Broken Fingers: On the Usage of the Fingerprint API in Android

Antonio Bianchi
University of California, Santa Barbara
antoniob@cs.ucsb.edu

Yanick Fratantonio
University of California, Santa Barbara
EURECOM
yanick.fratantonio@eurecom.fr

Aravind Machiry
University of California, Santa Barbara
machiry@cs.ucsb.edu

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

Simon Pak Ho Chung
Georgia Institute of Technology
pchung34@mail.gatech.edu

Wenke Lee
Georgia Institute of Technology
wenke.lee@gmail.com

***Abstract*—Smartphones are increasingly used for very important tasks such as mobile payments. Correspondingly, new technologies are emerging to provide better security on smartphones. One of the most recent and most interesting is the ability to recognize fingerprints, which enables mobile apps to use biometric-based authentication and authorization to protect security-sensitive operations.**

In this paper, we present the first systematic analysis of the fingerprint API in Android, and we show that this API is not well understood and often misused by app developers. To make things worse, there is currently confusion about which threat model the fingerprint API should be resilient against. For example, although there is no official reference, we argue that the fingerprint API is designed to protect from attackers that can completely compromise the untrusted OS. After introducing several relevant threat models, we identify common API usage patterns and show how inappropriate choices can make apps vulnerable to multiple attacks. We then design and implement a new static analysis tool to automatically analyze the usage of the fingerprint API in Android apps. Using this tool, we perform the first systematic study on how the fingerprint API is used.

The results are worrisome: Our tool indicates that 53.69% of the analyzed apps do not use any cryptographic check to ensure that the user actually touched the fingerprint sensor. Depending on the specific use case scenario of a given app, it is not always possible to make use of cryptographic checks. However, a manual investigation on a subset of these apps revealed that 80% of them could have done so, preventing multiple attacks. Furthermore, the tool indicates that only the 1.80% of the analyzed apps use this API in the most secure way possible, while many others, including extremely popular apps such as Google Play Store and Square Cash, use it in weaker ways. To make things worse, we find issues and inconsistencies even in the samples provided by the official Google documentation. We end this work by suggesting various improvements to the fingerprint API to prevent some of these problematic attacks.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23079
www.ndss-symposium.org

I. Introduction

As smartphones become widely used, more and more security-sensitive tasks are performed using these devices. For instance, mobile payment or mobile banking applications have been steadily increasing for the past few years [11]. That is, smartphones are increasingly used to access remote accounts containing valuable and sensitive user information such as purchase histories or health data. Needless to say, the security of smartphones and mobile apps, including authenticity, integrity, and confidentiality, is of paramount importance.

Smartphone technologies bring both new opportunities and threats to security. A smartphone is a very convenient choice to be the "second factor" in two-factor authentications (2FA) because the users do not have to carry additional security tokens. A very common two-factor scheme is to authenticate a user based on both the user's password and proof that the user is in possession of her smartphone, with the latter commonly achieved by sending text messages to the registered smartphone. On the other hand, as more and more sensitive operations that are protected by 2FA are performed using smartphones, the security threat from a stolen/compromised phone or malicious apps running on the phone significantly increases. In particular, by performing sensitive operations on smartphones, both factors required by 2FA will be available on the smartphone, making it a single point of failure.

In theory, technologies commonly available on modern smartphones can be used to implement 2FA schemes that are secure even in the face of stolen/compromised phones or malicious apps running on the phone. In particular, most smartphones already come with Trusted Execution Environments (TEE) that can be used to generate and store cryptographic keys.[1] Furthermore, the TEE can already be programmed to directly communicate with a fingerprint reader (which is widely available on modern smartphones) so that it will only perform operations using the stored keys when the fingerprint reader detects a registered fingerprint,

[1]In devices running Android, the TEE is typically enforced by using the ARM TrustZone technology [5].

signaling the user's explicit consent to such operations. Since the TEE is a hardware-enforced isolated execution environment, the keys it stores and the operations performed with those keys cannot be leaked or misused even if the smartphone's operating system (OS) is compromised.[2]

A second factor implemented by combining the TEE and the fingerprint reader is at least as strong as what proposed in the Security Key protocol [32] (and implemented by YubiKey [52]), the current state-of-the-art authentication solution in the desktop world, promoted by Google, as a member of the FIDO Alliance [20]. Under the Security Key protocol, a cryptographic private key is stored on an external hardware device and is used to *sign* authentication tokens provided by the remote service the user wants to authenticate with. This signing operation only happens if the user authorizes it, by pressing a physical button on the external hardware device. In fact, one can argue that a second factor that combines a smartphone's TEE and its fingerprint reader is going to provide more security than YubiKeys in the scenario where the hardware security token is stolen; in the former, the attacker cannot misuse the hardware token without the owner's fingerprint, while in the latter, anybody in possession of the token can misuse it to bypass 2FA. Additionally, the device's screen (which is not present in standard hardware tokens), could be used to inform users about the operation they are authorizing by touching the sensor.

Motivated by the significant security benefits that the TEE-backed fingerprint sensor can offer, in this work we perform the first comprehensive study on the usage of the fingerprint API in Android. In particular, we first systematically explore the various nuances of this API, and we uncover several aspects, many of which subtle, that can lead to this complex API to be misused. As an example, developers could just check if the user touched the sensor, without binding this operation to the usage of a cryptographic key, contrary to what is suggested by Google's guidelines [23].

We then bring some clarity to the many threat models that should be considered when performing security evaluations concerning the fingerprint API. For example, we explore what are the capabilities of an attacker that can compromise the untrusted operating system, i.e., a "root attacker." At first glance, one may say that a root attacker will trivially defeat any fingerprint API and that the fingerprint API itself is not designed to protect from root attackers. On the contrary, *we argue that many important design choices related to this API are motivated specifically to protect from root attackers.* The most significant example is that current implementations of the fingerprint API work by unlocking a TEE-backed cryptographic keystore: if the threat model were not considering root attackers, apps could simply store cryptographic material in app-private storage (that non-root attackers cannot access), without needing to rely on any TEE support.

We hypothesized that the lack of clearly stated design goals and, as we will see, misleading documentation bring confusion and app developers might misuse this API. To explore this hypothesis, we first developed a static analysis approach to characterize how Android apps use the fingerprint API, whether this API is misused, and how they are resilient to the various threat models. We then use this system to perform the first systematic empirical study of how the current fingerprint API is used in the Android ecosystem. Specifically, we used our tool to analyze 501 apps requiring the fingerprint permission (out of a dataset of 30,459 popular apps). The results are worrisome. For example, the tool identified that 53.69% of the apps, including the widely deployed Google Play Store app, do not make use of the cryptographic keystore unlocked by a successful fingerprint touch: this means that a root attacker can easily completely bypass the fingerprint security mechanism by just programmatically "simulating" the user's touch to, for example, perform in-app purchases.

One explanation for this low percentage could be that not all use case scenarios for the fingerprint API can be protected from root attackers. One example is an app that uses the fingerprint to merely assess user presence: in this case, it is very challenging to find a "role" for the cryptographic material, and it is thus not possible to protect this use case from root attackers. To determine how many apps fall into this category, we then performed manual analysis on a subset of applications flagged as problematic. For example, we manually analyzed a random subset of 20 apps for which our tool identify usages of the fingerprint API flagged as "fully bypassable." To our surprise, 16 of them are apps that use the fingerprint API to authenticate the user against a remote backend, or apps that store secret information: These are *exactly* the use case scenarios that a proper usage of the fingerprint API could easily protect even from powerful attackers such as root attackers. This manual analysis effort, even though admittedly limited, suggests that the number of apps misusing the fingerprint API is significant. Moreover, our tool also flagged only the 1.80% of the apps in our dataset as using the fingerprint API to *sign* transactions, which is the most secure way to use this API.

In summary, this paper makes the following contributions:

- We systematically study the various ways in which the fingerprint API can be used in Android and how attackers with different capabilities can exploit sub-optimal usages of it.
- We develop a static-analysis tool to automatically identify how real-world popular apps use the fingerprint API. We make its code publicly available online [8].
- By using this tool, we perform the first systematic study of the usage of the fingerprint API in Android, and we uncover a significant number of apps potentially misusing the fingerprint API. This improper usage significantly weakens the security guarantees these apps could achieve if using the API correctly.
- We identify shortcomings and weaknesses of the current API and its implementation, and we propose different improvements to it.

II. Background

A. Android Security Mechanisms

The Android operating system is a customized Linux kernel on top of which the Android framework runs. User-installable third-party apps run as user-mode processes and are typically written in Java, even though apps may also include libraries written in native code. These apps interact with the Android framework using system calls or invoking remote procedures in "system services."

[2]As an empirical measure, among all the vulnerabilities mentioned in the "Security Bulletins" released by Google about Android security [27] up to August 2017, 33 of them allow an attacker "to execute arbitrary code within the context of the kernel," whereas only 2 allow "to execute arbitrary code in the TrustZone context."

Third-party apps run in separate containers with isolated resources (e.g., private files) and a limited set of capabilities. The precise list of capabilities is determined by the "Android permissions" granted to an app. In modern Android versions (starting from Android 6), permissions classified as dangerous need to be specifically approved by the user. Other permissions are instead automatically granted to any app that requires them, but the app still needs to request them in its Manifest file. The `USE_FINGERPRINT` permission, which grants the ability to use the fingerprint reader sensor, is an example of "normal" permission.

