfixed for weeks to months post-disclosure. This gap provides windows of opportunity for not only sophisticated but also ordinary attackers with the knowledge and time to strike. We believe Ausera can help reduce this gap by expeditiously informing banks of assessing their security risks on apps, further generating patches, and verifying their security.

6 Related Work

Security Analysis of Android Apps. Taint analysis is a commonly-used method to reveal potential privacy leakage in Android apps. For example, TaintDroid [56] is a dynamic taint-tracing tool which tracks flows of private data by modifying Dalvik virtual machine; FlowDroid and IccTA [57, 58] are both static taint analysis tools that accept the *source* and *sink* configurations for privacy leaks. However, these tools target on general apps, and thus may not be able to unveil specific security weaknesses (summarized in Table 1) when applied for banking apps.

Security Assessment of Banking Apps. In 2015, Reaves et al. [12] 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 [15]. Chanajitt et al. [59] 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 [12, 15, 59] with regards to the scope of measurement. Whereas [12, 15, 59, 60] mainly leverage case studies to study banking apps, the focus of our paper is to develop novel automated detection techniques. 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 [61, 62], performance [63] and fragmentation [64].

Demographic Analysis of Banking Apps. Castle et al. [13] 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. [65] 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. [14] studied 9 mostly-used mobile money apps across 9 Australasian countries, and reported the 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. [66] 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.

Encryption and Authentication of Android Apps. SSL issues have been widely discussed in [67], which suggests revisiting the SSL handling in applied platforms (e.g., iOS and Android). Followed by recent reports [14, 65] and our observation, we find that many banking apps have fairly weak or even no authentication and encryption mechanisms. Sounthiraraj et al. [19] proposed to combine static and dynamic analysis to identify security problems in SSL/TLS for Android apps. Georgiev et al. [18] 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. [16] 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 points, and examine whether banking apps contain these points.

7 Conclusion

In this paper, we developed and integrated a set of baseline security weaknesses to assess the security risk of banking apps. Based on the baseline, we implemented a three-phase detection system, termed Ausera, to automatically identify weaknesses, achieved with a precision of over 98%. Despite effective and efficient, we acknowledge that Ausera may bring false positives since NLP and static analysis are not 100% accurate, which renders human inspectors to reduce false positives. We also emphasize if an app has packer protection and decompilation failure (e.g., obfuscation and native code), then the app is not checkable by the methods developed in this paper.

Our findings have revealed that 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 hold true. Our communications with banking agencies show that the timing of responsible disclosure does not closely align with the date a patch is applied to the app, providing windows of opportunity for attacker exploitation.

These findings highlight the importance of the design of automated security risk assessment systems, considering different types of weaknesses and regulations. By leveraging these insights, we hope that the RegTech companies and all stakeholders can progress in improving the risk assessment and patching process for securing contemporary FinTech.

References

[1] 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.

[2] Prateek Panda. Security Report of Top 100 Mobile Banking Apps-APAC. Technical report, Appknox, October 2015.

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

[4] Douglas W Arner, Jànos Barberis, and Ross P Buckley. Fintech and Regtech in a Nutshell, and the future in a Sandbox. *Research Foundation Briefs*, 3(4):1–20, 2017.

[5] Qihoo360 (Appscan). http://appscan.360.cn/, 2017.

[6] Androbugs. https://github.com/AndroBugs/, 2015.

[7] Mobile-Security-Framework-MobSF. https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2017.

[8] QARK: Tool to look for several security related Android application vulnerabilities. https://github.com/linkedin/qark, 2017.

[9] Banks targeted by SMS phishing scam. http://www.acma.gov.au/theACMA/engage-blogs/engage-blogs/Cybersecurity/Banks-targetted-by-SMS-phishing-scam, 2016.

[10] 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.

[11] 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] Bradley Reaves, Nolen Scaife, Adam M Bates, Patrick Traynor, and Kevin RB Butler. Mo (bile) Money, Mo (bile) Problems: Analysis of branchless banking applications in the developing world. In *USENIX Security*, pages 17–32, 2015.

[13] Sam Castle, Fahad Pervaiz, Galen Weld, Franziska Roesner, and Richard Anderson. 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*, page 4. ACM, 2016.

