VButton: Practical Attestation of User-driven Operations in Mobile Apps

Wenhao Li [1,2], Shiyu Luo [1,2], Zhichuang Sun [3], Yubin Xia [1,2], Long Lu [3], Haibo Chen [1,2], Binyu Zang [1], Haibing Guan [2]

1. Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
2. Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
3. Northeastern University

{liwenhao,luoshiyu,xiayubin,haibochen,byzang,hbguan}@sjtu.edu.cn, {z.sun,l.lu}@northeastern.edu

ABSTRACT

More and more malicious apps and mobile rootkits are found to perform sensitive operations on behalf of legitimate users without their awareness. Malware does so by either forging user inputs or tricking users into making unintended requests to online service providers. Such malware is hard to detect and generates large revenues for cybercriminals, which is often used for committing ad/click frauds, faking reviews/ratings, promoting people or business on social networks, etc.

We find that this class of malware is possible due to the lack of practical and robust means for service providers to verify the authenticity of *user-driven operations* (i.e., operations supposed to be performed, or explicitly confirmed, by a user). We design and build the VButton system to fill this void. Our system introduces a class of attestation-enabled app UI widgets (called VButton UI). Developers can easily integrate VButton UI in their apps to allow service providers to verify that a user-driven operation triggered by a VButton UI is indeed initiated and intended by a real user. Our system contains an on-device *Manager*, and a server-side *Verifier*. Leveraging ARM TrustZone, our system can attest operation authenticity even in the presence of a compromised OS. We have implemented the VButton system on an ARM development board as well as a commercial off-the-shelf smartphone. The evaluation results show that the system incurs negligible overhead.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**;

KEYWORDS

Mobile platform, TrustZone, User-driven security, Attestation

ACM Reference Format:
Wenhao Li [1,2], Shiyu Luo [1,2], Zhichuang Sun [3], Yubin Xia [1,2], Long Lu [3], Haibo Chen [1,2], Binyu Zang [1], Haibing Guan [2] . 2018. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proceedings of MobiSys'18*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3210240.3210330

MobiSys'18, June 10–15, 2018, Munich, Germany

ACM ISBN 978-1-4503-5720-3/18/06.
https://doi.org/10.1145/3210240.3210330

1 INTRODUCTION

Mobile malware, especially those controlling the OS, e.g., rootkits, can issue requests, or perform actions on users' behalf without users' awareness. In fact, *user-impersonating malware* provide an effective means for cyber criminals to commit various kinds of fraud, including fake ad clicks, ticket scalping, voter fraud, fake reviews, etc [17, 19, 32].

Leveraging legitimate user identities and hiding within real user devices, such malware and their forged requests are very difficult for service providers to detect. A recent article [5] reported a click-farm in China that employs more than 10,000 smartphones to automatically forge user activities, including rating products, following accounts and sending 'likes' on websites. It was reported that "Companies pay tens of thousands of pounds to get their products as many likes as possible." [5].

Motivated by the surge of user-impersonating malware, we aim at addressing its root cause— the lack of methods for online service providers to reliably verify the authenticity of *user-driven operations* originated from mobile devices. We identify two requirements that an "*authentic user-driven operation*" has to meet:

- **R1. User Initiation.** The operation is *initiated* by a human (e.g., via a button touch/click), as opposed to by an automatic agent, such as malware.
- **R2. User Intention.** The user intends to initiate the operation and understands its meaning, rather than being tricked.

In practice, **R2** is hard to ensure in a technical way. A more realistic description of **R2** could be: "The user is *shown* correct information to initiate the operation and *confirm* in an explicit way".

Due to the lack of methods for reliably verifying **R1** and **R2**, online service providers nowadays can only resort to a handful of techniques to *approximately infer* if **R1** and **R2** are met for a given operation request.

CAPTCHA is the most commonly-used technique to tell bots apart from human users. Despite its negative useability impact, which keeps growing as new attacks emerge [39], CAPTCHA only indicates human presence—it is not meant for directly proving user initiation or intention, e.g., a user can still be tricked into solving a CAPTCHA while not understanding the real operation to be performed. Alternatively, two factor authentication schemes such as one-time SMS code has become increasingly popular on mobile devices not only for user authentication but also for seeking user confirmation on certain operations. However, this method is ineffective against powerful user-impersonating malware which are

often rootkits and can easily read or intercept authentication codes sent via SMS, email, push notification, etc.