This separation between different apps and the different apps' capabilities is enforced by using a combination of standard Linux mechanisms (e.g., Linux groups), SELinux rules, and specific checks in the Android framework. In fact, apps cannot perform any sensitive operation directly, but they have to send a request to a running system service, which verifies whether the app calling it has the permission required to perform the requested operation. Thus, system services (which run as users with higher privilege than normal apps) mediate most of the interactions between apps and the kernel.

Attackers often try to exploit bugs in either the system services or the kernel to gain *root* privileges, using what are typically called "root exploits." Although a significant effort has been made to limit the attack surface exposed by system services and the kernel to normal apps [44], root exploits are still a concrete danger in the Android ecosystem [27]. However, even when an attacker can fully compromise the Linux kernel, achieving persistent kernel-level code execution (by bypassing the Verified Boot mechanism) requires further exploitation of the system [30]. Similarly, achieving code execution within the TrustZone-enforced TEE, which we describe in the next section, requires the exploitation of significantly less common vulnerabilities in the relatively small code base running within the TEE.

B. TEE and TrustZone

A TEE is an isolated environment designed to execute sensitive code with more isolation/protection than what provided by a standard "feature-rich" operating system. While other instantiations of TEE exist, in this paper, we will focus on ARM's implementation of the TEE, called TrustZone, which is available on the majority of Android devices.

Under ARM's TrustZone, a "trusted" kernel and a set of Trusted Applications (TAs) run in the "secure" world, isolated by hardware from the Android OS and third-party apps, which, conversely, run in the "non-secure" world. Only code signed by the hardware manufacturer can run in the "secure" world. Also, while third-party apps run in isolation from the TAs, these apps can utilize services provided by the TAs through well-defined APIs. Two services offered by the TAs are relevant to fingerprint-based authentication, which is the focus of this paper:

- **keymaster:** It allows to create cryptographic keys, store them inside secure-storage, and use them to encrypt, decrypt, verify, or sign data, coming from the untrusted world. Internally, this service utilizes the secure-storage capability offered by the trusted kernel to securely store encrypted and authenticated data on the device's mass memory.
- **fingerprintd:** It handles the storage of fingerprint data, acquired from the fingerprint reader sensor, and verifies that the finger touching the sensor corresponds to any previously registered fingerprint. It is important to notice that "raw" fingerprint data (i.e., the image of the registered fingerprint) never leaves the TEE and therefore it is not accessible by any untrusted code.

C. The Fingerprint API in Android

In the discussions that follow, we will focus on apps that access the fingerprint reader (which is commercially named the "Imprint" sensor) through the Java API provided by Google. Unless otherwise specified, we will consider the implementation of this API running in Android version 7 on Google's devices. In particular, for our experiments we used a Google's Nexus 5X.

Also, we will follow Google's [23] and OWASP [35] guidelines and consider that the best way to use the fingerprint reader is in conjunction with some cryptographic operations. In particular, instead of just recognizing the legitimate user has touched the fingerprint sensor, an app should use this fingerprint reading to unlock a cryptographic key protected by the TEE. In other words, by utilizing both the **keymaster** and the **fingerprint** in the TrustZone, this method can guarantee that even an attacker with root privilege cannot misuse the cryptographic key without presenting the right fingerprint. As we will see in Section V, the latter method is significantly stronger.

We will now briefly provide the major steps an app has to perform to interact with the fingerprint sensor and determine whether a legitimate user touched it. For clarity, we will omit unnecessary details of the complex Android cryptographic API, and we suggest interested readers to read the official documentation for a more detailed explanation [28], [26].

Generate a cryptographic key: An app can generate a cryptographic key or a public/private key pair by using the method `initialize` of the class `KeyGenerator` or `KeyPairGenerator`. Developers must specify properties of the generated key (e.g., the algorithm used) by passing a `KeyGenParameterSpec` object to the mentioned `initialize` method.

Among the various aspects a developer can control about a generated key, the most important one in this context is triggered by calling the `setUserAuthenticationRequired` method (passing `true` for its `required` parameter). By calling this method, a developer can ensure that the generated key is usable (i.e., it is "unlocked") only after a legitimate user has touched the fingerprint reader sensor. In case a *pair* of keys is generated, calling this method will only constraint the usage of the private key, leaving the public one freely accessible by the app.

Unlock the key by authenticating the user: By calling the `authenticate` method, an app activates the fingerprint reader sensor. Two parameters of this method are important: the cryptographic key that is unlocked if a legitimate user touches the sensor and a list of callback functions, called after the sensor is touched.

Override the fingerprint callbacks: When a user touches the sensor, specific callback functions are called. In particular, the method `onAuthenticationSucceeded` is called when a legitimate user touches the sensor, whereas other callback functions are called in case of error conditions (e.g., a non-legitimate user touched the sensor).

Use the unlocked key: After the `onAuthenticationSucceeded` method is called, an app should use the now unlocked key. For authentication purposes, Google's guidelines suggest the use of a previously generated private key to *sign* a server-provided authentication token and then send this authentication token to the app's remote backend.

It is worth mentioning two properties of the generated keys. First, the Android framework ensures that only the app generating a key can use it. Second, in modern devices, private keys are stored within the TEE (an app can verify if in a specific device keys are stored within the TEE by calling the `isInsideSecurityHardware` API) and cannot be exported (not even by the app generating them and not even after a legitimate user has touched the fingerprint sensor). In other words, "unlocking" a key does not allow an app to read its "raw" value, but only to use it to encrypt, decrypt, or sign data. If the key is stored in the TEE, these operations are guaranteed to happen *within* the TEE.

D. Two-Factor Authentication Schemes

To overcome security and usability limitations of classical username and password authentication, many service providers suggest or mandate the usage of an additional "second factor" during authentication. One common solution is to use a One-Time Passcode (OTP). However, OTPs are still vulnerable to phishing and man-in-the-middle attacks [15], [43] and have serious usability drawbacks, since they require the user to somehow receive the OTP code and insert it into the authentication interface. Furthermore, protocols based on OTPs rely on the confidentiality of the communication channel of the OTP, which is often not guaranteed. For instance, text messages are a common communication channel used to send OTPs to smartphones. However, the insecurity of this channel has been shown in many occasions [46], [22].

Secure authentication schemes using challenge/response offer better security and usability. In particular, the current state-of-the-art is constituted of the Security Keys formalized in the Universal Second Factor (U2F) protocol [51]. This protocol is composed of two phases. During the registration phase, a key pair is generated in an external hardware device. The generated public key is sent to the remote server, whereas the private key remains securely stored within the hardware device. Later, during the authentication phase, the server sends the client a challenge. The client then asks the hardware device to sign this challenge with the stored private key, and the signed response is then sent back to the remote server, which can verify it using the previously obtained public key. Both during the registration and the generation phases, the user is required to physically touch the hardware device as a Test of User Presence (TUP) to authorize creation and usage of cryptographic keys.

III. Threat Model

This section explores the different threat and attacker models considered in this paper. We first define different "levels of compromise" that an attacker may achieve. Then, we discuss several different threat models, ranging from being just able to install a malicious app on the victim's device to be able to fully compromise the Android Linux (untrusted) operating system. We will also argue why each of these threat models are particularly relevant for any work studying the fingerprint API. We end this section by clarifying which threat models are considered as out of scope.

A. Levels of Compromise

To ease our exposition, we now define three labels describing three different levels of compromise an attacker can achieve in the different scenarios. We discuss the three levels starting from the least powerful. We note that, of course, an attacker will always attempt to achieve the third and most powerful level of compromise. However, depending on the attacker capabilities and how a given app uses the fingerprint API, this may not always be possible.

Confused Deputy. An attacker might be able to interfere with the usage of the fingerprint API to change the intended effect a user wants to achieve when she touches the fingerprint sensor. For example, consider a user who wants to authorize the transaction "pay \$1,000 to $Friend$" by pressing the fingerprint sensor: an attacker might be able to change this transaction to "pay \$1,000 to $Attacker$." Another example is an attacker that can lure the user to provide the fingerprint by spoofing a completely unrelated scenario, such as the lock screen.

More in general, these examples are instances of a *confused deputy problem*. An attacker can achieve her goal by abusing this problem, but she needs the user to touch the fingerprint sensor *once for each malicious attempt*.

Once For All. In this scenario, the attacker can completely bypass the need for "fingerprint" by just luring the user to provide a fingerprint *once*. That is, after the attacker obtains one fingerprint, the attacker can spoof any subsequent fingerprint request. We note that, in this context, the term "spoofing" does not entail spoofing the "real" physical fingerprint. Instead, with this term, we indicate that an attacker can trick the vulnerable app, and the backend it communicates with, to believe a legitimate fingerprint was provided.

As a representative example, consider an app that, after the user provides a fingerprint, decrypts, using a TEE-backed cryptographic key, an authentication token. If an attacker manages to access this decrypted token, the attacker can now just reuse the token undisturbed for subsequent authentication and authorization attempts, without needing to lure additional fingerprints. Thus, this scenario provides a more practical opportunity for an attacker.

Full Fingerprint Bypass. In this last case, an attacker can completely bypass the need of luring fingerprint touches without requiring a "real" touch, not even once. For example, consider a banking app that requires the user to confirm every monetary transaction by pressing the fingerprint sensor. If an attacker can compromise the app to this last level, the attacker can authorize an unlimited number of transactions, at will, without having the user touch the sensor. This case provides significant practicality benefits for an attacker. In fact, the attacker does not need to "wait" to hijack a user's touch: as a matter of fact, in this scenario the attack does not need any user interaction at all.

We note that it may not always be possible for a root attacker to indefinitely wait for a user's touch, because, for instance, thanks to the Verified Boot protection mechanism, it may be impossible to persistently compromise a device.

