Understanding the (In)Security of Cross-side Face Verification Systems in Mobile Apps: A System Perspective

Xiaohan Zhang*, Haoqi Ye*, Ziqi Huang*, Xiao Ye*, Yinzhi Cao†, Yuan Zhang*, Min Yang*
*Fudan University, Shanghai, China, {xh_zhang, hqye20, huangzq21, xiaoye21, yuanxzhang, m_yang}@fudan.edu.cn
†Johns Hopkins University, Baltimore, USA, yzcao@cs.jhu.edu

***Abstract*—Face Verification Systems (FVSes) are more and more deployed by real-world mobile applications (apps) to verify a human's claimed identity. One popular type of FVSes is called cross-side FVS (XFVS), which splits the FVS functionality into two sides: one at a mobile phone to take pictures or videos and the other at a trusted server for verification. Prior works have studied the security of XFVSes from the machine learning perspective, i.e., whether the learning models used by XFVSes are robust to adversarial attacks. However, the security of other parts of XFVSes, especially the design and implementation of the verification procedure used by XFVSes, is not well understood.**

In this paper, we conduct the first measurement study on the security of real-world XFVSes used by popular mobile apps from a system perspective. More specifically, we design and implement a semi-automated system, called XFVSCHECKER, to detect XFVSes in mobile apps and then inspect their compliance with four security properties. Our evaluation reveals that most of existing XFVS apps, including those with billions of downloads, are vulnerable to at least one of four types of attacks. These attacks require only easily available attack prerequisites, such as one photo of the victim, to pose significant security risks, including complete account takeover, identity fraud and financial loss. Our findings result in 14 Chinese National Vulnerability Database (CNVD) IDs and one of them, particularly CNVD-2021-86899, is awarded the most valuable vulnerability in 2021 among all the reported vulnerabilities to CNVD.

I. INTRODUCTION

Due to recent advances in face recognition techniques, Face Verification Systems (FVSes) are now being more and more deployed and used to verify a human's claimed identity. During the era of the global pandemic, one popular use case of FVS, called cross-side FVS (XFVS), is to take a picture or a short video of a target person using his/her mobile phone (potentially untrusted) and then upload them to a trusted server for verification. Such an XFVS is different from traditional use cases such as local verification inside a trusted execution environment (TEE), e.g. iPhone's FaceID [1], or fully remote verification inside trusted physical environments like government facilities [2]. More specifically, many popular mobile apps, such as WeChat (the world's largest standalone mobile app) and Alipay (the most popular online payment app in China), offer XFVSes for user verification in sensitive operations, e.g. changing payment password.

An XFVS needs to first collect the target person's face data and then verify his/her identity, which usually takes "two phases": (i) liveness detection and (ii) identity verification. The former is to ensure that the target person in front of the camera is alive instead of a replayed video or a static picture, while the latter is to verify the target person's identity by comparing the collected face with the pre-registered reference face. Currently, both these two phases usually require the use of machine learning-based models. For example, models are used in liveness detection to recognize face movements or detect how the user's face reflects different colors of light.

Prior works have studied the attacks against XFVSes (and other FVSes) from the perspective of adversarial machine learning (ML), including presentation attacks [3], [4], [5], [6] and adversary attacks [7], [8], [9], [10]. For example, Li et al. [11] show that liveness detection can be fooled by a deep-faked video crafted from the target person's images. In response to these attacks, researches [12], [13], [14] have also proposed corresponding defensive methods. For example, Tang et al. [15] utilize face textual features and reflection time to defend 2D presentation attacks.

However, despite the previous success in studying the ML-driven attacks against XFVSes and the escalating arms race between defenses and attacks, one largely ignored perspective of XFVS security is the system design. In other words, it remains unclear whether the verification protocol adopted by XFVSes is secure due to the involvement of complex cross-sided two-phase face verification. From the ML perspective, the evasion bar is becoming higher and higher with low attack accuracy due to more advanced countermeasures. However, from the system perspective, the evasion bar is still low because the existence of one logic vulnerability in the two-phase face verification protocol will make *all* parts vulnerable to adversaries, as indicated by the famous *wooden barrel theory*.

In this paper, we conduct the first systematic measurement study on the security of real-world XFVSes used by popular mobile apps from the system perspective, i.e., understanding the security of the cross-sided two-phase face verification. More specifically, we design and implement a semi-automated testing framework, called XFVSCHECKER, to detect and analyze the security of XFVSes inside Android apps. By thoroughly studying XFVS code and documents, we summarize a generic workflow of XFVSes and pinpoint four security properties an XFVS should satisfy: (i) SP1. reliable

environment, (ii) SP2. camera security, (iii) SP3. reliable liveness, and (iv) SP4. data consistency. XFVSCHECKER then uses explicit evaluation procedures and scoring rules to assess how XFVS apps comply with these security properties.

We evaluate XFVSCHECKER on 43,422 Android apps from Google Play and Xiaomi App Market, which reveals that 5.1% of apps adopt XFVSes, with the top SDKs making up the majority of the XFVS ecosystem. The security analysis on top SDKs shows that most existing real-world XFVS apps—even top ones with billions of downloads including the aforementioned WeChat and Alipay—are vulnerable to at least one of 4 different attacks, i.e., Liveness Bypassing Attack (AT1), AB Attack (AT2), Camera Hijacking Attack (AT3) and Downgrade Attack (AT4). To the best of our knowledge, XFVSCHECKER is the first system to study all these attacks except for AT3 [11]. Moreover, XFVSCHECKER is the *first* to find real-world apps that are vulnerable to these four types of attacks including AT3.

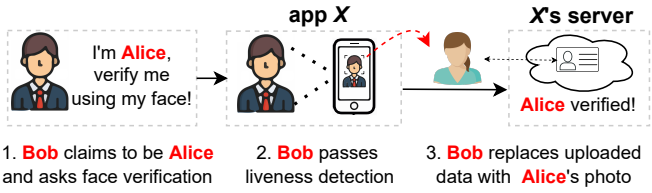


Fig. 1. An Attack Example on XFVSes in Mobile Apps.

Fig. 1 illustrates an example of AT2 (AB attack) found in several apps, such as *Pingan Jinguanjia*, one popular insurance app in China with 240 million users [16]. Specifically, Bob, the adversary, uses his face to pass the first phase (liveness detection) and then adopts a photo of Alice, the victim, to pass the second phase (identity verification). This is a typical case where *SP4. data consistency* is not satisfied, so that the face matched is not the face that passes liveness detection. These attacks can bring serious security hazards, including a complete takeover of victims' accounts, identity impersonation, financial loss, etc.

We responsibly disclose all our findings and attacks to related vendors, communicate intensively with their developers, and provide mitigation suggestions to protect the security of XFVSes in their apps. We find that it is very challenging for developers to fully fix the security problems. Interestingly, although WeChat fixes the vulnerability after our disclosure, the fix is still vulnerable, and we have gone through 3 rounds of battles (i.e., another escalated attack after a fix) with WeChat to finally patch the vulnerability. For vulnerable apps from China, we also reported our findings to the Chinese National Vulnerability Database (CNVD)[1] [17] and receive 14 CNVD IDs, and one of them (CNVD-2021-86899) is awarded the most valuable vulnerability in 2021 by CNVD. Note that at the time of publication, all vulnerabilities mentioned in this paper have been fixed.

[1]CNVD is the counterpart of NVD in China and documents zero-day vulnerabilities of Chinese applications like the US NVD.

In summary, this paper makes the following contributions:

- We perform the first comprehensive measurement analysis on the security of cross-side face verification systems (XFVSes) from a system perspective, i.e., understanding the security of the cross-sided two-phase verification procedure.
- We design a principled method and implement a semi-automated framework named XFVSCHECKER to detect XFVS apps and analyze the security of XFVSes by inspecting how four security properties are satisfied with clear assessment rules.
- Our evaluation result reveals that real-world XFVSes, including those adopted by WeChat and AliPay, are under significant security threats. We summarize four typical attacks and illustrate them with case studies. We responsibly report the security issues and provide mitigation suggestions to related vendors.

Organization. §II states the research problem and threat model of this paper, and §III describes the measurement of XFVSes in real-world apps. In §IV we analyze the security of XFVSes, and the results are shown in §V. We discuss lessons learned and possible mitigation in §VI, and ethic and disclosure issues in §VII. We discuss related work in §VIII and conclude our work in §IX.

II. THREAT MODEL: XFVS

A face verification system (FVS) is designed to verify whether a person is who he/she *claims* to be, by comparing his/her submitted face data with previously enrolled reference face data. A similar concept is a face identification system (FIS), which aims to identify a person from a database of known faces. In other words, an FVS performs "one-to-one face matching", whereas an FIS does "one-to-many face matching" [18], [7].

To verify a person's identity, an FVS first collects his/her face data (face collection), and then verifies whether the collected face data is matched with the reference face data (face matching). In real-world applications, the two steps of face collection and face matching may occur in different places. Based on the split between face collection and face matching, we classify FVSes into three types: local FVS, remote FVS, and cross-side FVS, as shown in Fig. 2:

- Type I: Local FVS. A local FVS is designed to collect and match faces within the user's device. The face data are saved locally and never leave the device. To ensure its security, modern local FVSes, such as FaceID [1] on iOS and its counterparts on Android [19], [20], are often protected by a trusted execution environment (TEE), such as the *Secure Enclave* [1] on iOS devices, or TrustZone [21] on Android (Arm) devices.
- Type II: Remote FVS. A remote FVS is designed to collect and match faces at a location away from the user, such as those used at border gates [2]. As these FVS devices are maintained at a physical location, they

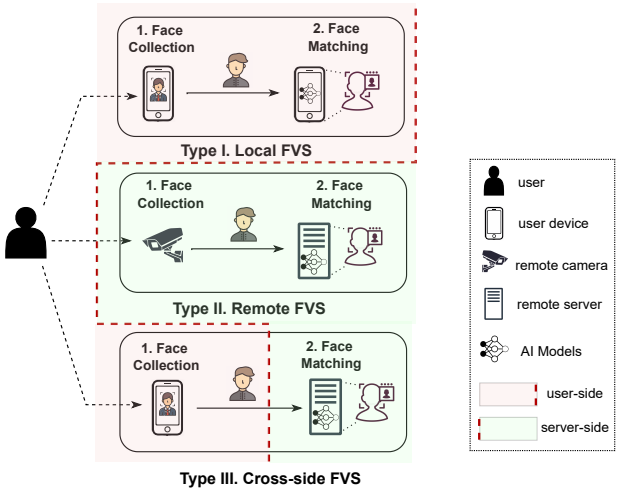


Fig. 2. FVS Classification according to Where Face Collection and Face Matching Take Place.

typically incorporate protections against physical and network adversaries.