In this paper, we propose the VButton system which allows mobile app developers or service providers to attest the authenticity of a user-driven operation from an untrusted mobile device to verify if the operation is indeed initiated and intended by a human user (i.e., attesting **R1** and **R2**).

To app developers, VButton is a class of customizable UI widgets for users to initiate/perform sensitive operations that need to be attested by a remote party. We provide helper APIs and libraries which allow developers to use VButton in the same way as they use regular UI widgets. We design two particular types of VButton UI, resembling a `Button` and a `View`, that respectively correspond to two attestation modes: *Explicit Attestation* and *In-situ Attestation*. The Explicit Attestation mode is suitable for operations whose complete semantics need a larger and separate UI area to be displayed (e.g., a `View` that shows the content of an email before sending). The In-situ Attestation mode applies to semantically simple operations whose meaning can be fully conveyed by their trigger UI, e.g., a `Button` in a Twitter app that says "Follow User X". In both cases, the VButton UI and the operation semantics are generated and displayed in a secure fashion without relying on the (untrusted) app or the OS.

The underlying attestation mechanism is split into two parts which are provided by the two components of our system: the VButton Manager and the VButton Verifier. First, the VButton Manager is responsible for on-device system support. Running inside the ARM TrustZone, i.e., the TCB (Trusted Computing Base), the Manager renders the VButton UI, monitors relevant hardware events, e.g., screen touches within VButton UI regions, and generates and signs attestation blobs. Second, the VButton Verifier is the component deployed on server-side or verifier-side. It generates VButton UI upon request for either in-situ or explicit attestation. Such UIs are sent as signed images back to requesting apps, and then passed to the Manager for display. The Manager records any user action on VButton UI and sends the event, along with semantics and context data, to the Verifier for attestation.

A successful attestation confirms: (1) the operation is initiated by a human user, (2) the request comes from an enrolled device, and (3) the request matches captured user intention. As a result, both **R1** and **R2** are met. VButton does not place any trust on either the mobile operating system or applications. This is necessary to defend against rootkits, and reduces the TCB down to the code running in the TrustZone.

In designing and building the VButton system, we solved three major technical challenges: (1) defining generic, easy-to-use UI and system primitives for the proposed attestation; (2) designing the system mechanism without trusting the OS or apps while minimizing the trusted codebase; (3) limiting or avoiding additional user interaction.

In summary, this paper makes the following contributions:

- A method and its system-level mechanism, called VButton, that enables a server to attest the authenticity of *user-driven operations*.
- A design and implementation of VButton leveraging ARM TrustZone which achieves satisfactory performance in practice, and security against powerful malware, e.g., rookits and compromised OS.
- A set of well-defined and easy-to-use APIs for developing VButton-enabled applications as well as application servers.
- A case study where VButton is applied to popular apps to defend against real attacks.

The rest of this paper is organized as follows. Section 2 describes the motivation and our threat model. Section 3 describes two modes of our design: the explicit attestation mode and in-situ preview mode. Section 4 details the implementation of VButton. Section 5 and Section 6 present the evaluation of our system on both security and performance followed by a discussion on various issues in Section 7. We then present analysis of related work in Section 8. Finally, Section 9 concludes this paper.

2 MOTIVATION AND THREAT MODEL

2.1 Motivation

For online service providers, it is critical, yet currently impossible, to reliably verify if requests coming from (untrusted) mobile devices are authentic and not the result of malware or deceived users. Consequently, fraudulent requests often cause remarkable damage to service providers while bringing considerable revenue to attackers. For example, fraudsters, paid to promote given twitter accounts, employ malware to compromise mobile devices of legitimate users and stealthily follow promoted accounts on behalf of victim users. Such attacks led to increased questions like "I found myself following users I don't recognize?", to which Twitter responds "scan your devices for spyware/malware" [40].

There are generally two types of unauthentic requests: *forged requests* from bots and *unintended requests* from misled users, as listed in Table 1. Request-forging bots take full control of user devices and issue requests without users' permission or knowledge. When controlling user devices is impossible, attackers try to mislead or trick users into performing unintended actions, e.g., paying a wrong person, via UI hijacks or manipulation.