B. Attacker Capabilities

We consider the following three increasingly powerful attacker capabilities.

Non-Root Attacker. In this threat model, we consider an attacker that is just able to install a malicious application on the victim's device. In this case, we assume that the attacker is unable to subvert the security of the operating system, and therefore the installed malicious app is still constrained by all the limitations imposed by the Android framework. The installed app can, however, request permissions (as any other benign third-party app installed on the device) to obtain specific capabilities, and, in this case, we assume that the user will grant them.

Additionally, the installed app, can show maliciously crafted messages or, more in general, interfere with the device's user interface (UI), to lure a legitimate user to touch the fingerprint reader sensor. These UI attacks greatly vary in terms of complexity and flexibility, and they are well explored by several existing works [34], [14], [9], some of which, such as Cloak & Dagger [21], achieve almost complete compromise of the device. While these attacks are indeed powerful, we note that the fingerprint API might be one of the few aspects that could, at least in principle, prevent full compromise. In fact, even though the Cloak & Dagger attack can simulate arbitrary user input, it cannot "spoof" a physical fingerprint user's touch.

The key conceptual point here is that there is no trusted path from the fingerprint API to the UI. Thus, as previous works have shown, the attacker can exploit an instance of the confused deputy problem. We postpone the discussion on the practicality and implications of these attacks to Section VI-B.

Root Attacker. In this threat model, we assume that an attacker can fully compromise the Android operating system, by using, for instance, a "root exploit." Therefore, the attacker can completely bypass apps' restrictions put in place by the Android framework. For example, the attacker can access app-private storage (which is usually protected by the sandboxing mechanism). Moreover, exploiting confused deputy instances via the UI attacks mentioned above becomes much simpler for a root attacker.

Additionally, the attacker can spoof "messages" from the operating system: Specifically, an attacker can freely communicate with the TEE, and thus send arbitrary messages to it. At this point the attacker can programmatically invoke the `onAuthenticationSucceeded` method implemented within the victim app (and thus simulating a user's touch), even if the user has never touched the fingerprint sensor.

We note that, although a root attacker is powerful, she does not get access to everything. In particular, the fingerprint API enforces the following three security properties even on a system in which the untrusted OS is completely compromised:

1) an attacker cannot retrieve "raw" fingerprint data;
2) an attacker cannot retrieve the value of cryptographic key stored into the TEE (i.e., keys are not *exportable*);
3) an attacker cannot use TEE-backed cryptographic keys, unless a legitimate user touches the fingerprint sensor.

However, if the victim app does not properly use such TEE-backed cryptographic keys, the attacker might be able to achieve her goal anyways, as we will explain later.

That being said, we also note that, for some usage scenarios, an app does not have any technical way to secure itself from root attackers. For example, if the app uses fingerprint not to secure a secret or token, but as a local "Test of User Presence" (TUP), there is currently no way a developer could make use of cryptographic algorithms. On the other hand, crypto primitives can be definitively used when implementing remote user-authentication mechanisms. We postpone the discussion about these scenarios to Section VI-A.

Finally, for this threat model, we will assume that the device is not in a compromised state when the cryptographic keys ("unlocked" by touching the fingerprint sensor) were first created by the app that the attacker wants to compromise. The creation of cryptographic keys typically happens only during the first usage of an app and, therefore, it may be impossible for an attacker to interfere with their creation if the compromise of a device happens only after this stage of an app's lifecycle.

Root-at-Bootstrap Attacker. In this threat model, we consider an attacker with the same capabilities of the previous one. Additionally, we also assume that the device is in a compromised state even in the moment in which the victim's app generates the cryptographic keys. Therefore, in this case, the attacker can interfere with their creation.

C. Out-of-Scope Attacker Capabilities

We assume that the TEE is not compromised. In other words, we consider an attacker that can compromise the code running (or the data stored) within the TEE as out of scope. In fact, an attacker able to compromise the TEE can trivially fully compromise the fingerprint functionality, by stealing all the cryptographic keys in the secure storage. Moreover, as previously mentioned, exploits able to gain this capability for an attacker are extremely rare.

We will consider attacks on the physical recognition of the fingerprint as out of scope. These attacks, although possible [42], deal with the physical aspects of the fingerprint acquisition process and with the algorithms used to compare fingerprint data. Conversely, in this paper, we focus on a higher-level aspect: the operations inside TEE that are triggered by the legitimate user touching the fingerprint sensor, the operating system, and the apps using the fingerprint sensor API. Therefore, we will assume that the fingerprint sensor and the code inside the TEE handling it are always able to understand if the user that is touching the sensor is the legitimate one (i.e., a user who has previously registered her fingerprint as valid using the appropriate operating system interface).

IV. Fingerprint API usages

In this section, we will explain how the fingerprint API is used by Android apps. In particular, we will classify apps' usages of the fingerprint API based on if and how cryptographic keys (stored inside the TEE) are used to verify that a legitimate user touched the fingerprint sensor. This aspect has profound implications on what attackers can do to subvert the fingerprint checks and how they can achieve their malicious goals. In Section VI-A, we will then explain how the verification of the user touching the sensor is used as a part of the authentication schemes implemented by apps and their corresponding backends.

A. Weak Usage

The easiest way to use the fingerprint API is to execute some code after a legitimate user touched the sensor, without using any cryptography. To achieve this, a developer just has to call the `authenticate` method to activate the fingerprint reader sensor and override the `onAuthenticationSucceeded` method to be notified when the user touched it.

From the implementation standpoint, recall that the `authenticate` method takes, as an argument, the cryptographic key that is unlocked when the user touches the sensor (see Section II-C). Thus, an app can set this parameter to `NULL` and, as a side-effect, the fingerprint will not unlock any cryptographic keystore. Of course, an app could also require access to the keystore and it could then discard this object without using it. In other words, a specific fingerprint-protected functionality is not "protected" by cryptographic operations if a cryptographic key is unlocked but never properly used.

B. Decryption Usage

In this case, a cryptographic key is created, stored inside the TEE, and used to decrypt (once the key is "unlocked" by a legitimate user touching the fingerprint sensor) locally stored files. Google's guidelines suggest using the fingerprint API in this way when "securing access to databases or offline files." In practice, we have seen this method often used to decrypt an authentication cookie stored in an encrypted vault within the app's private storage. This authentication cookie, typically valid for multiple sessions, can be used by the app to authenticate with the remote server.

We have found two ways in which this mechanism is implemented. The easiest case is when a *symmetric* key is created and used to encrypt/decrypt the content of the "encrypted vault." The disadvantage of this method is that it requires the user to touch the sensor (to "unlock" the key) to both *read* something from the vault and to *write* something into it. As a consequence, if, for instance, the remote backend decides to change the value of the authentication cookie stored inside the vault, the user would need to touch the fingerprint sensor to unlock the key.

A more user-friendly way is to use an *asymmetric* key pair. In this case, the public key (which does not need to be "unlocked" before usage), is used to *write* inside the vault, and the private key (which requires the user's touch) is only used to *read* from the vault (e.g., when the stored authentication cookie is needed to authenticate with the app's backend).

Surprisingly, the example officially provided by Google [25] about using the fingerprint API together with a symmetric key does not show how to use cryptography safely. In fact, the provided code generates a symmetric key and, after the user touches the sensor, uses it to encrypt a fixed, hardcoded string. Then, the code just checks whether the encryption operation (performed using the `doFinal` API) threw an exception, an indication that the used key is (still) locked (i.e., it has not been unlocked). While the intent might have been to verify that the user has touched the sensor, this particular example code makes the usage of cryptography pointless because an attacker with "root" privileges can just fake the result of the decryption operation and clear the thrown exception (as we will describe better in Section VIII-A2). In practice, in terms of security, we consider the Google's example on how to use symmetric keys as a case of *Weak* usage of the fingerprint API, rather than a case of *Decryption* usage.

C. Sign Usage

The fingerprint API can also be used to implement challenge/response authentication schemes. This offers significantly more security over a wide range of attackers, but, unfortunately, it is rarely used by developers.

In this case, typically during the app's first usage, a key pair is generated: the public key is sent to the app's remote backend server, whereas the private one is stored within the TEE. When the app needs to authenticate a user to the remote backend, the following steps take place:

1) The remote backend sends a challenge to the app.
2) The app calls the `authenticate` API to "unlock" the previously stored private key.
3) The legitimate user touches the fingerprint reader sensor, and the private key is "unlocked" by the TEE.
4) The `onAuthenticationSucceeded` method (overridden by the app) is called.
5) The app uses the now-unlocked private key to *sign* the challenge from the app's backend.
6) The app sends the signed challenge to the backend.
7) The backend verifies the signature on the challenge, using the public key previously obtained from the client.
8) The backend communicates to the app the result of the verification and considers the user as authenticated.

D. Sign + Key Attestation Usage

As we discuss in more detail in Section V-C, the "Sign" usage is vulnerable to an attacker that can perform a man-in-the-middle attack at the app bootstrap time, when the initial key exchange takes place. In this attack, the attacker would provide to the backend *her* public key (for which she has the associated private key), and she could then bypass the fingerprint. However, starting from Android 7, a countermeasure to this attack is possible, since Android can provide an "attestation" certificate chain, attesting that a key has been created by a "trusted" TEE. A similar attestation mechanism is present in the Security Keys protocol [32].