- Type III: Cross-side FVS. A cross-side FVS (XFVS) collects faces on user-side devices, sends the face data to a trusted server, and performs face matching on the server side. Unlike local and remote FVSes which have to be done on specific devices, currently XFVSes usually allow users to use them from different devices, thus users can remotely verify their identities at any time and any place.

Our threat model of XFVS security involves two parties: (i) an untrusted client, and (ii) a trusted server. First, an XFVS client is untrusted because attackers can use XFVSes on a device under their control, such as a custom ROM, a rooted OS or even a phone with special hardware. More specifically, in-scope attacks involve but are not limited to code injection and data tampering, which modify client-side XFVS code and data. Second, an XFVS server is trusted because attackers do not have any control over the server-side code. At the same time, an XFVS server may be vulnerable: For example, a vulnerable server may forget to validate the untrusted data coming from the client.

III. XFVS Apps: A Measurement Study

In this section, we describe how XFVSChecker detects the use of XFVSes in real-world mobile apps and then measure the statistics of XFVS usage.

TABLE I
Downloaded Apps from Google Play and Xiaomi App Market.

App Market	# Apps	Privacy Policies	Categories
Google Play	12,503	11,860 (94.9%)	33
Xiaomi App Market	30,919	30,529 (98.7%)	16
Sum	43,422	42,389 (97.6%)	-

Dataset. We collect 43,422 apps from Google Play [22] and Xiaomi App Market [23] during May 2021, as shown in Table I. Specifically, we download top free and trending apps across all 33 categories from Google Play, and top apps from all 16 categories of the Xiaomi App Market. Note that all game-related apps are categorized as one "Game" category. We also download the privacy policies of these apps from their app introduction pages in the market.

A. Detecting XFVS Apps

The key idea of detecting XFVS apps is that the semantics of "face" and "verification" should be closely present in the apps, with clear boundaries from the rest of the code. Therefore, we use a lightweight static analysis method to locate such semantics from app code and privacy policies, as illustrated in Fig. 3.

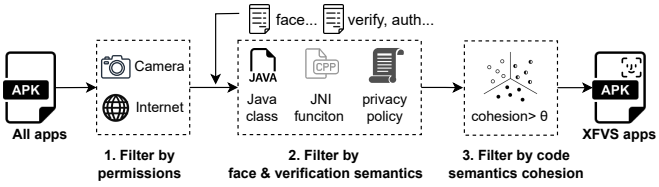


Fig. 3. Overview of How XFVSChecker Detects XFVS Apps.

First, XFVSChecker excludes apps that do not request *CAMERA* or *INTERNET* permissions in their manifest files, because XFVS apps need to use cameras to capture face data and send them to their servers via the network.

Second, XFVSChecker matches *face* or *verification* related keywords in app code and privacy policies. Given an app in APK format, XFVSChecker uses Androguard [24] and LIEF [25] to extract all fully qualified Java class names and JNI function names from its binary code. Then XFVSChecker searches the above code names and privacy policies, to find all occurrences of keywords shown in Table II, which are carefully selected to avoid false positives. For example, "detect" is not included because it is more related to locating faces in an image (face detection) rather than verifying users' identities. Note that natural language processing (NLP) techniques, including word segmentation, lemmatization and negative sentiment analysis are used before keyword matching. For example, deny words such as "Facebook" and "interface", and negative sentiments such as "we do not use your face information" are excluded.

TABLE II
Face and *Verification* Keywords for App Code and English Privacy Policy, While Their Chinese Equivalents Are Used for Chinese Privacy Policy.

Semantics	Keywords
Face	"face", "facial", "liveness"
Verification	"verify", "authenticate", "compare"

Third, XFVSChecker computes the semantics cohesion for each code package, which is the percentage of class or

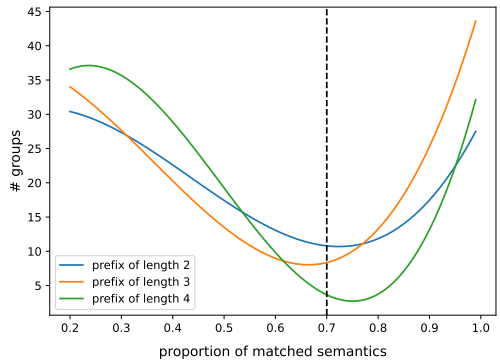


Fig. 4. Selecting the Threshold based on Semantics Cohesion.

function names with keywords related to face or verification. For example, if a package has three classes in total and two of them contain corresponding keywords, its cohesion value is 2/3. XFVSCHECKER keeps apps with high semantics cohesion, because the code of XFVSes usually tightly appears in a group of code, i.e. SDKs or libraries, rather than scattered all over the app. To do this, XFVSCHECKER calculates semantics cohesion for each code package and counts the number of code packages with same cohesion. The results, in different prefix length, are shown in Fig. 4, which shows most of the code packages, either have a semantics cohesion well below a value or well above a value. We then choose the threshold of 0.7, the average of the lowest points of the three lines in Fig. 4, to differentiate XFVS code from others (more details in Appendix B).

Manual confirmation. Following the above steps, XFVSCHECKER finds 2,273 real-world apps with rich face verification semantics. We then randomly select 100 apps to manually confirm whether or not they are XFVS apps. Two experts are asked to dynamically run these apps and locate XFVSes in them. Among all 100 apps, 89 apps are confirmed to be XFVS apps as we successfully triggered XFVSes in them. There are 6 apps that we cannot trigger XFVSes because we lack special privileges such as a specific bank card, but we find on the Internet that there are people using XFVSes in them. The rest five apps are false positives.

Further manual analysis of these false positives reveals that they do contain XFVSes. More specifically, they do integrate third-party XFVS SDKs, but do not use them. Instead, they use face-related techniques to provide photo beautification or emoji generation. Therefore, we filter apps with tags such as "beautify" and "emoji" in their app descriptions. After this step, we eliminated 60 apps from the original 2,273 apps, leaving 2,213 apps.

B. XFVS Measurement Results

We then measure the prevalence and distribution of XFVS apps, as well as top XFVS SDKs.

Prevalence. XFVSCHECKER finds 2,213 XFVS apps out of 43,422 real-world apps (5.1%) in two markets, indicating that XFVS apps are popular now. Note that XFVS apps are often top apps with large numbers of users, for example, these 2,213 XFVS apps have an average of 38M downloads. Another interesting finding is that Xiaomi App Market has more XFVS apps than Google Play (6.9% VS. 0.6%), indicating that XFVSes are more used in China.

TABLE III
XFVS Apps Prevalence.

Market	Google Play	Xiaomi App Market	Sum
# XFVS	78 (0.6%)	2,135 (6.9%)	2,213 (5.1%)

Distribution. We study the distribution of XFVS apps across different categories, and the results are shown in Fig. 5. Note that we merge similar categories across app markets and within each market for the convenience of presentation, and we get 13 categories with XFVS apps. We can see that *finance*, *social*, and *transportation* are the top categories, and apps in these categories often contain very sensitive data and operations. For example, we find that XFVSes are widely used in these apps to verify users' identities when they log in, query sensitive information and recall passwords, etc. Therefore, attacks on XFVSes can cause serious consequences to users.

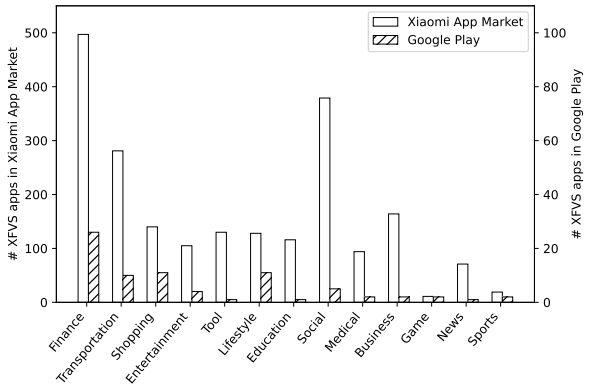


Fig. 5. Categories of XFVS Apps in Xiaomi App Market and Google Play. Note that Because There are Far More XFVS Apps in Xiaomi App Market than in Google Play, We Use Different y-axes.

Top XFVS SDKs. We cluster the XFVS code according to the prefixes of their package names and then identify XFVS SDKs. Table IV lists the top 10 SDKs in the Xiaomi App Market and the top 4 SDKs in Google Play, as well as the number of apps that have integrated these SDKs for face verification. These SDKs are from well-known software developers, including Internet giants such as Alibaba [26], Tencent [27], Baidu [28], and leading AI companies such as Sensetime [29], Megvii [30], FaceTec [31], etc, and they are widely used by a large number of users.

In all 2,213 XFVS apps, 2,108 apps (95.3%) use at least one of these top SDKs. We find that a large number of apps have integrated multiple SDKs, resulting in the top SDKs in the list being used 2,689 times. Specifically, there are 53 apps

TABLE IV
Top XFVS SDKs of Xiaomi App Market (Top 10) and Google Play (Top 4), and The Number of Apps that Use Them.

No.	SDK Name	# Apps	No.	SDK Name	# Apps
1	Alibaba [26]	475	8	PingAn [32]	191
2	Webank [27]	452	9	Linkface [33]	63
3	Baidu [28]	332	10	YITU [34]	50
4	SenseTime [29]	307	11	FaceTec [31]	25
5	Alipay [35]	277	12	Jumio [36]	23
6	CloudWalk [37]	241	13	Onfido [38]	20
7	Megvii [30]	223	14	Daon [39]	10

that use more than 3 SDKs. One reason is that an app may use different XFVS SDKs for different functions. For example, an app for handling government affairs in Zhejiang Province China, named *ZheLiBan*, uses *Aplipay* SDK for users to login with their faces, and *Sensetime* SDK for checking digital ID documents.

These statistics show that top SDKs make up a major proportion of the XFVS ecosystem, implying that their security is representative of XFVS apps. Therefore, the study below will mainly focus on these top SDKs.

IV. XFVS Security Analysis

In this section, we first summarize a generic workflow of XFVS (§IV-A) by studying top XFVS SDKs. Then we propose the key security properties that should be satisfied (§IV-B). After that, we discuss the methods to achieve these properties and our evaluation framework for testing the security of XFVS apps (§IV-C).

A. XFVS Workflow

For the top XFVS SDKs listed in Table IV, we thoroughly analyze their official documents, manually test their usage, and reverse-engineer their code, to figure out their general steps. We then abstract a generic workflow of XFVSes without loss of correctness, as shown in Fig. 6.