Facing the lack of effective and robust methods for verifying the authenticity of user-driven operations, service providers in practice resort to the following techniques to mitigate unauthentic requests by bots or misled users with limited success. Figure 1 shows a comparison of these methods in terms of their accuracy and user involvement.

CAPTCHA: Asking users to solve hard-for-computer yet easy-for-human problems, CAPTCHA aims to tell apart bots from humans. However, CAPTCHA is not reliable and often evaded. Recently, Google has cracked its own CAPTCHA mechanism [22] with 99.8% accuracy. Further, CAPTCHA is not designed to verify user initiation or intention, but user presence.

User Authentication: After receiving sensitive operation requests, e.g., changing passwords, placing orders, etc., some service providers require users to re-authenticate themselves via passwords or multi-factor means. However, repeated authentications or logins affect user experience, and, more importantly, offer little help with verifying users' intent or checking for the presence of compromised or deceptive client apps. Moreover, some of these

Table 1: Types of malicious requests.

Attack Types	Description	Possible Attacks	Mitigation in Practice
Malware (Bot)	The attacker gets full control over the victim's device, and operates on behalf of the device owner.	Unaware payment, following unknown twitter ID, Botnet for SPAM, etc.	PIN code, Fingerprint, CAPTCHA, Multi-factor authentication.
Misleading Users	The attacker gets partial control of the victim's device, e.g., can display some fake icon but cannot fake user input, and lures the owner to do unintended operations.	Payment cheating, Facebook "like" hijacking, Tweetbomb, Hidden UI element.	Server-side anomaly/fraud detection.

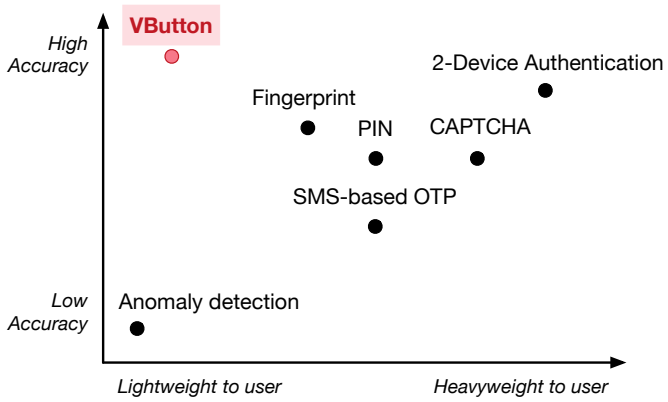

Figure 1: Comparison based on two dimensions: One is user friendliness, and the other is accuracy (both false positive and false negative).

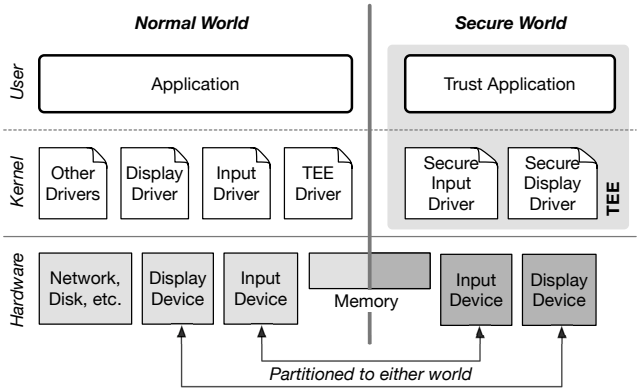

Figure 2: Hardware and software components of TrustZone. The memory is split into Secure World and Normal World, as well as the I/O devices.

techniques, such as SMS-based one-time-passwords (OTP), can be bypassed by user-impersonating malware, which have access to users' SMS database or push notification history.

Anomaly/Fraud Detection: Server-side anomaly or fraud detections, especially those using machine learning [26, 28], are useful for identifying fake or harmful inputs on the client side. However, the accuracy of such methods heavily depends on the model and training data, and may vary significantly across different scenarios. Even if some anomaly is detected, in order to reduce false positives, the server usually falls back to other traditional methods like CAPTCHA.

Despite the mitigations discussed above, a practical, accurate (low false positive and low false negative), and trustworthy technique is needed for online service providers to attest the authenticity of user-driven operations/requests originated from untrusted mobile devices.