To enable it, a developer, when creating a key pair, has to call the `setAttestationChallenge(attestationChallenge)` API with a non-`NULL` value for `attestationChallenge`. Then, the app can retrieve the certificate chain, attesting the generated public key using the `getCertificateChain` API. The app's backend can then verify that the root of this chain is signed by a trustworthy Certificate Authority (typically Google). The certificate, among other pieces of information about properties of the generated keys, contains the `attestationChallenge` previously set, allowing the app's backend to verify that the retrieved key was created as a consequence of a specific request.

V. Protocol Weaknesses and Attack Scenarios

We will now highlight the weaknesses of each usage scenarios described in Section IV. For each identified weakness, we will also determine which classes of attacker (as defined in Section III) can exploit it. Our findings are summarized in Table I.

A. *Weak Usage: Fake TEE response*

In the *Weak* usage scenario, fingerprint-based authentication is considered successful as long as the TEE communicates that a legitimate touch happened. This message is delivered by the OS to the client app (by invoking the `onAuthenticationSucceeded` method). In this case, any entity that can control/impersonate the OS to deliver such message can successfully authenticate and authorize any transaction to the server, without having to wait for the user to present the fingerprint even once. In other words, *any "root" attacker can achieve Full Fingerprint Bypass against Weak usage by faking OS messages.* Additionally, a non-root attacker can exploit confused deputy problems by mounting UI attacks. Once again, these attacks are possible because of the lack of trusted UI in Android. We also note that these attacks are possible independently from the specific attacker capabilities and from the specific usage scenario. We refer the reader to Section VI-B for more details.

B. *Decryption Usage: Replay Attack*

In the *Decryption* usage scenario, the TEE is used to decrypt a value (e.g., an authentication cookie), and the same value is communicated to the client app (and the backend server) for every attempt to authenticate or authorize a transaction. In this scenario, *an attacker only needs to capture this value once to then be able to fully authenticate and authorize any transaction any time in the future, by simply replaying this captured value over and over*.

C. *Sign Usage: Man-in-the-Middle Attack*

In the *Sign* usage scenario, the TEE is used to protect a private key used in a challenge/response scheme. In this scenario, a root attacker cannot easily compromise the system — in a way, she has similar capabilities as a non-root attacker, and she could thus attempt to exploit confused deputy problems via UI attacks.

However, we note that an attacker can launch a man-in-the-middle attack if she can interfere with the "app bootstrap" process, during the initial key exchange. The attack would work in this way: at bootstrap, instead of sending to the backend server the real key output by the TEE, the attacker can use her own key instead. In this way, the attacker can use the key thus registered to answer any future challenge (because the attacker knows both the public and the private key), thus achieving Full Fingerprint Bypass. Clearly, since this attack requires the attacker to have control over when the key exchange is carried out, it is only possible for Root-at-Bootstrap attackers.

D. *Sign + Key Attestation Usage: Key Proxying*

The "Sign + Key Attestation" usage scenario significantly raises the bar for attacks, even for a very powerful attacker such as Root-at-Bootstrap attacker. However, from a conceptual point of view, it is possible to attack this usage scenario as well, by performing a so-called *cuckoo attack* [37]. Specifically, while this mechanism attests that a key has been created by the TEE on a user's device with the goal of preventing an attacker from knowing its private value, it cannot prevent an attacker from "proxying" the app's request for creating a key pair to *her* attacker-controlled device and using the TEE of *her* device. We note that this attack scenario presents serious practicality and scalability issues for the attackers. That being said, we will further discuss this aspect in Section IX-C, where we propose improvements on the current implementation of this mechanism.

TABLE I. SUMMARY OF ATTACK POSSIBILITIES WITH RESPECT TO ATTACKER CAPABILITIES AND FINGERPRINT API USAGE.

Attacker Capabilities \ Fingerprint API Usage	*Weak*	*Decryption*	*Sign*	*Sign + Key Attestation*
Non-Root	C.D. [1]	C.D.	C.D.	C.D.
Root	Full	Once	C.D.	C.D.
Root-at-Bootstrap	Full	Full	Full	C.D.

[1] "C.D." stands for Confused Deputy.

VI. DISCUSSION

This section discusses aspects related to the fingerprint API that are not strictly related to the API itself or to the specific vulnerable "usage scenarios" described above.

A. *Application Contexts*

Typically, the fingerprint API is used as a part of an authentication scheme. In this section, instead of focusing on how apps use the fingerprint sensor in terms of API calls and encryption, we will discuss common functionality apps aim to accomplish when they use the fingerprint sensor.

"Local-Only" Usage. Some apps use the fingerprint API to implement the "screen-lock" functionality. For instance, they prevent access to a list of user-selectable apps, unless the fingerprint sensor is touched by a legitimate user. In this case, the fingerprint sensor just constitutes a local Test of User Presence (TUP).

For these apps, only a *Weak* usage of the fingerprint API is reasonable. In fact, the app does not have any remote backend to authenticate with nor it stores any secret data.

Remote User-Authentication. More interestingly, in many cases, the fingerprint API is used as one part of an authentication scheme. Upon first usage, apps have to provide a single-factor or multi-factor user authentication system, since no cryptographic key is created and stored by the app inside the TEE yet. On subsequent usages, the app (and the corresponding backend) may require the user to touch the fingerprint sensor. Some apps can be configured to require the user to touch the sensor every time the app is opened and it connects to the remote backend. Others ask for this action before performing any sensitive operation, such as a payment.

Typically, when the fingerprint functionality is enabled, the app will allow the use of a fingerprint touch instead of inserting the account's password. While this is convenient in term of usability, it has mixed security consequences. As a security benefit, an attacker achieving "root" cannot steal the account password, since the user is not asked to insert it. However, as we will explain in Section VI-B, even a non-root attacker can potentially lure a user to touch the fingerprint sensor and, compared to phishing a password, stealing a fingerprint touch is significantly easier. In fact, touching the fingerprint sensor is a common action, since it is used, for instance, very frequently to unlock the phone. Therefore an attacker can just pretend to be the lock-screen without raising much suspicion. Secondly, a fingerprint touch requires less user's effort and time to be performed and therefore is more likely to happen. Finally, an attacker does not need to ask for a *specific* password, but just to generically touch the sensor.

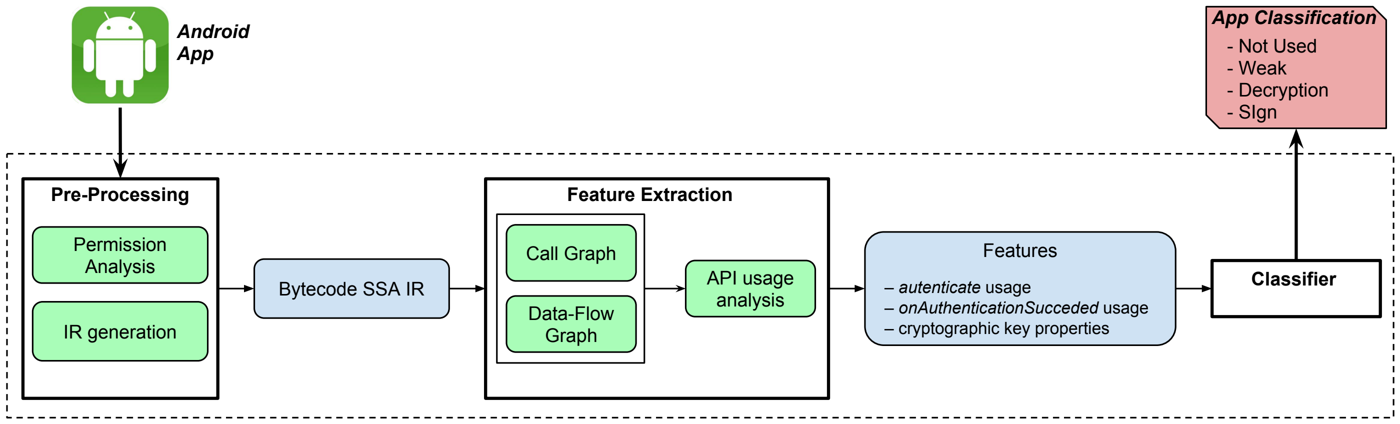

Fig. 1. Overview of the developed static analysis tool

B. *Practicality and Impact of UI Attacks*

As we mentioned earlier, a malicious app can show maliciously crafted messages or, more in general, interfere with the device's user interface to lure a legitimate user to touch the fingerprint reader sensor. In particular, we mentioned how several existing works [34], [14], [9] show the possibility to perform UI attacks, and that a very recent work, dubbed Cloak & Dagger [21], can achieve almost complete compromise of the device. In particular, this last work showed that apps installed from the Play Store are automatically granted the `SYSTEM_ALERT_WINDOW` permission (which allows to create overlays windows on top of any other) and that it is possible to lure the user to unknowingly grant accessibility permissions to a malicious app through "clickjacking."

These attacks are powerful, especially because they can be performed by any unprivileged app (what we refer to as "non-root attacker"). However, we note that the fingerprint API might be one of the few aspects that could, at least in principle, prevent full compromise: a physical fingerprint "touch" cannot be spoof via UI-only attacks.

That being said, there are many attacks that one could perform. These attacks are all instances of a confused deputy problem, and they are all possible due to one key observation: no "Secure UI" is currently used by the fingerprint API, and the user does not have any mechanism to establish with which app she is interacting with. As a very practical example of these attacks, Zhang et al. [53] show how an attacker can create a fake "screen lock" to lure the user to provide her fingerprint: the fingerprint, under the hood, is actually "passed" to a security sensitive app in the background.