A typical XFVS includes four steps: initialization, face collection, liveness detection, and identity verification. The former two steps are preparation works; the latter two are the two phases used for face verification. We introduce each step below.

S1: Initialization. The main purpose of this step is to set up the configurations for XFVS client-side code and models. In this step, the XFVS client collects information about the local environment and sends them to the server (*1a & 1b*), along with the user ID to be verified, say user *A*. The server then considers various criteria, including the risk level of the client environment, the preferences of user *A*, and risk control settings (allowlists or denylists), to decide the XFVS configurations and send them back to the client (*1c & 1d*). For instance, if the server finds that the local environment is risky or *A* is on the denylist, it may require more complex liveness detection, request additional validations, or even reject the use of face verification.

A challenging task here is to correctly and completely collect the local environment information, such as whether the OS is rooted, whether the APK is repackaged, whether the process is being hooked, etc. The client may use tools such as Google's *SafetyNet* [40] or its successor *Play Integrity API* [41] to help them decide whether the environment is reliable. However, several previous works [42], [43] show these tools can be manipulated or bypassed by the attackers, as these tools still run in the user-controlled client without TEE-level protection.

After this step, the server knows which user to verify and the client model is set with corresponding configurations (*1e*).

S2: Face Collection. The purpose of this step is to collect user *A*'s face data by invoking the camera. Different XFVSes may collect face data of different forms, such as one or more pictures and videos. The collected face data is then used in subsequent steps (S3 & S4). Note that this step may take place several times according to the result of the liveness check (S3). In some SDKs, S2 and S3 may take place at the same time, i.e. real-time liveness check.

S3: Liveness Detection. The goal of a liveness check, or liveness detection, is to ensure that a real person presents in front of the camera, rather than a screen displaying images or videos, or a man with a mask. In theory, a liveness check should be performed on the server side as client-side results cannot be trusted. However, if all images and videos are sent to the server for liveness check, it can cause huge server-side resource consumption as well as privacy concerns. Therefore, most SDKs have a client-side liveness check, or called local liveness check, to serve as the first layer of filtering to block requests that are obviously not started by real people.

Local liveness check is performed by the client models, using configurations from S1 (*3a & 3b*). Different liveness detection methods, such as silent-based and action-based liveness detection, may be applied. In practice, client models may use watermarks and checksums to ensure the validity and integrity of the detection result (*3c*).

S4: Identity Verification. The goal of this step is to validate user *A*'s identity based on the provided face data, as well as client-side checking results, which are examined to ensure they are not tampered with (*4a & 4b*). Then the server-side liveness check is performed to further confirm that the one to be verified is indeed a live person (*4d*). After that, face matching is performed to check if the provided face can be matched with user *A*'s previously-enrolled reference face (*4e*). Whether the faces match or not determines the verification result, which is then sent back to the client (*4f & 4g*). After this step, a face verification request is completed.

According to our research on top SDKs, most SDKs follow the above workflow to conduct cross-side face verification. However, due to the absence of a unified standard, the specific implementation of each SDK varies. For instance, some SDKs overly rely on client-side liveness detection and omit the server-side liveness check, thus posing security risks. The

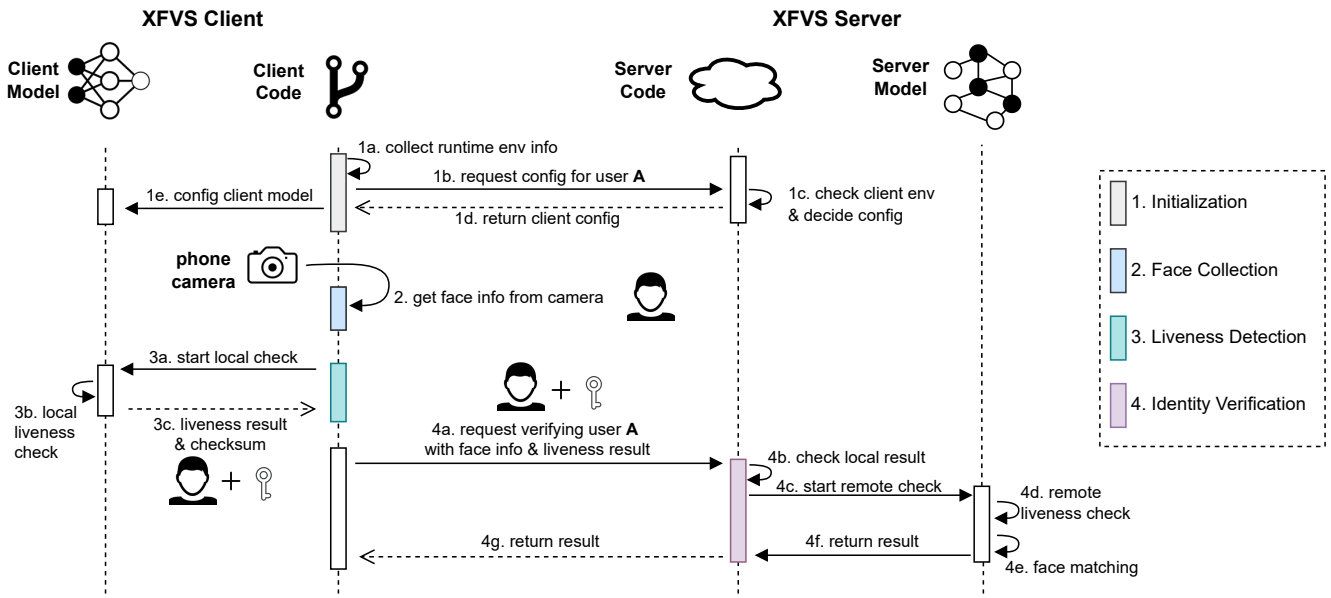


Fig. 6. A Generic Workflow of XFVS.

following subsection presents our principled analysis of this workflow.

B. XFVS Security Properties

As previously pointed out, the untrusted client environment makes XFVSes insecure by nature. However, in practice, a variety of protection methods are proposed by real-world apps to protect the integrity of XFVS code and data. This subsection describes the necessary security properties that these protection methods must meet.

TABLE V
Security Properties for XFVS Implementations.

No.	Security Properties	Concerned Steps
SP1	Reliable Environment	S1, S2, S3
SP2	Camera Security	S2
SP3	Reliable Liveness	S3, S4
SP4	Data Consistency	S1, S2, S3, S4

As shown in Table V, we propose four key security properties, by considering confidentiality and integrity in and across each step of the workflow, so that together they can guarantee the security of XFVS.

SP1: Reliable Environment. *Reliable Environment* property is to ensure that the code, data and machine learning models at XFVS client are not tampered with by local attackers. To meet this property, XFVS apps should be robust to the following threats: 1) Threats to OS-level reliability, including rooted OSes, emulators, and custom ROMs. For example, attackers might use a custom ROM with modified networking APIs, so that the communication data between the XFVS client and server can be controlled by the attackers. 2) Threats to static code integrity, including repackaged APKs and patched binaries. For example, attackers can control the logic of XFVS client-side code by statically instrumenting the SDK's native share libraries (.so files) if they are not well protected. 3) Threats to dynamic code integrity, including debugging and hooking. For example, attackers may use hooking tools such as Frida [44] and Xposed [45] to change the local liveness result. SP1 is the fundamental security property for client-side XFVS steps from S1 to S3.

SP2: Camera Security. *Camera Security* property is to ensure that the face data to verify should come directly from the physical camera lens, instead of an injected data stream. Attackers may manipulate the camera APIs and camera drivers (by violating SP1), or even physically modify the hardware, to inject prepared images or videos, such as a deepfake video [46], into the XFVS apps. As most current mobile devices do not support TEE-level or chip-level secure cameras, this security property is very hard to satisfy, making SP2 the shortest plank of the wooden barrel. Even so, XFVS apps should still use mitigation methods, e.g. by using low-level APIs or adding checks to raise the bar for attacks. SP2 is related to S2 in the workflow.

SP3: Reliable Liveness. *Reliable Liveness* property is to accurately determine whether the collected face data are being presented by a *live* person, as opposed to a fake or synthetic face. The main purpose of this security property is to combat the threat of presentation attacks. For example, attackers may use a printed picture [47], a screen showing the victim's images or videos [48], or even wear a carefully crafted 3D mask [49]. Note that this property aligns with previous works [50], [14], [51] on face recognition security. As previously discussed, SP3 should be met both at the client side and server side, i.e. S3 and S4 in the workflow.

SP4: Data Consistency. *Data Consistency* property is to ensure that XFVS data—including the collected face, the liveness configurations, and the validation results—are consistent between all steps across the client and the server. Most importantly, the server should validate the data consistency by itself, instead of blindly trusting results from the client side. For example, the app in Fig. 1 does not guarantee that the face data for liveness detection and for face matching are consistent. SP4 should be satisfied across all XFVS steps, from S1 to S4.

We summarize four security properties for apps to ensure the security of XFVSes. Note that these security properties are not independent of each other, and there may be overlaps between different properties. For example, in most cases, SP4 may not be satisfied unless SP1 is met. Nevertheless, as the wooden barrel effect suggests, any unmet SP may result in the failure of the XFVSes.

C. Checking Security Properties

Scoring Rules. XFVSCHECKER checks how an XFVS app satisfies each security property, from SP1 to SP4, along the XFVS workflow (Fig. 6). To ensure objectivity and accuracy in the evaluation process, we rely on automated methods as much as possible, leveraging pre-tailored sandboxes and testing scripts. For any areas where automation is not possible, we conduct manual testing by multiple security experts for result cross-validation. This approach enables us to automate testing for the majority of SP1 and SP2, while largely relying on manual efforts to evaluate SP3 and SP4.

We assign a score from 1 to 6 to each security property. If our security experts conclude that an XFVS app fully satisfies a security property, for example, all data are ensured to be consistent, it will obtain a score of 6 (★★★★) on SP4. If only the preliminary requirements are met, such as using silent liveness detection on SP3, a minimum score of 1 (☆) is assigned. In other cases, an intermediate score is given based on how well the XFVS app meets the security property. The complete scoring rules are listed in Table IX in Appendix. We discuss each step in detail below.

1) Checking Environment (SP1): The first step is to check whether XFVS apps adopt anti-reversing defenses to be *resilient* to unreliable environments. The Open Source Foundation for Application Security (OWASP) [52] has a project named Mobile Application Security Testing Guide (MASTG) [53], which recommends a common testing procedure on the *resiliency* of mobile apps and is widely used by existing works [54], [55]. It mainly contains 13 resiliency metrics in its latest version (v1.5.0), where 6 of them, i.e. *MSTG-RESILIENCE-1* to *MSTG-RESILIENCE-6*, are related to environment checking of XFVS, as shown in Table VI.