2.2 Background: ARM TrustZone

TrustZone is a security extension in ARM SoCs (Systems on Chip) first introduced in ARMv6 [12]. It offers a hardware-backed trusted execution environment (TEE), referred to as the *Secure World*, for running security-critical code. The regular software stack, including the OS and apps, runs in a parallel and less-privileged environment called the *Normal World*. Figure 2 shows a typical architecture of TrustZone.

Strong isolation: TrustZone-enforced isolation applies to hardware resources including processors, memory, and peripherals. The allocation or assignment of the resources can be dynamically configured by the Secure World OS (or TEE OS). When a processor is running in the Normal World, it can only access the Normal World memory. In contrast, when running in the Secure World, a processor can access all memory. A peripheral can be assigned to either the Secure World or the Normal World. For instance, the TEE OS can assign the touchscreen to the Secure World while handling PIN input, and otherwise keep it in the Normal World. Some devices are always assigned to the Secure World, like fingerprint and iris scanners, while some devices whose drivers are too complex to run in the Secure World are always assigned to the Normal World, like network and storage.

Secure boot: When a phone boots, the processor first enters the Secure World and loads the signed TEE image to memory if the signature is valid and the image unmodified. Once the TEE OS is loaded with its security enforcement mechanisms initialized, it switches the processor to the Normal World to start the boot chain in the Normal World. As a result, the integrity of TEE OS is guaranteed and the isolation between the two worlds is set up before any untrusted code is loaded.

Remote verification: TrustZone allows a remote party to check whether a mobile device has deployed TEE and verify its secure

boot chain. This verification relies on per-device private keys installed during device manufacturing. These keys are only accessible in the Secure World. Device vendors keep the corresponding public keys. The provisioned keys can also be used for other forms of remote attestation which can be more easily developed now that ARM has established the Open Trust Protocol (OTrP) Alliance [15].

2.3 Threat Model

We assume that attackers could take full control over the Normal World, including the applications and the OS, but not the Secure World. Attackers' goal is to send requests or perform operations on behalf of users, via either user-impersonating malware or user deceptions. For example, a tainted Twitter app can automatically follow many twitter accounts without a user's permission or awareness. Similarly, a compromised PayPal app can stealthily request money transfers out of the user's account or redirect a user-initiated transfer to an unintended receiver.

We rely on the TEE as our TCB. Therefore, we do not consider attacks that may breach TrustZone isolation, including side channel attacks. Meanwhile, we assume users are benign and willing to use our system to prevent themselves from being impersonated or tricked by attackers.

3 DESIGN

We present the design of the VButton system, including the developer-facing interfaces, the TrustZone-based VButton Manager, and the server-side VButton Verifier. Developers can easily integrate VButton UI into their apps and in turn allow first- or third-party service providers to verify the following properties with regard to a sensitive user-driven operation or request:

- **R1. User Initiation:** *The operation is initiated by a human user via an explicit action.*
- **R2. User Intention:** *The user intends to initiate the operation, rather than being tricked to do so.*

3.1 Overview and Challenges

Intuition: The high-level idea behind VButton is as follows. Having strong incentives to prevent user deception and impersonation, online service providers and their first- or third-party app developers are the intended adopters of the VButton system. Such app developers use the *VButton SDK* to easily integrate VButton UI widgets, e.g., `Button` and `View`, in their apps. Virtually identical to normal UI objects, VButton UI widgets are used for app users to initiate critical user-driven operations including login and purchasing. Unlike normal app UI, these widgets are directly displayed and monitored by the VButton Manager in TEE, as opposed to their hosting apps or the underlying OS. This design ensures that attackers cannot modify, obscure, or trigger any VButton UI when being displayed. As a result, user initiations (**R1**) performed via VButton UI can be reliably and accurately recorded.