More in general, the lack of "secure UI" allows an attacker (independently from the fingerprint usage scenarios described in Section IV) to lure the user to present her fingerprint believing she is authenticating with app A or authorizing transaction X, while the fingerprint is actually used to unlock keys for a different app B or to authorize transaction Y.

These attacks are affected by practicality aspects. First of all, an attacker needs to solve two issues:

1) Put the victim app in a state in which, once the fingerprint sensor is touched, an unwanted malicious action happens.
2) Lure a legitimate user to touch the sensor.

Second, the attacker needs to steal a fingerprint touch every single time she wants to perform the attack. However, this last challenge can be easily addressed: since the fingerprint is often used to perform "screen unlock" and since the "screen unlock" action is an action that a user is used to perform tens of times every day, it is straightforward for an app to create a situation for which the user would provide a fingerprint.

From a technical standpoint, an attacker can exploit this by simulating that a device got automatically locked (which, by default, happens after a few seconds of non-usage). To achieve this, the attacker can show a fullscreen, black overlay on top of any existing Activity.[3] Moreover, by requiring the permission `WRITE_SETTINGS`, the attacker can also minimize the background light of the screen. At this point, the attacker can prevent the device from automatically locking itself (by using the `WakeLock` API, requiring the automatically-granted `WAKE_LOCK` permission). In this scenario, a user will likely assume that the device got automatically locked and try to unlock it by touching the fingerprint sensor.

As an attempt to defeat these UI attacks, a countermeasure is currently implemented by the Android framework. Specifically, an app can only request the usage of the fingerprint sensor if it is displayed in the foreground. Unfortunately, in evaluating if an app is in the foreground, the Android framework only evaluates its position in the Activity stack. Since the Android framework does not deem screen overlays as part of the Activity stack, an Activity will still be considered as in foreground, even when maliciously covered by an overlay.

VII. Automatic Analysis Tool

We have developed a tool to automatically analyze how an app uses the fingerprint API. The tool takes an Android app as input and classifies its usage of the fingerprint API into *Weak*, *Decryption*, and *Sign* usage, as defined in Section IV. We use the tool above to perform the first systematic study on how Android applications use the fingerprint sensor, pinpointing cases in which this API is incorrectly used. We believe app developers and app market operators can also use this tool to automatically understand if there is any issue in how an app uses the fingerprint API. Figure 1 provides an overview of the developed tool.

[3]An Activity is the standard "unit of interaction" in Android and loosely corresponds to a window in a desktop environment.

A. Challenges and Design Choices

Our tool performs static analysis on an app's bytecode. We choose static analysis on bytecode to be able to perform our analysis without needing source code (which is typically unavailable both to security researchers and market operators). Moreover, many apps using the fingerprint API belong to the "finance" category. This makes very difficult to automatically perform dynamic analysis on these apps, since we do not have the required financial account information needed to get past the login stage. Even approaches able to automatically register accounts while performing dynamic analysis, such as AppsPlayground [39], cannot solve this problem by automatically creating bank (or other financially related) accounts. This aspect also significantly complicates our manual investigation of the results and our attempts to dynamically execute a given app.

One of the main challenges when analyzing recent real-world Android apps is the amount of code these applications include (on average, the apps we have analyzed have about 51,000 methods). This is often because apps include big libraries, which, even if only marginally used, substantially increase the amount of code a static-analysis tool may end up analyzing. Empirically, recent research [7] has shown that even relatively easy data-flow analysis, such as flow-insensitive taint analysis, often ends up using unpractical amounts of resources and time, when applied to an entire app. However, for the analysis we are interested in, we only need to precisely characterize the usage of very specific API methods. For these reasons, we adopted a more localized approach, which constructs call graph and data-flow graphs starting from the APIs of interest, limited to the specific parameters we are interested in.

B. Pre-processing

The first step of our analysis is to determine which apps potentially use the fingerprint API. Since, to use the fingerprint hardware, an app has to require the `USE_FINGERPRINT` permission, our tool first checks whether a given app requires this permission by reading its manifest file. Apps not requesting this permission cannot use the fingerprint API.

After this step, we use the Java static analysis framework SOOT [47] to obtain an intermediate representation of the app's bytecode. To simplify further data-flow analysis, we choose the *Shimple* intermediate representation, which is in single static assignment (SSA) form.

C. Call Graph Construction & Data Flow Analysis

Our analysis is based on two static analysis primitives: call graph generation and data-flow graph analysis. The call graph represents method invocations among different methods in the analyzed app. In building the call graph, we perform intra-procedural type-inference [36] to determine the possible dynamic types of the object on which a method is called. If this analysis fails, we over-approximate the possible dynamic types as all the subclasses of its static type (including the static type itself).

Our call graph also considers some implicit control flow transitions introduced by the Android framework [12]. In particular, when the `onAuthenticationSucceeded` callback is invoked by the Android framework, typically developers call the `postDelayed` method, by passing, as parameter, an instance of a specific inner-class, implementing the `Runnable` interface. On this inner-class, the method `run` will be later called and executed in a different thread. This is a common behavior in Android, since code dealing with UI elements has to run in a different thread than code dealing with network operations, to ensure app's responsiveness.

TABLE II. OVERVIEW OF THE COLLECTED FEATURES

`authenticate`	*Null/NonNull*
`onAuthenticationSucceeded`	*NoCrypto/Constant/ Decrypt/Signature*
Key Properties	*DecryptionKey/SigningKey UnlockedKey/LockedKey*

Our tool handles these cases by identifying the possible dynamic types of the instance passed to the `postDelayed` method. Then, it adds edges in the call graph between the `postDelayed` method and the implementations of the `run` methods that can be possibly called, according to the identified types (typically, just one).

To perform data-flow analysis, starting from a variable of interest V (e.g., a specific parameter of an API call), we recursively follow the def-use chain to obtain an *inter-procedural* backward slice. Moreover, when a field access is encountered, we continue the analysis starting from all the instructions accessing it. As an output of this analysis, we obtain a slice of instructions (encoded as a tree) in which each instruction uses variables that may influence the value of V.

D. Feature Extraction

At a high-level, our analysis extracts three kinds of features:

1) how the `authenticate` API is used;
2) which code is triggered when the `onAuthenticationSucceeded` callback is called;
3) the parameters used to create cryptographic keys.

Table II enumerates the features we extract to characterize these three aspects.

***authenticate* API Usage.** For the `authenticate` API, for each occurrence of a call to this method, our analysis generates a backward slice, starting from the parameter named `crypto`. This parameter is used to specify the cryptographic key that is "unlocked" whenever a legitimate user touches the fingerprint sensor. Then, by analyzing the generated slice, we check if the value of this parameter is `NULL`. In this case, it means that the `authenticate` API will activate the fingerprint sensor, but no key will be unlocked when the user touches it. We mark this case as *Null*, otherwise we mark it as *NonNull*.

***onAuthenticationSucceeded* Callback Usage.** We analyze the code that is executed when the `onAuthenticationSucceeded` callback is invoked, to determine if and how cryptographic operations happen after the user touched the fingerprint sensor. Starting from each occurrence of a method overriding `onAuthenticationSucceeded`, we start a forward exploration of the call graph, looking for calls to specific cryptographic methods.

Specifically, if we encounter a call to the methods `sign` or `update` of the class `Signature`, we mark this usage of `onAuthenticationSucceeded` as *Signature*, whereas if we encounter a call to the methods `doFinal` or `update` of the class `Cipher`, we mark it as *Decrypt*.

As a special case, if after the `onAuthenticationSucceeded` callback a decryption operation is detected, but it is performed on a fixed, hardcoded string (as explained in Section IV-B), we mark this case as *Constant* (instead of *Decrypt*). To determine this, we generate a backward slice starting from the parameter specifying the decrypted content, and we analyze it to determine if it results in a constant string.

In case we do not encounter any of the aforementioned cryptographic methods we mark the usage of the `onAuthenticationSucceeded` callback as *NoCrypto*, since it shows that no cryptographic operation is performed as a consequence of the user touching the fingerprint sensor.

Cryptographic Key Properties. To determine the type of the used cryptographic keys, we generate a backward slice starting from the `purpose` parameter of the `KeyGenParameterSpec.Builder` constructor. In case we determine it to have the value `PURPOSE_SIGN` we mark the key as a *SigningKey* otherwise we mark it as a *DecryptionKey*.

We also verify if the `setUserAuthenticationRequired` method is invoked (by passing `true` for its `required` parameter). If this is the case, we mark the key as *Locked*, otherwise, we mark it as *Unlocked*.

Other Features. To integrate the information collected by the features just described, we also check if an app is using the `getCertificateChain` and `setAttestationChallenge` APIs. While we do not use this information to classify how an app uses the fingerprint API, we will use this information to study if apps use key attestation (see Section V-D and Section VIII-G).

E. App Classification

After collecting the aforementioned features, we use them to classify how the analyzed app uses the fingerprint API. The rationale behind this classification rules is first to identify cases in which the fingerprint API is not used (e.g., no fingerprint-related API is called) or used in a *Weak* way (e.g., no cryptographic operation is performed). Then, we analyze the properties of the used cryptographic keys and the cryptographic methods called to determine whether to classify the app as *Decryption* or *Sign*.

First of all, we note that for some of the analyzed apps that request the `USE_FINGERPRINT` permission, we cannot identify any usage of the `authenticate` API or the `onAuthenticationSucceeded` callback. We classify these apps, together with those not requesting the `USE_FINGERPRINT` permission, as "Not Used."