To test these resiliency items, XFVSCHECKER uses a combination of automated sandbox testing and manual analysis. First, XFVSCHECKER builds several sandboxes according to the detailed items in MASTG. These sandboxes include rooted systems and custom ROMs (MR1), debugging environment (MR2), repackaged apps (MR3), emulators (MR5), and hooked systems (MR4 & MR6). XFVSCHECKER manages the sandbox settings to make sure that one sandbox only reflects one resiliency item. For example, to build a sandbox that only has hook abilities (MR4 & MR6) but does not have root features (MR1), XFVSCHECKER embeds hooking code such as Frida agent or Xposed modules into the target app and utilize root-hiding tools [63], [64] to remove root features.

TABLE VI
Resiliency Metrics Tested by XFVSChecker. Note That 1) *MR1* is Short for *MSTG-RESILIENCE-1* and So Are the Others; 2) We Merge MR4 and MR6 Because They Are All Related to Hooking Techniques in Most Cases.

Resiliency Item	MSTG-ID	Example Methods
Root & ROM Detection	MR1	root feature detection [56], custom ROM detection, SafetyNet [40]
Anti-Debugging	MR2	debugger prevention [57], debugger port/flag/API detection
File Integrity	MR3	APK signature [58], *.so* file integrity[59]
Anti-Hooking	MR4&6	memory integrity detection, hook tool detection [60], hooking prevention [61]
Emulator Detection	MR5	emulator file/property detection [62]

Then XFVSCHECKER dynamically invokes XFVSes of the target app in these sandboxes. If users can finish face verification in any one of these sandboxes, it indicates that the target app lacks resilience to the corresponding metric. If the target app detects all the sandboxes, we then introduce manual efforts to further test its resiliency against risky environments. Specifically, two security experts with extensive experience in mobile app security and reverse engineering manually evaluate the target app. §A in Appendix gives some manual evaluated examples.

The scoring of SP1 is straightforward, that is, the score an app gets is directly proportional to the number of resiliency items it can satisfy.

2) Checking Camera Security (SP2): We summarize possible methods that XFVS apps can use to protect camera data, listed in order of their protection strength from weak to strong.

- Using framework level camera APIs, such as *onPreviewFrame* and *takePicture*.
- Using native camera APIs [65] instead of framework APIs. However, low-level APIs may not be compatible with old devices and they can also be attacked.
- Using the above APIs with protection methods, such as validating frames from both front and back cameras.
- Recording the screen to validate data from the camera. Data injected into the camera APIs may display exceptions on the screen, such as a mismatch between resolution and screen size, and recording the screen can detect such exceptions.
- Using hardware-level protection such as a secure cam-

era [66]. However, this capability is not widely supported in current mobile devices.

XFVSCHECKER instruments all the aforementioned camera APIs on a custom ROM, dynamically triggers the face verification process, and automatically finds out which camera APIs are invoked. If this method fails due to ROM detection or other reasons, we manually inspect their code to determine how the XFVS apps get the camera data. The scores for SP2 are given based on how they use the above methods to ensure camera security.

3) Evaluating Liveness (SP3): XFVSCHECKER then checks how XFVS apps ensure reliable liveness detection. After studying the documentation of XFVS SDKs, reverse-engineering real-world apps, and referring to existing studies [11], we classify existing liveness detection into the following security levels:

- *Silent liveness detection*, where no actions are required and the texture and depth information of human faces are captured for detection.
- *Action-based liveness detection*, where some fixed patterns of actions are required, such as closing eyes, shaking heads and opening mouths. This level also includes reflection-based liveness, where the screen emits different colors of light and the reflection of the human faces is captured for detection.
- *Randomized interactive liveness detection*, where users are randomly instructed to perform certain actions in a given time, such as reading random texts or numbers.

In this step, we manually test the liveness detection of the target app and can directly observe which type the client side uses. The server-side liveness check is invisible to us and we evaluate it using black-box testing, as done in prior work [11]. More specifically, we manually reverse-engineer the XFVS apps to locate APIs that send face data to the server, and then we change the face data that are inputted into these APIs. By uploading videos of different levels and observing the results returned by the server whether they pass or not, we can infer the type of server-side liveness detection. For example, we can upload a still video to see if the server uses silent or action-based liveness detection. Note that the still video is synthesized by repeating one image of the victim, changing one bit per frame to avoid having the same checksum value and being detected by the server. We give a score to each XFVS app based on the strength of the client-side and server-side liveness detection.

4) Checking Data Consistency (SP4): Finally, XFVSCHECKER checks whether the data are consistent throughout all steps and between the client and server. Specifically, the following data are considered: 1) *Face data*, including the images and videos of the person to be verified; 2) *Liveness configurations*, sent by the server to the client to configure which type of liveness detection should be used; 3) *Validation results*, sent from the client to the server, including whether local environment is reliable and whether the local liveness detection is passed.

XFVS apps may adopt multiple methods to ensure data consistency. For instance, XFVS apps can compute checksums for each validation result using MD5, or more secure hashing algorithm such as HMAC, or using PKI-based signature. They can also apply watermarks to each face image and video after the local liveness detection to prevent data tampering. Most importantly, rather than relying on the client's results, the XFVS server should carefully check the data consistency independently. XFVSCHECKER gives a score based on how well data consistency is guaranteed (more details in Table IX).

V. XFVS EVALUATION RESULTS

In this section, we present the security analysis results on real-world XFVS apps. First we describe the overall results on representative apps of top SDKs. Then we propose four typical attack types with real-world case studies. After that, we summarize the possible consequences of these attacks and demonstrate the difficulties to prevent these attacks.

A. Overall Result

We apply our testing framework to evaluate the security of top SDKs listed in Table IV. Since an SDK cannot be used alone, we choose representative apps for each SDK. We fail to find representative apps for two SDKs, namely "Linkface" and "Daon" in Table IV, because their apps require additional permissions that we do not have, such as membership of certain companies. We analyzed over 80 apps in total with at least three apps per SDK, and all the analyzed apps are vulnerable to at least one attack described below. Finally, we select 12 representative apps for 12 different top SDKs. These representative apps are all well-known and popular ones, and some of them even have been downloaded billions of times.

The evaluation result is shown in Table VII, which reveals that security properties are not well satisfied in real-world XFVS apps, even top ones. We come up with the following observations:

- It is very difficult for XFVS apps to guarantee the reliability of their running environment (SP1). Our security experts can bypass their environment detection methods in various ways. For example, a well-crafted custom ROM with modified camera APIs and networking APIs can evade the detection of all the tested apps.
- Most of the tested apps simply use framework-level camera APIs such as *onPreviewFrame* and *takePicture*, which can be easily hijacked under the circumstance that SP1 is not satisfied.
- For SP3, many apps only use actions of fixed patterns, even silent liveness detection, allowing attackers to synthesize a compliant video from a single image of the victim.
- In terms of SP4, about half of the apps have no mechanisms to ensure data consistency, such as validating checksums for the uploaded face data on the server side.

Note that these scores only reflect the corresponding versions of these apps at the time we downloaded the dataset

TABLE VII
The Overall Evaluation Results on Top XFVS SDKs and Their Representative Apps (at the Time of May 2021). Note that "AT2&3&4" Means the Attack is a Combination of AT2, AT3 and AT4, While "AT2,3,4" Means Any One of AT2, AT3, AT4 Can Be Launched.

No.	App Name	Package Name	Version	Downloads	SDK	Security Protection				Attacks	Reported	Fixed
						SP1	SP2	SP3	SP4			
1	Taobao	com.taobao.taobao	10.0.0	4.8B	Alibaba	★	☆	★	★★☆	AT3	✔	✔
2	WeChat	com.tencent.mm	8.0.6	4.3B	Webank	★	☆	★★☆	★★☆	AT2&3&4	✔	✔
3	Baidu	com.baidu.searchbox	12.23.5.10	3.6B	Baidu	★☆	☆	★☆	☆	AT2,3	✔	✔
4	Alipay	com.eg.android.AlipayGphone	10.2.13.9020	2.6B	ZOLOZ	★	☆	★★	★☆	AT2,3,4	✔	✔
5	ZheLiBan	com.hanweb.android.zhejiang.activity	6.17.0	5.9M	SenseTime	★☆	☆	☆	☆	AT2,3	✔	✔
6	Antifraud	com.hicorenational.antifraud	1.1.20	43M	CloudWalk	★	☆	★☆	★	AT2,3,4	✔	✔
7	China Unicom	com.sinovatech.unicom.ui	9.0.1	489M	Megvii	★	☆	★☆	★	AT2,3,4	✔	✔
8	Pingan Jinguanjia	com.pingan.lifeinsurance	8.02.00	269M	PingAn	★★	☆	★★	★☆	AT3	✔	✔
9	PuHuiDaoJia	com.baoli.blzj	6.3.7	8.4M	YITU	★★	☆	★	★★	AT2,3	✔	✔
10	ZenGo	com.zengo.wallet	3.2.2	100K	FaceTec	★	★★	★☆	★★	AT3	✔	✔
11	Jumio Showcase	com.jumio.demo.netverify	4.0.0	10K	Jumio	★☆	★★	★☆	★★	AT3	✔	✔
12	USAA Mobile	com.usaa.mobile.android.usaa	10.8.1	5M	Daon	★	☆	★	☆	AT1,2,3	✔	✔

(May 2021), while XFVS security has been significantly enhanced in the recent versions of these apps, as discussed in §VI-B.

B. Typical Attacks & Case Studies

As the security properties are not well met, XFVS apps face a variety of attacks. In this subsection, we study four typical types of attacks, through which attackers can easily cheat or bypass face verification in XFVS apps. The apps in Table VII were vulnerable to at least one of the four attacks, and we have included a demo video for each app here[2].

Note that in contrast to previous attacks [4], [3], [10], [11] against face-related AI models (*AI perspective*), the *root* cause of these attacks is the failure of the studied SDKs to meet security properties from the system perspective. More importantly, since the studied SDKs are vulnerable, any XFVS apps relying on the SDKs are vulnerable. Fig. 7 depicts how the four types of attacks work and the steps targeted in the XFVS workflow. We discuss the details of each attack with a real-world case study below.