In this context, verifying user intention (**R2**) for an operation entails solving two problems: (1) showing the information of the initiated operations to the user; (2) ensuring the user is aware of the information. To solve the first problem without having to heuristically infer operation semantics, we let VButton UI fully and explicitly express operation semantics. To do so while keeping our system operation-agnostic, the VButton Verifier helps service providers dynamically generate VButton UI widgets to be displayed in apps. We solve the second problem by designing two types of VButton UI that require user's explicit interaction, at the same time supporting a wide range of operations. For example, a VButton UI for following a user in a Twitter app is rendered by a remote Twitter server upon the request of the VButton SDK in the app, which then passes this UI to the VButton Manager for display. Now the verification of **R2** becomes a simple comparison between the apperance of the server-generated UI and the UI captured on the client device.

System Overview: At the core of the VButton system is the *VButton Manager* which runs in TEE. It securely displays all VButton UI for apps, monitors user input events on these UI, and generates a signed attestation after a VButton UI is triggered. On the server side resides the *VButton Verifier*. For each user-driven operation, it generates the VButton UI on demand to be displayed in the client app, and after the UI is triggered it verifies the attestation blob sent from the client app alongside the operation request. Our system also includes the *VButton SDK* which app developers use to adopt VButton UI widgets in their apps. The SDK hides the attestation-related logic from developers including requesting the VButton UI from the Verifier, passing the UI to the Manager for display and monitoring, and after the UI is triggered, obtaining the attestation blob from the Manager and sending it to the Verifier along with the operation request.

Our system supports two attestation modes, *Explicit Attestation* and *In-situ Attestation* which app developers can choose. The *Explicit Attestation* mode is suitable for semantically complex operations whose meanings cannot be fully expressed by a simple UI, such as a button. In cases like transferring X amount of money from A to B on certain date, our system uses an *operation preview* as the VButton UI, which presents the operation semantics to the user for confirmation. The *In-situ Attestation* mode is designed for simple operations whose semantics can be expressed in a small UI object, e.g., following user X. In this mode the VButton UI looks like a standard button that blends into the rest of the app's UI. No explicit preview or confirmation is needed. A touch/click on the button alone is enough for our system to capture user initiation and true intention.

The design of VButton overcomes the following challenges:

- **Untrusted apps and OS**: The VButton Manager needs assistance from apps, e.g., to trigger the display of preview, and relies on the OS for certain operations like network I/O. However, both the app and OS may be compromised, and may send fake data to the VButton Manager. Thus, our design must be robust against possible evasions or manipulations.
- **Minimal and generic support in TEE**: The code running in TEE (i.e., our TCB) should be as small as possible while

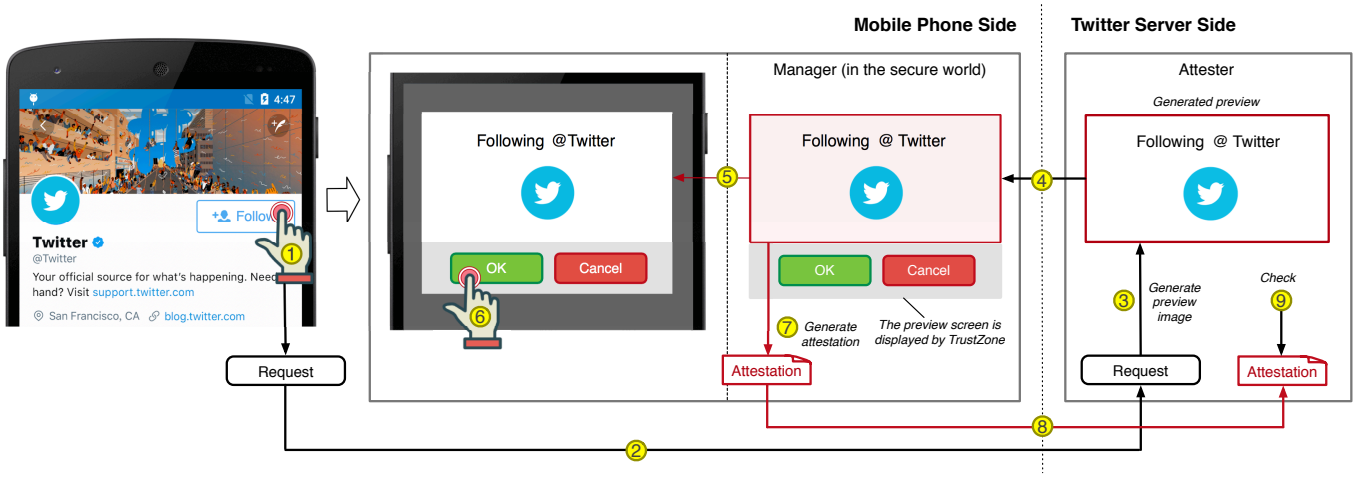

Figure 3: Performing the "follow" operation in a Twitter app that uses the explicit attestation mode of VButton.

maintaining generic support for all apps (as opposed to being app-specific or operation-specific). This requires VButton Manager to be small in size and independent of app semantics and functionality.