[14] Swathi Parasa and Lynn Margaret Batten. Mobile money in the Australasian region-A technical security perspective. In *International Conference on Applications and Techniques in Information Security*, pages 154–162. Springer, 2016.

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

[16] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[17] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 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*, pages 50–61. ACM, 2012.

[18] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 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*, pages 38–49. ACM, 2012.

[19] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 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, 2014.

[20] Tom Chothia, Flavio D Garcia, Chris Heppel, and Chris McMahon Stone. 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*, pages 579–597. Springer, 2017.

[21] The EU General Data Protection Regulation. https://www.eugdpr.org/, 2017.

[22] Typical Mobile Banking Services. https://en.wikipedia.org/wiki/Mobile_banking, 2017.

[23] Jason Andress. *The basics of information security: Understanding the fundamentals of InfoSec in theory and practice*. Syngress, 2014.

[24] OWASP: OWASP Mobile Security Project. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10, 2 2017.

[25] Google Best Practices for Security & Privacy. https://developer.android.com/training/best-security.html, 2 2017.

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

[27] CWE: Common Weakness Enumeration. https://cwe.mitre.org/, 2018.

[28] CVE: Common Vulnerabilities and Exposures. https://cve.mitre.org/, 2018.

[29] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. Screenmilker: How to milk your Android screen for secrets. In *NDSS*, 2014.

[30] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. SUPOR: Precise and scalable sensitive user input detection for Android apps. In *USENIX Security Symposium*, pages 977–992, 2015.

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

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

[33] SandBox Security Policy. https://getsandbox.com/policy/security, 2017.

[34] AUSERA. https://sites.google.com/view/ausera/, 2018.

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

[36] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving Android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 365–376. ACM, 2016.

[37] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering*, pages 303–313, 2015.

[38] Apktool: A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/, 2017.

[39] Apache OpenNLP 1.8.3. https://opennlp.apache.org/news/release-183.html/, 2018.

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

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

[42] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *Computers & Security*, 73: 326–344, 2018.

[43] Burp Suite. https://portswigger.net/burp, 2017.

[44] Fiddler: Free Web Debugging Proxy - Telerik. http://www.telerik.com/fiddler, 2017.

[45] 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.

[46] PCI: Security Standards Council. https://www.pcisecuritystandards.org/, 2017.

[47] European Parliament and Council of the European Union. 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*, 281:0031–0050, 1995.

[48] Kenya sets world first with money transfers by mobile. https://www.theguardian.com/money/2007/mar/20/kenya.mobilephones, 2007.

[49] Kenya tops Africa in use of mobile financial services. http://kenyanwallstreet.com/kenya-tops-africa-use-mobile-financial-services-report, 2017.

[50] 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.

[51] The Common Vulnerability Scoring System. https://www.first.org/cvss/, 2018.

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

[53] Apkmonk: Download Android apps apks directly from Apkmonk without installing any other app or playstore. https://www.apkmonk.com/, 2018.

[54] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 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*, pages 570–596, 2017.

[55] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 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*, pages 2187–2200, 2017.

[56] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*.

[57] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices*, 49 (6):259–269, 2014.

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

[59] Rajchada Chanajitt, Wantanee Viriyasitavat, and Kim-Kwang Raymond Choo. Forensic analysis and security assessment of Android m-banking apps. *Australian Journal of Forensic Sciences*, 50(1):3–19, 2018.

[60] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 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*, pages 797–802. ACM, 2018.

[61] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-scale analysis of framework-specific exceptions in android apps. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 408–419, 2018. ISBN 978-1-4503-5638-1.

[62] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. Efficiently manifesting asynchronous programming errors in android apps. *arXiv preprint arXiv:1808.03178*, 2018.

[63] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, 2014. ISBN 978-1-4503-2756-5. URL http://doi.acm.org/10.1145/2568225.2568229.

[64] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 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, pages 226–237, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5.

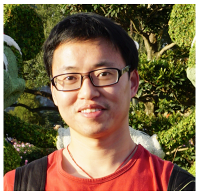
[65] Kiron Lebeck, Temitope Oluwafemi, Tadayoshi Kohno, and Franziska Roesner. Rethinking Mobile Money Security for Developing Regions. Technical report, University of Washington, 2015.