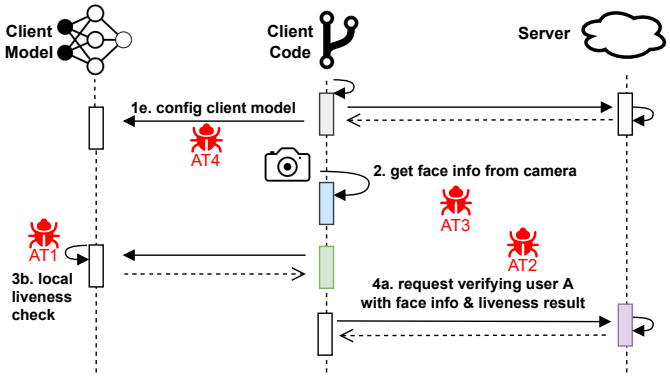


Fig. 7. Four Typical Attacks on XFVS Apps.

AT1: Liveness Bypassing Attack. This attack targets step 3b in the workflow, and apps that violate SP1, SP3 and SP4 are vulnerable to this attack. The root cause is that the XFVS app lacks reliable environment checks to protect the integrity of client code (SP1), allowing attackers to modify client logic by hooking or repackaging, to completely skip local liveness detection. After that, if there are no data consistency requirements on the face data between local liveness detection and data uploading (SP4), then the attacker can replace the uploaded data with the face data of the victim, which may be obtained from social media. Finally, if the face data can pass the server-side liveness check (SP3), the attacker will be verified as the victim. In most cases, attackers only need the victim's account ID, such as phone number, email address, or social ID number, and one photo of the victim to launch such an attack.

Case1: USAA. The USAA mobile app is developed by the United Services Automobile Association to provide services on finance and insurance for its members, which has been downloaded more than 5M times on Google Play. To secure users' accounts, it asks users to scan their faces each time the app starts. It adopts action-based liveness detection, i.e., asking users to blink their eyes, on the client side but no liveness detection on the server side. However, because it lacks basic environment detection, the integrity of its client code cannot be guaranteed. An attacker can hook the client code and skip local liveness detection, and then send one image of the victim to the server to be verified as the victim.

AT2: AB Attack. This attack targets step 4a in the workflow, and apps that violate SP1, SP3 and SP4 are vulnerable to this attack. Compared to AT1, the XFVS apps may adopt strong protections on local liveness, so that it is difficult for an attacker to directly skip the local liveness check. However, the attacker (say A) can stand in front of the camera to finish the local liveness detection by him/herself, then upload the face data of the victim (say B), and the identity of B will be verified by the server. In this attack scenario, the apps lack adequate data consistency check (SP4), resulting in the face data being replaced by the attacker when they are uploaded to the server. This can be done by injecting manipulating code to the client (SP1), or modifying communication data with a man-in-the-middle (MITM) attack (SP4). Again, if the XFVS

[2] https://www.youtube.com/playlist?list=PLecyq8GEUCfmBX2xc-xml1h7LRU04lQVP

server adopts no liveness detection or weak liveness detection (SP3), this attack can be launched by obtaining one face image of the victim. 8 out of 12 apps in Table VII are vulnerable to this attack.

Case2: Pingan Jinguanjia. Pingan Jinguanjia is one of the most popular insurance apps in China, which has been downloaded more than 269M times in XiaoMi App Market. It requires users to verify their identities using an XFVS before they can buy insurance products. The local liveness detection in this app is highly effective, which requires users to do actions according to random instructions. However, its server-side liveness detection is weak, only requiring one image of the user's face for silent liveness detection. Furthermore, the face data uploaded to the server are not well protected to ensure they are consistent with the client. Therefore, an attacker can launch the AB attack on this app and use one face image to impersonate the victim.

AT3: Camera Hijacking Attack. This attack targets step 2 in the workflow, and apps that violate SP2, SP3, and optional SP1 are vulnerable to this attack. As currently most mobile devices do not use hardware-level secure cameras and the client side of XFVS does not run in a trusted environment, the data from the camera can be modified by the attacker. An attacker can hijack the camera APIs by hooking the process or using a custom ROM, or even modify the camera hardware to inject a prepared video of the victim. In this case, the prepared video must pass liveness detection both on the client side and the server side, and also the face verification on the server side, thus this attack is essentially a face spoofing attack [11] on both client-side and server-side AI models. This proposes a higher demand on the attacker's capability to generate high-quality videos, thus video synthesis techniques, especially deepfake methods [67], [68], [11] are used here. In theory, all apps in Table VII are vulnerable to this attack, if adequate videos can be generated.

Case3: ZenGo. ZenGo is a popular cryptocurrency wallet app, which has more than 100K downloads on Google Play and uses an XFVS to verify users' identities when they log in. ZenGo has deployed various methods to protect the security of its data, for instance, it encrypts and checks the data uploaded to the server so that AT1 and AT2 are difficult to be launched. However, the usage of camera APIs is not well protected, and an attacker can hijack the camera to inject a video of the victim. More specifically, as its server asks the user to take pictures from different distances, the attacker can mimic this change in distance by zooming in and out of the victim's still image to synthesize a video.

AT4: Downgrade Attack. This attack targets step 1e in the workflow, and apps that violate SP4, SP3, and optional SP1 are vulnerable to this attack. The root cause of this attack is that the liveness configurations are not consistent between the client and the server (SP4), and an attacker can modify the configurations by hooking the client-side code (SP1) or MITM attack (SP4). As a result, the liveness detection strength on the client side can be downgraded, for example, by changing from randomized interactive liveness detection to silent liveness detection. This attack must be combined with the former attacks to finally work. 4 apps in Table VII are vulnerable to this attack.

Case4: Alipay. Alipay is the most popular payment app in China, being downloaded more than 4 billion times in XiaoMi App Market. It is widely used by people for shopping, investment, business remittance, etc, and it allows users to log in with their faces using an XFVS. Alipay adopts randomized interactive liveness detection, computes checksums of face images in encrypted native code, and checks them on the server side, posing great difficulty for attackers to launch previous attacks. However, there's an optional silent mode at the initialization of liveness detection, which would never be triggered in the normal authentication process. The problem here is that the XFVS server does not ensure that this mode cannot be used on the client side. Therefore, attackers can modify this configuration to downgrade to silent liveness detection, and then AT2 & AT3 can be launched.

We can see that the threshold for attackers to launch these attacks is relatively low. Except for AT3 that may require more than one photo or even a video, attackers only need one photo of the victim and his/her account ID to launch AT1, AT2 and AT4 in most cases.

C. Attack Consequences

As the above case studies have shown, XFVSes are often used for user authentication and authorization in mobile apps, as they provide the ability to remotely validate the user's claimed identity. Therefore, if XFVSes are compromised, potentially catastrophic consequences can occur. We study top XFVS SDKs and apps to summarize the consequences of these attacks.

Direct Impacts to Vulnerable Apps. Depending on how mobile apps use XFVSes, attacks on them may bring various types of security consequences, as listed below:

- *Complete Account Takeover*. Some apps may allow users to use XFVSes to directly log in to their accounts, or recall their forgotten login passwords. In such cases, attackers can use the vulnerabilities of XFVSes and one or more images of the users, to fully control their accounts. A typical example is "Taobao" (the 1st one in Table VII), one of the most popular shopping apps in China, which allows users to turn on the option of face login. If users do so, they can log in to their accounts by their faces on any device without additional verification. Therefore, attackers can attack the XFVSes to gain full control of the victims' accounts, viewing sensitive information such as shopping history and delivery address, and even placing orders if the victims enabled password-free payment.
- *Identity Fraud*. Some apps provide services according to the users' real identities. For instance, loaning apps must issue loans to the exact individual. In such cases, attackers can first register with the victim's name, then attack the XFVSes to be verified as the victim, and finally, they can take out loans in the victim's identity. This also applies

to some driving apps, training apps, and insurance apps in our dataset.

- *Financial Loss*. There are many apps with payment functions that require users to set a "payment password". If this password can be modified by cross-side face verification, then it may be attacked by the above attacks. For example, attackers can first use the victim's face data to log in to the account, and then attack the XFVS again to change the victim's payment password. In such cases, for example *Case4: Alipay*, the attackers can take all the victim's money.

Indirect Impacts on the Ecosystem. Due to the use of OAuth and sub-apps, these attacks may also have a wider range of indirect effects in addition to direct impacts on the vulnerable apps. As an illustration, several apps in Table VII, including Taobao, Baidu, and Alipay, offer OAuth login for other apps. These apps are also "super-apps", allowing other "sub-apps" to run on top of them. According to a prior study [69], WeChat has more than 3.8 million sub-apps, whereas Alipay has more than 2 million. As a result, if they are compromised, attackers may use them to attack other apps and sub-apps, posing further risks.

D. Patch Challenges

We use a real-world case to show how challenging for developers to directly patch/fix the security problems of XFVSes.

WeChat is one of the most popular apps in China, which covers many aspects of people's lives, including social, payment, finance, entertainment, and so on, thus carrying a large number of highly sensitive user data. Furthermore, it also serves as a super-app [70], where millions of sub-apps [69] provide different services to users. WeChat uses an XFVS to verify users' identities, which is invoked when users do sensitive operations such as binding a bank card in WeChat, or when a sub-app asks WeChat to verify users' identities.

Originally, the XFVS in WeChat uses a combination of action-based liveness detection and reflection-based liveness on the client side, but the server-side liveness detection is weak so we can launch *AT2: AB attack* on WeChat. Note that WeChat calculates a checksum for each image with a fixed hashing function, thus we can actively invoke this function by *AT3: Camera Hijacking Attack* to get the checksums for our replaced images.

After we report this vulnerability to CNVD, the WeChat team strengthened the verification by ensuring the consistency of uploaded data. Specifically, the client side will upload a video when doing a client liveness check and the server will make sure that both the video and uploaded images have the same face.

However, we find that the video is used only for data consistency check but not for server-side liveness detection, so the second round of attack is straightforward. We use one image of the victim to generate a video and upload it to the server, and pass the server-side protection.

After our second round of attack, the WeChat team then deployed a strong liveness check on the server side, i.e. using randomized reflection-based liveness detection. This defense can prevent our attack, but it introduces extremely high server-side costs because AI models have to be invoked for liveness detection, especially given the fact that WeChat verification APIs are heavily used by many users in WeChat itself and lots of sub-apps.

In such case, we find that *AT4: Downgrade Attack* can be used to modify the configures for AI models. Specifically, we find that the randomness of reflection-based liveness can be reduced, and we can specify a fixed pattern of reflection colors. Therefore, we pre-processed the face images of the victim with a fixed sequence of different colors using graphic processing techniques, and then send them to the server and pass the verification.