Then, we classify an app as *Weak* if any of the following conditions are met:

1) We do not detect any key generation (i.e., the `KeyGenParameterSpec.Builder` API is never used).
2) All the usages of the `authenticate` API are marked as *Null*. This corresponds to the case in which no cryptographic key is unlocked as a consequence of the user touching the fingerprint sensor.
3) All the usages of the `onAuthenticationSucceeded` callback are marked as *NoCrypto* or *Constant*. This corresponds to the case in which no cryptographic operation is performed after the user touched the sensor (or the only cryptographic operation happening is performed on a constant value).
4) An *Unlocked* key is used. In fact, in this case, the used key is not locked, and any root attacker can immediately use it, without having the user touching the fingerprint sensor.

TABLE III. STATIC ANALYSIS TOOL RESULTS SUMMARY

Total Apps	Analysis Errors	Not Used
501	5 (1.00%)	72 (14.37%)

Category	*Weak*	*Decryption*	*Sign*
Detected apps	269 (53.69%)	146 (29.14%)	9 (1.80%)
Misclassifications	0/20	1/10	1/9

At this point, we know that some proper cryptographic operation happens after the user touches the fingerprint sensor. To determine whether the app uses the fingerprint API in a *Decryption* or in a *Sign* way, we use the following rule. We classify an app as *Sign* if any key marked as *SigningKey* is generated and any usage of the `onAuthenticationSucceeded` callback is marked as *Signature*. Otherwise, we classify the app as *Decryption*.

VIII. AUTOMATIC ANALYSIS RESULTS

A. Evaluation Methodology

To determine the correctness of the classification of our tool, we employed the following two-step methodology:

1) Driving the App to Ask for Fingerprint: In the first step of our evaluation, we manually drive the analyzed app to the point where it starts communicating with the TEE for fingerprint-based authentication.

One significant challenge in this step is that most of the considered apps require specific accounts to go beyond the initial login interface, and it is impractical to create accounts for many such apps. This is because many of the apps we analyzed are mobile-banking apps, for which it is not possible having an account without also being customers of the connected bank. In other cases, the app's backend requires financial information such as Social Security Numbers or debit card numbers to create an account, which further hindered our ability to interact with these apps.

2) Verify the Existence of Expected Weaknesses: Once we drive the analyzed app to start interacting with the TEE, we verify our tool's classification for this app by simulating a root attacker and see if the fingerprint-based authentication is vulnerable to weaknesses of the corresponding class as predicted in Section V. For simulating a root attacker, we used the Xposed Framework [1], a tool which allows us to easily modify apps' and framework's Java code at runtime.

In particular, if our tool classifies the app as using the *Weak* usage, our simulated attack modifies the behavior of the `authenticate` API to directly call the `onAuthentication-Succeeded` callback. Furthermore, we deal with the case in which the victim app invokes any cryptographic operation using a key stored inside the TEE. In this case, the app would raise an exception, since this key has not been "unlocked." This scenario may occur in the case in which the result of the decryption is not used (and therefore we classify the app as *Weak*), but still, a TEE-protected key is used to decrypt a hardcoded string, as it happens, for instance, in the Google's sample code [25]. We deal with this case, by masking the generated "User Not Authenticated" exception.

For apps classified as using the fingerprint API in a *Decryption* way, we first record the outputs of decryption operations using TEE-protected keys (simulating a Root attacker). Then, we modify the `authenticate` API as explained before and, additionally, we replay the collected decryption outputs when necessary.

B. Dataset

We collected all the free apps classified as "Top" (i.e., most popular) in each category of the Google Play Store. These apps were downloaded in February 2017. Additionally, we added apps preinstalled on a Nexus 5X device running Android 7. In total, we created a dataset of 30,459 apps. Among these apps, 501 (1.64%) declare the `USE_FINGERPRINT` permission and, therefore, can potentially use the fingerprint API. In the rest of this section, we will focus on this subset of 501 apps.

C. Apps Classification

Table III summarizes the outputs of our tool. We ran our tool in a private cloud, and for the analysis of each app we provided 4 virtual-cores, 16 GB of RAM and 1 hour time limit. For the 501 apps, our tool needed on average 354 seconds ($\sigma = 363$) of computation and used 6.13 GB ($\sigma = 1.07$) of RAM per app. In 5 cases (1.00%), our analysis did not finished due to bugs in the SOOT framework or analysis timeout.

For 72 (14.37%) apps, although they ask for the `USE_FINGERPRINT` permission, our tool did not detect any usage of the fingerprint API. This result is not particularly surprising since previous research has shown that apps tend to require more permissions than they use [48]. To further verify this finding, we manually analyzed a random sample of 10 of these apps. We both manually run them and perform tool-assisted reverse engineering. For 7 of them, we could confirm that they do not use the fingerprint API, whereas for the other 3 our tool was unable to detect its usage because these apps use native code components to activate the fingerprint reader sensor, which our tool is unable to analyze.

For apps classified as *Weak* we took a random sample of 20 apps among those in which we were able to dynamically reach the fingerprint interface. Our dynamic analysis confirmed that they were all correctly classified (i.e., our simulated attack in Section VIII-A2 is successful). Among these 20 apps, 16 access a remote account or store secret data, therefore a *Weak* usage of the fingerprint API is not appropriate (as explained in Section VI-A).

For apps classified as *Decryption* we took a random sample of 10 apps and we confirmed that 9 were correctly classified (using, again, the simulated attack explained in Section VIII-A2), whereas 1 was classified as *Decryption* while in reality is *Weak*.

Finally, about the 9 apps classified as *Sign*, we were able to dynamically reach the fingerprint interface in one app and dynamic analysis confirmed the classification of this app as correct. This app, called "True Key," requires to *sign* an authentication token during login and performs this operation with a TEE-protected private key, "unlocked" only when the user touches the fingerprint reader sensor. To have a better evaluation, we also extensively reverse engineer the other 8 samples classified as *Sign*. Our manual analysis revealed that 7 of them have been classified correctly, whereas 1 has been classified as *Sign* while being *Decryption*.

In summary, we manually analyzed (either by reproducing our attacks as explained in Section VIII-A2 or by reverse engineering) 39 apps and we found that all the apps except 2 were classified correctly. In one case the misclassification is due to overapproximations in the call graph. In the other, the app "signs" some data, but this data is constant, since it is provided by the backend when the user logins the first time. For this reason, the app falls into the *Decryption* category. In fact, an attacker can trivially replay the result of this signing operation after it happened once. However, our tool was unable to detect this scenario and, therefore, it classified the app as *Sign*.

Overall, results show how our tool is reasonably accurate in determining how an app uses the fingerprint API. Moreover, the few misclassifications "overestimate" the security of an app (classifying it as using the fingerprint API in a stronger way than in reality). Therefore, we believe that our results, showing a low usage of the fingerprint API in the *Sign* way and a high usage in the *Weak* way, are particularly worrisome and confirm our intuition that apps generally do not use appropriately the fingerprint API. In the next sections, we will provide concrete examples of these inappropriate usages.

D. Case Study: Unlocking "Unlocked" Keys

As explained in Section II-C, a key is stored inside the TEE and "unlocked" by a fingerprint touch only if the `setUserAuthenti-cationRequired` method is invoked (by passing `true` for its `required` parameter) when the key is generated. On the contrary, without calling this method, a generated key is always "unlocked," regardless of the usage of the fingerprint API.

Surprisingly, we found this aspect as a source of implementation errors. In particular, we looked for apps implementing proper cryptographic operations as a consequence of the user touching the fingerprint sensor (i.e., calling the `authenticate` API to "unlock" a key used to decrypt or sign some data), but not calling properly the `setUserAuthenticationRequired` method. This indicates that the developers wanted to have a key "unlocked" when the legitimate user touches the fingerprint sensor, but forgot to "lock" the key in the first place.

To identify these apps, we checked for apps that

1) are classified as *Weak* by our tool;
2) do not call the `setUserAuthenticationRequired` method (or they call it specifying `false` as its parameter);
3) if they had called the `setUserAuthenticationRe-quired` method properly they would have been classified as *Decryption* or *Sign*.

Our tool identified 15 apps in this scenario and we were able to fully dynamically interact with 4 of them, verifying their improper usage of the fingerprint API.

As an example, one of these applications allows a user to purchase items in an online marketplace and requires the user to touch the fingerprint sensor during the checkout procedure. The user's password is stored encrypted by a supposedly TEE-secured key, as is common when the fingerprint API is used in a *Decryption* way. During the checkout, when the user touches the fingerprint sensor, this key is used to decrypt the user's password. However, we verified that the decryption key is not really "locked" since the `setUserAuthenticationRequired` method is not called. Therefore, from a cryptographic perspective, the use of the fingerprint API is useless. As a consequence, a root attacker can easily bypass its usage.

E. Case Study: Google Play Store

Among the apps our tool classified as *Weak*, one is the "Google Play Store" app. This app is present on every Google-branded phone, and it handles the purchase of apps, media, and in-app purchases and can be setup to "protect" these purchases by a fingerprint touch. In this case, the user would be required to touch the sensor before every purchase. Since this app can directly spend user's money and interacts with a remote server, the most appropriate usage of the fingerprint API would be *Sign*, as also stated and exemplified in the guidelines from Google itself.

However, our tool classified the Google Play Store app as using the fingerprint API in a *Weak* way and our evaluation (as described in Section VIII-A) confirmed this result. In fact, this app calls the `authenticate` API with a `NULL` value for its `crypto` parameter, and, therefore, no key is "unlocked" and no *sign* operation certifies that the purchase happened as a consequence of the user touching the fingerprint reader sensor.