- **Ease of use for developers and users**: VButton's impact on app development and app usability should be minimal. The system cannot require a large effort from app developers or service providers for adoption. Similarly, it should avoid imposing burdens to app users.

Table 2: The fields of an attestation blob.

Field	Description
$hash$	The hash value of the preview image.
$nonce$	A random number generated by the server.
t_{aware}	Interval between displaying preview and pressing "OK".
$signature$	A signature of the above contents signed by TEE.

3.2 Mode-1: Explicit Attestation

In the explicit attestation mode, a preview will be shown to the user for confirmation once a VButton is clicked. We use the "follow" operation in a Twitter app as an example to explain the explicit attestation mode. The process is shown in Figure 3.

Attestation Flow: As the figure shows, when a user touches the "follow" button (Step-1), the app sends a request for the VButton UI to the Twitter server (Step-2). The server then generates a *preview* that captures the semantics of the operation (Step-3) and sends the signed preview image to the app (Step-4). The app forwards the preview to the VButton Manager, which displays it and collects user response, either confirming or dismissing the preview (Step-5). If the user confirms it by touching the "OK" button (Step-6), the VButton Manager in TEE receives the touch event and then generates an attestation that contains the hash of the preview, the timestamp, and a nonce (Step-7). The attestation is signed with the per-device private key before it is sent to the Twitter server along with the request to follow the intended user (Step-8). The server then verifies the attestation using the device public key and checks the hash value (Step-9) to determine if the user saw the exact preview that was generated for this operation by the server.

Since the preview is generated by the server, VButton does not need to understand the operation semantics. In fact, regardless of what a to-be-attested operation is, our system goes through the same steps to verify the authenticity of the operation, i.e., VButton verifies **R1** and **R2** in a general, operation-agnostic way. This design significantly reduces the complexity of the attestation while supporting a wide range of user-driven operations.

Leveraging the privileged TEE, the VButton Manager ensures that the preview, when shown on the screen, cannot be obscured or modified by apps or even the OS. Similarly, only physical touch events (as opposed to software-generated ones) that fall inside the preview region are captured as user responses. The VButton Manager enforces these restrictions by assigning the touchscreen device to the Secure World during the period when the preview needs to be shown. This gives the Manager exclusive permissions to draw on the screen and receive (unforgeable) touch events for the duration of the preview.

The fields in an attestation blob are shown in Table 2. t_{aware} is the time interval from the moment the preview is shown to the moment the user responds. It allows service providers to set a minimum time span for users to read the preview.

Multi-page Preview: In some cases a single preview is not big enough to show the entire semantics of an operation, e.g., a relatively long email to be sent. We extend the preview to support multiple pages/screens, as shown in Figure 4. In this example, when the user is sending an email, she is presented a preview of two pages. The Verifier generates both pages which are displayed by the Manager. The Manager captures each preview page individually and attests user input to each page.

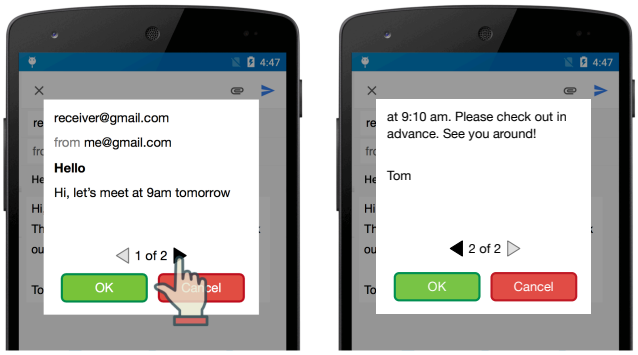

Figure 4: Multi-page preview when sending email using VButton.