After receiving this attack, the WeChat team further strengthened the server-side liveness detection, and also adopted other risk assessment methods to avoid the reduction of randomness and abnormal invocation of the verification systems. At this point, we find that it is very hard for attackers to launch the above attacks. However, as WeChat still runs on the untrusted mobile device, in theory, the XFVS in WeChat can be attacked, for example, using hardware-level camera hijacking with well-crafted videos of the victim.

VI. Lessons Learned & Mitigation

We discuss the lessons learned from the XFVS security analysis and possible mitigation in this section.

A. Lessons Learned

We have learned the following lessons from the above security analysis of XFVS apps:

- *Security of AI Systems vs. AI Models*. When AI techniques are deployed in real-world applications, its security is a systematic issue that includes not only the security of machine learning models, but also the security of the software, operating system, and hardware that make up the whole AI system. In the case of FVS security, attention should be paid to the design of the whole system, in addition to the security of AI models.
- *Wooden Barrel Theory*. The security of a system composed of multiple modules requires consideration of confidentiality, integrity, and availability within and between each module, while the security level of the entire system depends on the weakest link. For XFVSes, while the attacking bar for AI models is becoming higher, the system design and implementation need better protection.
- *Consideration of Worst-case Assumptions*. As our analysis and attacks have shown, attackers may use a variety of methods to attack an XFVS. Therefore, developers need to make the worst-case assumptions and ensure that all security properties are well met in XFVSes.

B. Mitigation

Based on our security analysis and the lessons learned, possible mitigation to the above XFVS attacks is discussed below.

1. Satisfying all security properties. The fundamental solution is to ensure that all security properties (SP1 to SP4) are well satisfied. First, processes of XFVSes and the cameras should be placed in a trusted environment, such as using secure cameras [66] to capture face data, using TEE to process the data on the client side, and sending them to the server within secured network channels. Furthermore, the server side should apply strong liveness detection, even manual verification, to prevent presentation attacks, where attackers show a video of victims on a screen or wearing 3D masks. However, providing a fully trusted environment across the client side and server side will introduce significant costs, and currently, most mobile devices do not support such a trusted environment. Besides, reliable liveness detection faces continuous challenges of AI perspective attacks [11].

2. Using PKI with Local FVSes. A potential alternative to XFVSes is using PKI-based Local FVS, such as the FIDO2 protocol [71]. FIDO2 relies on the underlying mobile device to provide secure local FVS [1], [20], [19]. It generates a pair of asymmetric keys and sends the public key to the app server when binding the user's account. The private key is saved in the TEE of the local device and can be used to encrypt data each time a successful local face verification is finished. In this way, it can allow users to log in to the app using their faces. However, this method has the following shortcomings: 1) It relies on the security of local FVS, which also faces the threat of presentation attacks. 2) The app server cannot get the face data of the users, which may not meet the app's requirements, for example, some bank apps need to view the faces of users. 3) The local FVS is bound with the devices where users bind their accounts, which may affect the flexibility of usage. 4) FIDO2 requires hardware and system support, and can only run on high versions of Android [72]. Nevertheless, using FIDO2 can achieve the same function for the app servers in most cases and is recommended by this paper.

3. Enhancing XFVSes. Although it is hard to fully satisfy all security properties, some methods can be adopted to enhance current XFVSes. The XFVS apps can use more robust environment-checking tools, increase the randomness of liveness detection patterns, and add more checks to ensure data consistency. According to our interactions with the developers, techniques listed in §IV-C can significantly raise the bar for attackers.

4. Protecting XFVSes. Another idea is to protect XFVSes by filtering out abnormal requests. To do so, the app server can use device fingerprinting methods to only allow users to use XFVS on a few trusted devices, and also it can check the IP address of the request and the behavior history of the users. For example, if a user tries to use an XFVS in two distinct physical locations within a very short period of time, then these requests may be problematic.

5. Using multi-factor authentication (MFA). The next possible mitigation is to use MFA in sensitive operations instead of using only an XFVS as the single factor. For example, if users want to change their payment passwords, they should provide more proof, such as an SMS code, etc.

After we report the vulnerabilities, most apps choose a combination of methods 3, 4, and 5 to secure their XFVSes. Methods 1 and 2 are seldom applied because they are restricted to certain usage scenarios discussed above.

VII. Discussion

Responsibly Disclosure. Our research began in early 2021, and the first vulnerability was discovered in May. We have actively contacted the SDK and app vendors since then to fix the vulnerability. In order to improve the efficiency of repair, we also reported the vulnerabilities of apps in China to CNVD, who coordinated the vendors for emergency fixes. However, as shown by this paper (§V-D), it is very challenging for developers to fully fix this vulnerability. Therefore, it takes a very long time before these vulnerabilities can be fixed. We have been actively involved in this process, providing fix suggestions and performing retests. As a result, we acquire 14 CNVD IDs (detail listed in Table VIII in Appendix), and one of them (CNVD-2021-86899) is awarded by CNVD the most valuable vulnerability in 2021.

At the time of writing this paper, the vulnerabilities in the apps mentioned in this paper have been well fixed, by methods discussed in §VI-B. The disclosure of this paper will not result in these apps being attacked due to the details discussed in this manuscript or in our demo videos.

Ethics. We take the following measures to make sure that our research does not bring any ethical issues:

- All of our attacks are tested on our own devices with our test accounts, which do not harm any other users or accounts.
- Our tests do not put traffic and computational pressure, or bring any negative effects to the app servers.
- We have been in close contact with app vendors regarding vulnerability details and mitigation suggestions. Alibaba, Baidu, and Facetec have therefore invited us for retesting with licenses.
- All of our researchers are fully aware of the potential risks and they do not disclose any details of the vulnerabilities to any other parties before the vulnerabilities are fixed.

XFVSes on iOS. This paper uses the Android apps as a demonstration, while the same security problem also affects other platforms such as iOS. First, XFVSes in iOS devices also run in untrusted environments, which can also be tampered [53]. Second, although the TrueDepth cameras on iOS devices can capture more accurate face data, it is not used in XFVSes. And most importantly, the attacks on XFVSes do not require access to the victim's device. In other words, attackers can launch the attacks on their own devices, e.g. a customized Android device.

Local FVSes and Remote FVSes. In this paper, we focus on the security of XFVSes, but the same idea may also apply to local FVSes and remote FVSes, if the proposed security properties are violated. For instance, if data consistency is not strictly guaranteed, attackers can also launch *AB Attack* to remote FVSes such as those deployed at border gates [2], e.g. by replacing the face image to be matched through MITM attacks between the border gate cameras and the servers.

VIII. Related Work

The security of face recognition (including face verification, face identification, face detection, etc.) systems has received much research attention, especially from the AI perspective.

Presentation Attacks. Presentation attacks, in which victims' faces are presented in different mediums (e.g. A4 papers, screens, 3D models), can be used to spoof the face verification systems. Y Li et al. [6] notice that public photos on social media can be used in presentation attacks, and present these social media photos on the LCD screen to bypass the face verification system. While Y Xu et al. [3] build 3D models of victims based on social media photos and conduct a VR-based spoofing attack. 3D masks are also used by N Erdogmus et al. [4], and they find that fraud detection capabilities of face verification systems mainly shape around 2D attacks. Kose et al. [5] present a study of 3D-mask attacks on two different FR methods, one using 3D data and the other using local binary patterns-based face-image comparison, and they found that both are vulnerable to mask attacks.

Adversarial Attacks. Some work [7], [8], [9], [10], [73], [74] conduct adversarial attacks on face identification systems. Y Dong et al. [73] evaluate the robustness of state-of-the-art face recognition models and propose a decision-based black-box attack that applies minimum perturbation to an input face image to cheat the models with fewer queries. M Sharif et al. [9] put adversarial patches on glasses to avoid face detection or disguise as other people. S Shan et al. [10] apply pixel-level patches to photos uploaded on social media, to avoid users from being recognized by unauthorized face identification systems. E Wenger et al. [7] investigate the current situation of anti-facial recognition tools. Z Deng et al. [75] conduct systematic research on adversarial sample attacks of DNN models in real-world apps.

Deepfake Attacks. Deepfake technology has also been used by several works [76], [11], [77], [67], [68] to attack face recognition systems. Unlike adversarial attacks, deepfake aims to generate faces that are difficult to identify with human eyes. P Korshunov et al. [76] show that the state-of-the-art face recognition systems based on VGG and Facenet neural networks are vulnerable to deepfake videos. O Wiles et al. [67], E Zakharov et al. [68] generate synthetic faces based on the pose and expression of the driving image. S Tripathy et al. [77] provide an interpretable and controllable face reenactment network to control the pose and expression of a given face image with human-interpretable signals (e.g., head pose angle). In summary, with only one target face image and a deepfake model, attackers can generate target faces with arbitrary poses and expressions, thus evading action-based liveness detection mechanism.

The work that is closest to our study is LiveBugger [11], which tests the ability of deepfake models to bypass existing server-side liveness detection models. LiveBugger finds that most XFVS servers do not use anti-deepfake detection and are therefore vulnerable to deefake attacks. However, LiveBugger only focuses on the server-side AI models from an ML perspective, while our work study the security of all components in XFVSes from a system perspective. More specifically, the attacks discussed in LiveBugger can be seen as a special form of *AT3: Camera Hijacking Attack*, where the attackers do not consider local checks, but only need to bypass server-side liveness detection.

Counter Measures for AI models. Many researches [51], [13], [15], [12], [50], [14], [78], [79] have been conducted to defend against the attacks on AI models. J Määttä et al. [50], [51] present texture analysis methods to detect face anti-spoofing. D Wen et al. [14] provide a face spoof detection algorithm based on image distortion analysis. Y Li et al. [13] use the accelerometer and gyroscope with the camera for liveness verification to prevent fake video attacks. D Tang et al. [15] believe that motion liveness gives the attacker too much reaction time, so they propose to use flash to eliminate this gap. Z Ming et al. [12] point out that there is a gap between face recognition and liveness detection that may lead to an attack window, and propose a scheme for doing both at the same time. These works can help improve the robustness of liveness detection, helping XFVS apps to satisfy *SP3: Reliable Liveness*, therefore they may be used on the server side of XFVSes to enhance their security.

Mobile App Security. As a countermeasure to risky environments, app integrity detection has been a heated topic of security works [80], [81], [82], [83], [84], [85], [86], [87]. T Vidas et al. [81] present techniques for detecting Android runtime analysis systems, demonstrating that malware may evade analysis in the same way. M Backes et al. [82] focus on Android libraries and quantify the security impact of third-party libraries on the Android ecosystem. K Lim et al. [84] propose a detection scheme against dynamic reverse engineering attacks. A Merlo et al. [83] look into the repackaging and anti-repackaging techniques, and summarize attack vectors and state-of-the-art anti-repackaging schemes.