[66] VF Taylor and I Martinovic. 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*, 2017.

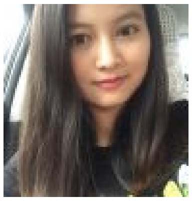
[67] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 49–60. ACM, 2013.

Sen Chen is pursing his Ph.D. degree at the Department of Computer Science and Software Engineering of East China Normal University. He is focusing primarily on areas of mobile security, Android malware, Android vulnerability, and program analysis. He also is a visiting scholar of Nanyang Technological University, Singapore. He is currently advised by Professor Lihua Xu (NYU Shanghai) and Yang Liu (NTU). More information is available on http://www.sqslab.com/senchen/.

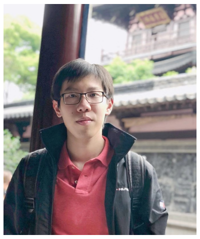
Guozhu Meng is currently an associate professor with Chinese Academy of Sciences. He received Bachelor and Master degree in school of computer science and technology from Tianjin University, China in 2009 and 2012. He worked in Temasek lab in National University of Singapore for one year as an Associate Scientist. Since 2013, he received his Ph.D degree in school of computer science and engineering of Nanyang Technology University, Singapore in 2017. His research interests include mobile security, software engineering and program analysis.



Ting Su received his B.S. in software engineering and Ph.D. in computer science from School of Computer Science and Software Engineering, East China Normal University, Shanghai, China, in 2011 and 2016, respectively. Currently, he is a postdoctoral research fellow in School of Computer Science and Engineering, Nanyang Technological University, Singapore. Previously, he was a visiting scholar of University of California, Davis, USA from 2014 to 2015. His research focuses on symbolic execution and mobile applications. He has published broadly in top-tier programming language and software engineering venues, including PLDI, ICSE, FSE and CSUR.



Lingling Fan received her B.S. in computer science from School of Computer Science and Software Engineering, East China Normal University, Shanghai, China in 2014, and now is pursuing her Ph.D. in the same school. Her research focuses on software testing and Android application analysis. Now, she is a visiting scholar of Nanyang Technological University, Singapore. She is currently advised by Professor Lihua Xu (NYU Shanghai) and Yang Liu (NTU).



Minhui Xue is currently a Research Fellow with Optus Macquarie University Cyber Security Hub and a visiting research scientist with Data61-CSIRO at Sydney, Australia. His current research interests are security and privacy. He is the recipient of the ACM SIGSOFT distinguished paper award and IEEE best paper award, and his work has been featured in The New York Times. He is on the PC committee of PoPETS 2019.

Yinxing Xue is currently a research professor (Pre-Tenure) with University of Science and Technology of China. He received the B.E. and M.E. degree from Wuhan University, China. In 2013, he received the Ph.D. degree in computer science from National University of Singapore (NUS). Since Feb. 2013, he had been a Research Scientist with Temasek Laboratories, NUS. Since Jan. 2015, he has been a Research Scientist with Temasek Laboratories, NTU. His research interest includes software program analysis, software product line engineering, cyber security issues (e.g., malware detection, intrusion and vulnerability detection).

Yang Liu graduated in 2005 with a Bachelor of Computing (Honours) in the National University of Singapore (NUS). In 2010, he obtained his PhD and started his post doctoral work in NUS, MIT and SUTD. In 2011, Dr. Liu is awarded the Temasek Research Fellowship at NUS to be the Principal Investigator in the area of Cyber Security. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang Assistant Professor. He is currently an associate professor and the director of the cybersecurity lab in NTU.

He specializes in software verification, security and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. His work led to the development of a state-of-the-art model checker, Process Analysis Toolkit (PAT). By now, he has more than 200 publications and 6 best paper awards in top tier conferences and journals. With more than 20 million Singapore dollar funding support, he is leading a large research team working on the state-of-the-art software engineering and cybersecurity problems.

Lihua Xu received her Ph.D. in Computer Science in 2009, from the University of California at Irvine. She is currently an associate professor at New York University Shanghai and East China Normal University. Before joining NYU Shanghai as part of its overseas talent introduction program in 2012, Lihua held tenure-track assistant professor position at Rochester Institute of Technology. Her current research focuses on software engineering, automated software analysis and testing, mobile security.