On July 2017, we contacted the Android's security team. The team promptly replied and forwarded our report to the Google Play's team, which is now aware of the issue and investigating it.

F. Case Study: Square Cash

Among the apps our tool classified as *Decryption*, one is the "Square Cash" app. This app is a personal payment app, which allows users to transfer money to and from connected debit cards and bank accounts.

The app can be configured to require the user to touch the fingerprint sensor before any transaction. The most appropriate usage of the fingerprint API in this case would be to use it to *sign* these transactions. However, Whorlwind, the open source library that Square (and other apps in our dataset) uses to implement the fingerprint functionality, implements a weaker scheme. In particular, this library is used to decrypt a locally stored authentication token. For this reason, by simulating an attacker with Root capabilities, we were able to reuse the same decrypted token to perform different payments.

We contacted the developers of the Whorlwind library in August 2017, detailing our findings and why we think that a *Sign* usage of the fingerprint API is more appropriate in this case.

G. Case Study: Key Attestation

We mentioned in Section V-D that, starting from Android 7, a new mechanism has been implemented to allow developers to "attest" public keys, ensuring they have been generated from "trusted" TEEs. According to the API, a properly verified certificate chain, "rooted at a trustworthy CA key," is only provided if the `setAttestationChallenge` API, with a non-NULL value for `attestationChallenge`, is called.

Conceptually, apps using both the fingerprint API in a *Sign* way and key attestation should be categorized in a different group in Table III. However, in our dataset, our tool found no app calling this API. This indicates that every app in our dataset is vulnerable to a Root-at-Bootstrap attacker, who can interfere with the initial key exchange process between the app and its remote backend.

IX. Fingerprint API Improvements

We will now propose some changes to the current fingerprint API, which would significantly improve its security. In this section, we will assume that apps use the fingerprint API in a *Sign* way, which, as previously shown in Section V, it is the right way to provide stronger security. However, even with proper usage, this API currently has some shortcomings, which we will address here.

A. Trusted-UI

The biggest limitations of the current API and its implementation are:

1) Users have no *trusted* way to understand what they are *signing* by touching the fingerprint sensor.
2) A malicious application (with or without "root" privileges) can interfere with what is shown to the user when asked to touch the sensor.

To solve both issues, we propose a mechanism in which the TEE can directly show to the user the content of a *sign* operation performed by a fingerprint-unlocked key. This mechanism is based on the known idea of having a *trusted* video path directly between the TEE and the device's screen. TEE-enforced video paths are already implemented in some Android devices (for DRM purposes) [49] and academia explored its use for authentication purposes [45]. However, differently from previous solutions, what we propose is also based on a *trusted* input which is the fingerprint reader sensor, able to directly communicate with the TEE.

We propose to change the current `authenticate` method to also take as an input a `message` string parameter, for instance "Do you want to authorize a payment of \$1,000 to $Friend$?" This message would be shown on a TEE-enforced Secure UI dialog window, alongside with a standardized graphic UI asking the user to touch the fingerprint sensor. *Untrusted* code, outside the TEE, cannot interfere with the visualization of this window, due to the usage of a secure video path. Specifically, *untrusted* code cannot read the content of this dialog window nor modify it.

When the sensor is touched by a legitimate user, a signature of this string (generated using the private key "unlocked," specified when the `authenticate` method is called) is available using a method called `getSignedMessage`. The remote backend can then verify that this message has been signed correctly and, therefore, be sure of what the user has authorized by touching the sensor. In other words, the remote backend can verify the "user intention," which is signed by the TEE.

The security of this system is guaranteed by the fact that both the code for handling the *sign* operation and the code for visualizing the message are within the TEE. Therefore, an attacker, even having root privileges, cannot decouple what is being shown to the user with what is being signed by the fingerprint-unlocked key. An attacker can still interfere with the communication between the backend, the app, and the TEE. However, this will be detectable by the user. In fact, suppose that the attacker changes the request the app sends to the backend from "Pay $Friend$ \$1,000" to "Pay $Attacker$ \$1,000." As a consequence the backend will send the following message to be signed by the TEE: "Do you want to authorize a payment of \$1,000 to $Attacker$?". In this case, the user will be able to notice that the message does not correspond to her intention.

Another issue is how to prevent an attacker from showing a malicious dialog window that resembles the window shown by the TEE when asking the user to touch the fingerprint sensor. Without requiring extra hardware (e.g., an LED would be turned on when "secure output" is displayed), we can exploit the fingerprint sensor itself to mitigate this attack. Since the fingerprint sensor can communicate directly and exclusively with the TEE, we propose that the TEE shows a hard-to-spoof visual clue (e.g., a loading bar) while the user touches the sensor.

Attackers would be unable to show this bar at the right time, since, outside the TEE, it is unknown when the user touched the sensor. Therefore, the absence (or the improper behavior) of this visual element would indicate to the user that the shown dialog window is not legitimate. Another possible solution, although less practical since it requires a setup phase, would be to use a secret (i.e., only know by the user and the TEE) personalized security indicator. This mechanism has been shown as an effective defensive mechanism in the Android ecosystem [9].

It is important to notice, however, that even without this defense, an attacker would not be able to lure users to *sign* a malicious transaction, but only to pretend that a transaction happened.

B. Other UI Changes

While a solution based on hardware-enforced secure-UI is the best way to address current API shortcomings, we understand that its adoption and deployment may be problematic because it requires non-trivial modifications to the code running inside the TEE and the coordination between this code, the Android operating system, and the display hardware. Therefore, we also propose easier-to-implement modifications to the current Android user-level framework. While attackers having "root" privileges can trivially bypass these mechanisms, they are still effective against a non-root attacker.

In particular, Android should automatically dismiss overlay windows on top of interfaces asking the user to touch the fingerprint sensor. A similar solution is already applied in the latest Android versions to protect "security sensitive" interfaces, such as the one used to grant/remove apps' permissions. In addition, the name (and the icon) of the app asking the user's touch should be clearly shown. To implement both solutions, a standard interface, which apps cannot modify except showing some text on it, should be shown when the `authenticate` API is called. In the current implementation, custom interfaces are possible, but uncommon. In fact, most of the apps show very similar interfaces (Android guidelines precisely define how this dialog should appear [29]), thus they will not need to significantly change their UI.

C. Better Attestation Mechanisms

As we previously mentioned, a key attestation mechanism has been implemented, starting from Android 7. However, in its current implementation state, this mechanism has several weaknesses.

First of all, the API defines two possible "levels" for the attestation "software" and "hardware," where only the latter guarantees that a key has been generated by the device's TEE. The level of attestation can be retrieved by parsing the attestation certificate associated with a generated public key. However, in the devices we have tried (Nexus 5X and Pixel XL, running Android 7), the generated keys are always "software" attested.

More fundamentally, while analyzing the generated certificates, we did not find any indication of the specific instance of the device generating a key. As also pointed out by the paper presenting the Security Key protocol [32], there is a trade-off between user's privacy and security of the protocol. Having a system that can identify the specific device generating a key would allow remote backends to detect suspicious situations in which the key associated with a specific user changes. Moreover, it would hinder the ability of an attacker to "proxy" key creation to an attacker-controlled TEE, since too many keys (used by many different users) generated by the same device would be easily detected as suspicious. However, this would violate user's privacy, allowing unique user's identification among different apps. Therefore, we recommend, as in the Security Key protocol, the implementation of a batch attestation scheme, in which a set of devices, using the same hardware (and potentially affected by the same security issues), shares the same attestation key.

Finally, we note that the current documentation about how to verify key attestation certificates is insufficient and the only official sample code [24] does not cover all the possible cases that need to be handled while parsing this type of certificates.

X. Limitations and Future Work

This paper focuses on the most common fingerprint API in Android, used by Google's devices. However, Samsung's and Huawei's devices offer their custom fingerprint hardware and a different API. Moreover, outside of the Android ecosystem, similar systems are offered on Apple's devices [3], [4]. Studying similarities and differences among these APIs and how apps that want to be compatible with multiple devices handle this fragmentation is the main future direction of this work.

Our static analysis is based on call graph generation and data-flow graph analysis. This approach has been proved effective by previous research [17] in determining how specific APIs are used in Android. However, this approach is unable to analyze reflective code, dynamically loaded, or native components. Regarding the first two aspects, we do not expect them to be a significant source of imprecision when analyzing non-malicious code and we did not find any sample misclassified because of these reasons. We consider the analysis of components written in native code outside the scope of this paper. Empirically, we found that the usage of native code prevented us from analyzing three apps (among those manually verified), as explained in Section VIII-C.

More fundamentally, the implemented static analysis can indicate the way in which an app uses *locally* the fingerprint API, but it cannot fully evaluate how this aspect affects the overall authentication mechanism implemented by the app and its backend. This analysis usually requires probing the remote backend (when, as it is typically, the backend's code is not available) to determine if it properly checks user's authentication. Merging our tool with more general remote protocol analyzers (as, for instance, the one proposed by Zuo et al. [55]) represents another interesting future direction.

XI. Related Work

Zhang et al. [53] show how UI attacks were extremely easy in Samsung devices (Samsung Galaxy S5 and S6) running Android 5. Specifically, these attacks were possible because, after a legitimate user touched the fingerprint reader sensor, a malicious app could use fingerprint-protected cryptographic keys generated by other apps. In addition, they show how in some devices it was possible to steal raw fingerprint information. In our work, we focus primarily on the newer Google's fingerprint API (released with Android 6), in which each app utilizes app-specific keys.