Research [54] is relevant to our work. They utilize MASTG to evaluate the resiliency of 455 popular financial apps, and find that they lack sufficient protection. This confirms our observation that it is extremely challenging for mobile apps to meet *SP1. Reliable Environment*. Also, S Sun et al. [81] conduct experiments on the validity of root detection in 182 apps and conclude that reliable methods for detecting rooting must come from integrity-protected kernels. This underscores the need for trusted hardware to provide secure XFVSes.

IX. Conclusion

In this paper, we present the first comprehensive analysis on the security of cross-side face verification systems (XFVSes) from the system perspective. We propose XFVSChecker, a semi-automated framework for analyzing the security of XFVSes by inspecting four security properties. We collect 43,422 Android apps from Google Play and Xiaomi App Market, locate XFVS apps in them, and evaluate their security. The result reveals that real-world XFVSes, including those adopted by top apps, are under significant security threats. We propose four typical attacks against top XFVS SDKs and apps, and responsibly report the security issues and mitigation suggestions to related vendors, rewarded with 14 CNVD IDs. We hope our study can shed light on this security issue and promote the security of XFVSes.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (62102091, 62172104, 62172105, 61972099, 62102093). Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700 and the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). The authors from Johns Hopkins University were supported in part by National Science Foundation (NSF) grants CNS-21-54404 and CNS-20-46361 and a DARPA Young Faculty Award (YFA) under Grant Agreement D22AP00137-00. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring.

References

[1] Apple, "About face id advanced technology," 2022, https://support.apple.com/en-us/HT208108.

[2] C. News, "Facial recognition technology is coming to canadian airports this spring," 2017, https://www.cbc.ca/news/science/cbsa-canada-airports-facial-recognition-kiosk-biometrics-1.4007344.

[3] Y. Xu, T. Price, J.-M. Frahm, and F. Monrose, "Virtual u: Defeating face liveness detection by building virtual models from your public photos," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 497–512.

[4] N. Erdogmus and S. Marcel, "Spoofing 2d face recognition systems with 3d masks," in *2013 International Conference of the BIOSIG Special Interest Group (BIOSIG)*. IEEE, 2013, pp. 1–8.

[5] N. Kose and J.-L. Dugelay, "On the vulnerability of face recognition systems to spoofing mask attacks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 2357–2361.

[6] Y. Li, K. Xu, Q. Yan, Y. Li, and R. H. Deng, "Understanding osn-based facial disclosure against face authentication systems," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 413–424.

[7] E. Wenger, S. Shan, H. Zheng, and B. Y. Zhao, "Sok: Anti-facial recognition technology," in *2023 2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 134–151. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00008

[8] M. Tan, Z. Zhou, and Z. Li, "The many-faced god: Attacking face verification system with embedding and image recovery," in *Annual Computer Security Applications Conference*, 2021, pp. 17–30.

[9] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, "Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition," in *Proceedings of the 2016 acm sigsac conference on computer and communications security*, ser. CCS '16, 2016, pp. 1528–1540.

[10] S. Shan, E. Wenger, J. Zhang, H. Li, H. Zheng, and B. Y. Zhao, "Fawkes: Protecting privacy against unauthorized deep learning models," in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 1589–1604.

[11] C. Li, L. Wang, S. Ji, X. Zhang, Z. Xi, S. Guo, and T. Wang, "Seeing is living? rethinking the security of facial liveness verification in the deepfake era," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2673–2690. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-changjiang

[12] Z. Ming, J. Chazalon, M. M. Luqman, M. Visani, and J.-C. Burie, "Facelivenet: End-to-end networks combining face verification with interactive facial expression-based liveness detection," in *2018 24th International Conference on Pattern Recognition (ICPR)*. IEEE, 2018, pp. 3507–3512.

[13] Y. Li, Y. Li, Q. Yan, H. Kong, and R. H. Deng, "Seeing your face is not enough: An inertial sensor-based liveness detection for face authentication," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1558–1569.

[14] D. Wen, H. Han, and A. K. Jain, "Face spoof detection with image distortion analysis," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 4, pp. 746–761, 2015.

[15] D. Tang, Z. Zhou, Y. Zhang, and K. Zhang, "Face flashing: a secure liveness detection protocol based on light reflections," in *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[16] APP.MI.COM, "Pingan Jinguanjia - Xiaomi App Market," 2023, https://app.mi.com/details?id=com.pingan.lifeinsurance.

[17] The National Computer Network Emergency Response Technical Team/Coordination Center of China, "China National Venerability Database," 2022, https://www.cnvd.org.cn/.

[18] U.S. Government Accountability Office, "Facial Recognition Technology: Privacy and Accuracy Issues Related to Commercial Uses," 2022, https://www.gao.gov/products/gao-20-522.

[19] Samsung, "Use facial recognition security on a galaxy phone or tablet," 2022, https://www.samsung.com/us/support/answer/ANS00062630/.

[20] Huawei, "Face recognition — huawei support global," 2022, https://consumer.huawei.com/en/support/content/en-us15834578/.

[21] Arm, "TRUSTZONE FOR CORTEX-A-Arm," 2022, https://www.arm.com/technologies/trustzone-for-cortex-a.

[22] Google, "Android Apps on Google Play," 2022, https://play.google.com/store.

[23] Xiaomi, "Xiaomi App Market," 2022, https://app.mi.com/.

[24] Androguard Team, "Androguard," 2023, https://github.com/androguard/androguard.

[25] romainthomas, "lief-project/LIEF," 2023, https://github.com/lief-project/LIEF.

[26] Alibaba, "Aliyun Vision: Facebody," 2022, https://vision.aliyun.com/facebody.

[27] Tencent, "Face Recognition—Tencent Cloud (Webank)," 2022, https://www.tencentcloud.com/products/facerecognition.

[28] Baidu, "Baidu AI Cloud-Infinite Possibilities," 2022, https://intl.cloud.baidu.com/.

[29] SenseTime, "SenseTime: Artificial Intelligence (AI) Software Provider," 2022, https://www.sensetime.com/cn.

[30] Megvii, "Megvii," 2022, https://en.megvii.com/.

[31] FaceTec, "FaceTec.com — 3D Liveness, 3D Face Matching, OCR, & NFC," 2022, https://www.facetec.com/.

[32] PingAn, "Ping An Technology," 2022, https://tech.pingan.com/en/.

[33] Linkface, "Linkface," 2022, https://www.linkface.cn/.

[34] YITU, "YITU Technology — YITU Explore the AI World," 2022, https://www.yitutech.com/en.

[35] Alipay, "ZOLOZ — Home," 2022, https://www.zoloz.com/.

[36] Jumio, "Jumio: End-to-End ID & Identity Verification and AML Solutions," 2022, https://www.jumio.com/.

[37] CloudWalk, "CloudWalk Technology Co., Ltd." 2022, https://www.cloudwalk.com/en/.

[38] Onfido, "Onfido: Digital identity made simple," 2022, https://onfido.com/.

[39] Daon, "Daon: The Digital Identity Trust Company," 2022, https://www.daon.com/.

[40] Google, "Protect against security threats with SafetyNet — Android Developers," 2022, https://developer.android.com/training/safetynet.

[41] ——, "Play Integrity API — Google Play — Android Developers," 2022, https://developer.android.com/google/play/integrity.

[42] M. Ibrahim, A. Imran, and A. Bianchi, "Safetynot: On the usage of the safetynet attestation api in android," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 150–162. [Online]. Available: https://doi.org/10.1145/3458864.3466627

[43] kdrag0n, "How does Universal SafetyNet Fix work?" 2022, https://github.com/kdrag0n/safetynet-fix/blob/master/docs/details.md.

[44] Frida, "Frida • A world-class dynamic instrumentation toolkit," 2022, https://frida.re/.

[45] X. Community, "Xposed General — XDA Forums," 2022, https://forum.xda-developers.com/f/xposed-general.3094/.

[46] S. AI, "Identity Verification Spoofing With Deepfakes," 2021, https://www.youtube.com/watch?v=SU9K1LsgX7c.

[47] P. co, "Face Liveness Detection spoofing attack," 2021, https://www.youtube.com/watch?v=Nia-5Npzsbg.

[48] W. Ushanka, "Spoof of Innovatrics Liveness - YouTube," 2022, https://www.youtube.com/watch?v=98Ixy-HoMn0.

[49] Forbes, "We 3D Printed Our Heads To Bypass Facial Recognition Security And It Worked — Forbes," 2019, https://www.youtube.com/watch?v=ZwCNG9KFdXs.

[50] J. Määttä, A. Hadid, and M. Pietikäinen, "Face spoofing detection from single images using micro-texture analysis," in *2011 international joint conference on Biometrics (IJCB)*. IEEE, 2011, pp. 1–7.

[51] Z. Boulkenafet, J. Komulainen, and A. Hadid, "Face anti-spoofing based on color texture analysis," in *2015 IEEE international conference on image processing (ICIP)*. IEEE, 2015, pp. 2636–2640.

[52] O. Foundataion, "OWASP Foundation, the Open Source Foundation for Application Security — OWASP Foundation," 2022, https://owasp.org/.

[53] O. M. team, "OWASP MASTG," 2022, https://mas.owasp.org/MASTG/.

[54] O. Zungur, A. Bianchi, G. Stringhini, and M. Egele, "Appjitsu: Investigating the resiliency of android applications," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021, pp. 457–471.

[55] S. Berlato and M. Ceccato, "A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps," *Journal of Information Security and Applications*, vol. 52, p. 102463, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212619305976

[56] scottyab, "rootbeer," 2021, https://github.com/scottyab/rootbeer.

[57] Q. Super Six, "Anti-debugging Skills in APK," 2016, https://www.programmersought.com/article/1472621633/.

[58] aimardcr, "APKKiller," 2022, https://github.com/aimardcr/APKKiller.

[59] lief project, "How to use frida on a non-rooted device," 2022, https://lief-project.github.io//doc/latest/tutorials/09_frida_lief.html.

[60] Darvin, "Detect Frida for Android," 2019, https://darvincitech.wordpress.com/2019/12/23/detect-frida-for-android/.

[61] stackoverflow, "Android disable ptrace debug for security," 2019, https://stackoverflow.com/questions/56179346/android-disable-ptrace-debug-for-security.

[62] framgia, "android-emulator-detector," 2017, https://github.com/framgia/android-emulator-detector.

[63] topjohnwu, "Magisk," 2022, https://github.com/topjohnwu/Magisk.

[64] Dr-TSNG, "Shamiko," 2022, https://github.com/LSPosed/LSPosed.github.io/releases.

[65] Google Developers, "Native APIs — Android NDK — Android Developers," 2022, https://developer.android.com/ndk/guides/stable_apis#camera.

[66] Qualcomm, "Mobile Security Suite — Snapdragon Security Platform," 2022, https://www.qualcomm.com/products/features/mobile-security.

[67] O. Wiles, A. Koepke, and A. Zisserman, "X2face: A network for controlling face generation using images, audio, and pose codes," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 670–686.

[68] E. Zakharov, A. Shysheya, E. Burkov, and V. Lempitsky, "Few-shot adversarial learning of realistic neural talking head models," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 9459–9468.

[69] L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, "Identity confusion in {WebView-based} mobile app-in-app ecosystems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1597–1613.

[70] Tencent, "Mini Program Development Guide - Weixin public doc," 2022, https://developers.weixin.qq.com/miniprogram/en/introduction/.

[71] FIDO Alliance, "FIDO2 - FIDO Alliance," 2022, https://fidoalliance.org/fido2/.

[72] ——, "News: Your Google Android 7+ Phone Is Now a FIDO2 Security Key," 2022, https://fidoalliance.org/news-your-google-android-7-phone-is-now-a-fido2-security-key/.

[73] Y. Dong, H. Su, B. Wu, Z. Li, W. Liu, T. Zhang, and J. Zhu, "Efficient decision-based black-box adversarial attacks on face recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7714–7722.

[74] C. Yang, A. Kortylewski, C. Xie, Y. Cao, and A. Yuille, "Patchattack: A black-box texture-based attack with reinforcement learning," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXVI*. Springer, 2020, pp. 681–698.

[75] Z. Deng, K. Chen, G. Meng, X. Zhang, K. Xu, and Y. Cheng, "Understanding real-world threats to deep learning models in android apps," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 785–799.

[76] P. Korshunov and S. Marcel, "Deepfakes: a new threat to face recognition? assessment and detection," *arXiv preprint arXiv:1812.08685*, 2018.

[77] S. Tripathy, J. Kannala, and E. Rahtu, "Icface: Interpretable and controllable face reenactment using gans," in *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, 2020, pp. 3385–3394.

[78] Y. Cao and J. Yang, "Towards making systems forget with machine unlearning," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 463–480.

[79] Y. Cao, A. F. Yu, A. Aday, E. Stahl, J. Merwine, and J. Yang, "Efficient repair of polluted machine learning systems via causal unlearning," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 735–747.

[80] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 447–458.

[81] S.-T. Sun, A. Cuadros, and K. Beznosov, "Android rooting: Methods, detection, and evasion," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015, pp. 3–14.

[82] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 356–367.

[83] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame, "You shall not repackage! demystifying anti-repackaging on android," *Computers & Security*, vol. 103, p. 102181, 2021.

[84] K. Lim, Y. Jeong, S.-j. Cho, M. Park, and S. Han, "An android application protection scheme against dynamic reverse engineering attacks." *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, vol. 7, no. 3, pp. 40–52, 2016.

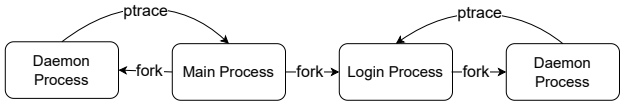
[85] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework." in *NDSS*, 2015.

[86] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen, "Uranine: Real-time privacy leakage monitoring without system modification for android," in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 256–276.

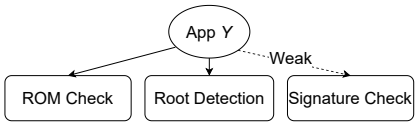
[87] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao, and Y. Chen, "FlowCog: Context-aware semantics extraction and analysis of information flow leaks in android apps," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018, pp. 1669–1685.

APPENDIX A
MANUALLY REVERSE-ENGINEERING APPS

To evaluate the security of XFVSes, we have two security experts manually reverse engineer the apps to assess how they meet the proposed security properties. To demonstrate how our experts work and to understand the difficulty of establishing a reliable environment (SP1) for the apps, we showcase the details of the two apps below.



(a) Environment Protection of *X*.



(b) Environment Protection of *Y*.

Fig. 8. Two Examples of Incomplete SP1 Protection.

Case1: Commercial Bank App *M*. The first example is a commercial bank app. It creates new processes when accessing highly sensitive operations (e.g., the Login Process shown in Fig. 8(a)), and each worker process forks a daemon process to ptrace itself to prevent debugging. Meanwhile, once the worker process detects that its daemon process has terminated, it will also terminate immediately. This effectively prevents attackers from injecting code into the process using Frida. However, we find that this app lacks runtime environment checks, so we can directly patch the .so file to replace the ptrace-related code with the nop instructions. Then we can attach Frida to the target process, and then control its execution code and data.

Case2: Government Affair App *N*. The second example is a government-related app, which is launched with strong detection of custom ROM and root privilege, and its code is heavily obfuscated, thus it is difficult to locate or modify them. However, we find that this app lacks sufficient signature checks, therefore can not guarantee the integrity of its own code. We inject Frida-Gadget into its APK file by repackaging, and then we can use Frida to hook it without root privilege.

The above examples show that although XFVS apps may employ a variety of defensive measures, it is difficult to fundamentally eliminate attacks by local attackers.

APPENDIX B
DETECTING XFVS APPS

We describe more details to detect XFVS apps here, as shown in Algorithm 1. To find the XFVS code, for the set S of all Java fully qualified class names and native JNI method names in an app, we compute the set P of prefixes of length 2 for all strings in S. We group strings with the same prefix in S according to each prefix p in P. We prepare a set K of keywords related to face verification in advance and calculate the $Proportion$ of strings with keywords in set K in each group separately. This $Proportion$ represents the face verification semantics cohesion of the code, and if the value exceeds the threshold θ, the group is considered as XFVS code. Therefore, the prefix p of the group is added to the result set R. Otherwise, we check the semantics cohesion of groups with longer prefixes (Len + 1).

Algorithm 1 Face Verification Code Clustering

Input: Java class names and native function names S, face and verification keywords K; semantics cohesion threshold θ.

```
ResultSet R = ∅ , PrefixSet P = getAllPrefix(S, Len=2)
for p in P do
    All = getAllStringsWithPrifix(S, p)
    Matched = getAllStringsHasKeywords(All, K)
    Proportion = |Matched| / |All|
    if Proportion ≥ θ then
        Add p to R
    else
        P' = getAllPrefix(All, Len+1)
        Add all item of P' to P
Return R
```

APPENDIX C
VULNERABILITY REPORT

We receive 14 CNVD IDs, as listed in Table VIII. Note these CNVDs are confidential for 10 years.

TABLE VIII
Accepted CNVD IDs.

Certificate NO.	State	Fixed Date
CNVD-2022-74482	Accepted & Fixed	2022-09-21
CNVD-2022-74483	Accepted & Fixed	2022-09-21
CNVD-2022-74484	Accepted & Fixed	2022-09-21
CNVD-2022-61291	Accepted & Fixed	2022-08-03
CNVD-2022-61293	Accepted & Fixed	2022-08-03
CNVD-2022-54088	Accepted & Fixed	2022-06-29
CNVD-2022-43391	Accepted & Fixed	2022-05-05
CNVD-2022-41381	Accepted & Fixed	2022-04-19
CNVD-2022-21775	Accepted & Fixed	2022-03-02
CNVD-2022-25891	Accepted & Fixed	2022-02-28
CNVD-2022-25895	Accepted & Fixed	2022-02-24
CNVD-2022-25896	Accepted & Fixed	2022-02-24
CNVD-2021-86899	Accepted & Fixed	2021-11-12
CNVD-2021-86898	Accepted & Fixed	2021-11-12

APPENDIX D
SCORING RULES

To ensure the objectivity and accuracy of the evaluation results, we have established strict scoring rules for each security property, as listed below.

TABLE IX
The Scoring Rules for Evaluating How XFVS Apps Meet the Four Security Properties.

SPs	Score	Scoring Rules
SP1: Reliable Environment	☆	The app checks no resiliency items in Table 6.
	★	The app checks one resiliency item in Table 6.
	★☆	The app checks two resiliency items in Table 6.
	★★	The app checks three resiliency items in Table 6.
	★★☆	The app checks four resiliency items in Table 6.
	★★★	The app checks five resiliency items in Table 6.
SP2: Camera Security	☆	The app uses framework (Java) camera APIs to get images/videos with no protection.
	★	The app uses native (C/C++) camera APIs to get images/videos with no protection.
	★☆	The app uses framework (Java) camera APIs to get images/videos with protection methods, including validating frames from both front and back cameras, and getting images of different resolutions to prevent camera injection.
	★★	The app uses native (C/C++) camera APIs to get images/videos with protection methods, including validating frames from both front and back cameras and getting images of different resolutions to prevent camera injection.
	★★☆	The app uses framework or native camera APIs but obtains images/videos by recording the screen to avoid direct camera injection.
	★★★	The app uses secure camera to get images/videos.
SP3: Reliable Liveness	☆	The app only uses silent liveness detection on the client side, with no liveness detection on the server side.
	★	The app uses action-based/reflection-based liveness detection with fixed action/flash sequences on the client side, with no liveness detection on the server side.
	★☆	The app uses action-based/reflection-based liveness detection with fixed action/flash sequences on the client side, with silent liveness detection on the server side.
	★★	The app uses action-based/reflection-based liveness detection with random action/flash sequences on the client side, with silent liveness detection on the server side.
	★★☆	The app uses action-based/reflection-based liveness detection with random action/flash sequences on the client side, with action-based/reflection-based liveness detection on the server side.
	★★★	The app uses randomized interactive liveness detection on client side, and also server-side liveness detection that cannot be bypassed by our black-box testing.
SP4: Data Consistency	☆	The app does not validate data consistency.
	★	The app checks data consistency by calculating a hash, e.g. MD5, for the transmitted data (face data, liveness configurations, and validation results).
	★☆	The app checks data consistency by calculating a secure hash, e.g. HMAC, for the transmitted data (face data, liveness configurations, and validation results).
	★★	The app checks data consistency by calculating a PKI-based signature for the transmitted data (face data, liveness configurations, and validation results).
	★★☆	Besides the above methods, the app also deploys image-based validation methods, such as using a watermark for each image to prevent it from being tampered with by attackers.
	★★★	Besides the above methods, the app server checks the consistency of all data it sends, instead of relying on the client's checking results.