A few other works focus on aspects concerning the security of the fingerprint hardware sensor and the storage of fingerprint data [40], [16]. In our work we expand and generalize findings of these previous works and, in addition, we systematically study how apps use the fingerprint API and how this aspect affects the overall security of an app's authentication scheme.

There is a number of works related to two-factor authentication and authentication in mobile systems. Lang et al. [32] describe "Security Keys," second-factor devices that protect users against phishing and man-in-the-middle attacks. They also discuss the deployment of this technology to the Chrome browser and Google online services. Throughout our paper, we show how the fingerprint API could potentially offer the same or better security properties, but some shortcomings in its implementation and in how apps use it prevent this from happening.

One-Time Passwords (OTPs) are a (weaker) alternative to Security Keys. TrustOTP [45] shows how smartphones can act as secure OTP tokens. Dmitrienko et al. [15] highlight weaknesses in the design and adoption of two-factor authentication protocols and mechanisms. In particular, they show how an attacker can mount cross-device attacks and bypass 2FA mechanisms such as SMS-based TANs (Transaction Authentication Numbers) used by banks, or login verification systems such as Google, Twitter, and Facebook. Chen et al. [13] discuss different OAuth implementations and their adoption by mobile applications.

Several works focus on the automatic detection of classes of vulnerabilities in Android apps. Previous work focused on detecting over-privileged apps [6], component-hijacking vulnerabilities [33], vulnerable content providers [54], permission leaking [31], and vulnerabilities related to the unsafe usage of crypto-related APIs [17], [55], SSL connections [18], and dynamic code loading [38]. Other recent works show how some apps implement vulnerable custom authentication schemes (trying to minimize users' effort during login) [10] and how apps often use payment libraries insecurely [50]. Acar et al. provide an overview of the different security mechanisms implemented in Android and the improvements suggested by academia [2].

Other relevant works are those focusing on highlighting GUI-related vulnerabilities and problems. For example, Felt et al. [19], Niemietz et al. [34], Chen et al. [14], Bianchi et al. [9], and Ren et al. [41] study the use of UI attacks to lure users to enter their credentials into malicious UIs. Moreover, a recent work shows how redressing attacks can even lead to complete compromise of the device UI [21], since an app can use these attacks to stealthy obtain "accessibility" permission and take full control of the user's input and the display's output.

XII. Conclusions

This work provides the first systematic study on the usage of the fingerprint API in Android. We show that its usage is not well understood and often misused by apps' developers. In particular, our study shows that several apps, including popular ones such as Google Play Store and Square Cash, do not use this API in the most secure way. We believe that the fingerprint API could significantly improve the security and the usability of existing authentication and authorizations schemes, especially because the hardware it needs is commonly available in modern mobile devices. We hope this paper will highlight current weaknesses and push Google to provide better documentation and to address the remaining problematic issues.

Acknowledgment

We would like to thank Martina Lindorfer for providing us the dataset we used in this paper and Carter Yagemann for the insightful discussion about user authentication.

This research was supported by the ONR under grants N0001409-1-1042, N00014-15-1-2162 and N00014-17-1-2895, by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP-006, by the NSF under grant CNS-1408632, and by the Google's "Security, Privacy and Anti-Abuse Award." Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ONR, DARPA, NSF, or Google.

References

[1] "Xposed Installer," http://repo.xposed.info/module/de.robv.android.xposed.installer, 2017.

[2] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "SoK: Lessons Learned From Android Security ResearchFor Appified Software Platforms," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[3] Apple, "About Touch ID advanced security technology," https://support.apple.com/en-us/HT204587, 2015.

[4] ——, "Get your apps ready for Touch Bar." https://developer.apple.com/macos/touch-bar/, 2017.

[5] ARM, "ARM TrustZone," https://www.arm.com/products/security-on-arm/trustzone, 2017.

[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[7] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining Apps for Abnormal Usage of Sensitive Data," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[8] A. Bianchi, "Source Code of the Developed Static Analysis Tool," https://github.com/ucsb-seclab/android_broken_fingers, 2018.

[9] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, “What the App is That? Deception and Countermeasures in the Android User Interface,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[10] A. Bianchi, E. Gustafson, Y. Fratantonio, C. Kruegel, and G. Vigna, “Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2017.

[11] Board of Governors of the Federal Reserve System, “Consumers and Mobile Financial Services 2016,” https://www.federalreserve.gov/econresdata/consumers-and-mobile-financial-services-report-201603.pdf, 2016.

[12] 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 *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

[13] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, “OAuth Demystified for Mobile Application Developers,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[14] Q. A. Chen, Z. Qian, and Z. M. Mao, “Peeking Into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks,” in *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2014.

[15] A. Dmitrienko, C. Liebchen, C. Rossow, and A. Sadeghi, “On the (In)Security of Mobile Two-Factor Authentication,” in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[16] T. Does and M. Maarse, “Subverting Android 6.0 fingerprint authentication,” Master Thesis at University of Amsterdam, 2016.

[17] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An Empirical Study of Cryptographic Misuse in Android Applications,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[18] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, “Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[19] A. P. Felt and D. Wagner, “Phishing on Mobile Devices,” in *Proceedings of the IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*, 2011.

[20] FIDO Alliance, “What is FIDO?” https://fidoalliance.org/about/what-is-fido/, 2017.

[21] Y. Fratantonio, C. Qian, P. Chung, and W. Lee, “Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[22] D. Goodin, “Thieves drain 2fa-protected bank accounts by abusing SS7 routing protocol,” https://arstechnica.com/information-technology/2017/05/thieves-drain-2fa, 2017.

[23] Google, “New in Android Samples: Authenticating to remote servers using the Fingerprint API ,” https://android-developers.googleblog.com/2015/10/new-in-android-samples-authenticating.html, 2015.

[24] ——, “Android Key Attestation Sample,” https://github.com/googlesamples/android-key-attestation/tree/master/server, 2016.

[25] ——, “Android FingerprintDialog Sample,” https://github.com/googlesamples/android-FingerprintDialog, 2017.

[26] ——, “Android Keystore System,” https://developer.android.com/training/articles/keystore.html, 2017.

[27] ——, “Android Security Bulletins,” https://source.android.com/security/bulletin/, 2017.

[28] ——, “FingerprintManager,” https://developer.android.com/reference/android/hardware/fingerprint/FingerprintManager.html, 2017.

[29] ——, “Material Design – Patterns – Fingerprint,” https://material.io/guidelines/patterns/fingerprint.html, 2017.

[30] ——, “Verifying Boot,” https://source.android.com/security/verifiedboot/verified-boot, 2017.

[31] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, “Systematic Detection of Capability Leaks in Stock Android Smartphones,” in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2012.

[32] J. Lang, A. Czeskis, D. Balfanz, and M. Schilder, “Security Keys: Practical Cryptographic Second Factors for the Modern Web,” in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2016.

[33] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[34] M. Niemietz and J. Schwenk, “UI Redressing Attacks on Android Devices,” *Black Hat Abu Dhabi*, 2012.

[35] OWASP-MSTG, “Local Authentication on Android,” https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05f-Testing-Local-Authentication.md, 2017.

[36] J. Palsberg and M. I. Schwartzbach, “Object-Oriented Type Inference,” in *Proceedings the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1991.

[37] B. Parno, “Bootstrapping Trust in a “Trusted” Platform,” in *Proceedings of the USENIX Summit on Hot Topics in Security (HotSec)*, 2008.

[38] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications,” in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[39] V. Rastogi, Y. Chen, and W. Enck, “AppsPlayground: Automatic Security Analysis of Smartphone Applications,” in *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[40] M. Rehman Zafar and M. Ali Shah, “Fingerprint Authentication and Security Risks in Smart Devices,” in *Proceedings of the International Conference on Automation and Computing (ICAC)*, 2016.

[41] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, “Towards Discovering and Understanding Task Hijacking in Android,” in *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2015.

[42] A. Roy, N. Memon, and A. Ross, “MasterPrint: Exploring the Vulnerability of Partial Fingerprint-Based Authentication Systems,” *IEEE Transactions on Information Forensics and Security*, vol. 12(9), 2017.

[43] J. Scott-Railton and K. Kleemola, “London Calling – Two-Factor Authentication Phishing from Iran,” https://citizenlab.ca/2015/08/iran_two_factor_phishing/, 2015.

[44] J. V. Stoep, “Android: protecting the kernel,” https://events.linuxfoundation.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf, 2016.

[45] H. Sun, K. Sun, Y. Wang, and J. Jing, “TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[46] Telegram, “Keep Calm and Send Telegrams!” https://telegram.org/blog/15million-reuters, 2016.

[47] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot – a Java Bytecode Optimization Framework,” in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[48] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, “Permission Evolution in the Android Ecosystem,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[49] Widevine, “Widevine DRM Architecture Overview,” https://storage.googleapis.com/wvdocs/Widevine_DRM_Architecture_Overview.pdf, 2017.

[50] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, “Show Me the Money! Finding Flawed Implementations of Third-party In-app Payment in Android Apps,” in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

[51] yubico, “FIDO U2F,” https://www.yubico.com/solutions/fido-u2f/, 2017.

[52] yubico, “YubiKeys,” https://www.yubico.com/products/yubikey-hardware/, 2017.

[53] Y. Zhang, Z. Chen, and T. Wei, “Fingerprints On Mobile Devices: Abusing and Leaking,” in *Black Hat USA*, 2015.

[54] Y. Zhou and X. Jiang, “Detecting Passive Content Leaks and Pollution in Android Application,” in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2013.

[55] C. Zuo, W. Wang, R. Wang, and Z. Lin, “Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services,” in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2016.