3.3 Mode-2: In-situ Attestation

The explicit attestation mode described above is a general design that can be used for all types of operations. However, it requires the user to go through an additional confirmation of a to-be-attested operation, i.e., responding to the VButton preview, which can degrade user experience if happens too frequently.

We design an alternative mode, called *In-situ Attestation*, to enable user-transparent attestation without sacrificing security. The design is inspired by our observation that, for many types of operations, an explicit or separate preview is not needed for presenting operation semantics to users and capturing their intent. In cases like user login and twitter following, we can simply embed the preview in the original button that users need to trigger anyways when performing those operations, i.e., no additional preview or confirmation is needed.

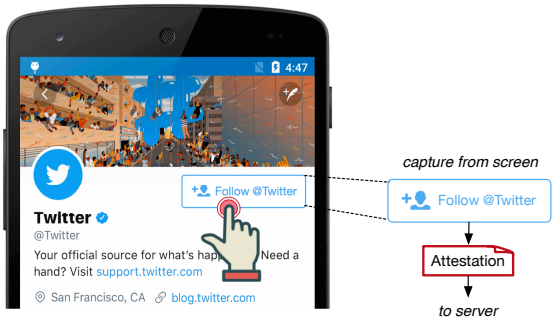

Figure 5: In-situ attestation mode: embedding preview within the button.

To demonstrate the in-situ attestation, we again use the "follow" operation in the Twitter app as an example (Figure 5). To use this mode, the app developer uses the corresponding VButton UI in place of the regular follow button. The VButton UI is generated by the service provider and fully conveys the operation semantics, e.g., saying "Follow *User X*". Unlike the explicit attestation mode, the VButton UI in this case does not cause additional user interaction and does not require user awareness. When the user touches the button, the Manager immediately captures the on-screen image of the entire VButton UI. The rest of the attestation process is similar to that of the explicit attestation mode.

Relying on untrusted apps to display VButton UI poses two challenges to our attestation. First, the visual integrity of VButton UI cannot be guaranteed. Second, the location and orientation of the VButton UI may change as apps update their UI which makes it difficult for the Manager to locate and monitor the UI. We overcome both challenges by having the VButton SDK automatically update the Manager about the current on-screen location of VButton UI elements. Therefore, any compromised or absent VButton UI is captured by the Manager and detected by the Verifier because the captured VButton UI is not identical to the one originally generated by the Verifier, therefore their hashes will not match. The SDK will also enable the button only when it is fully shown to the user, to ensure that the Manager will get the full image.

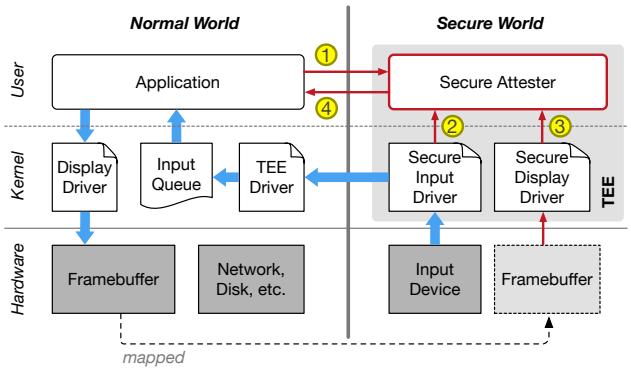

Figure 6: The in-situ preview mode of VButton.

Figure 6 shows the assignment of devices and data flow of the in-situ mode. When an application requires attestation (Step-1), the Attester will keep monitoring all the inputs (Step-2 and capturing screen (Step-3). Once user confirms, the Attester will generate attestation and send it to the application (Step-4).

The most significant difference between the in-situ mode and explicit attestation mode is that the VButton UI has to be displayed by the (untrusted) hosting app, rather than the Manager in TEE. This is because when the Manager draws on the screen, i.e., screen is assigned to the Secure World, the app and the OS in the Normal World can no longer update the on-screen content. This isolation is desirable in the explicit attestation mode where the preview is shown in the foreground while the app UI remains frozen in the background. However, this isolation, and the resulting app UI freeze, can disturb app functionality in the in-situ attestation mode where the app UI needs to be active.

Note that although the screen is shared between the Normal World and the Secure World, hardware interrupts from the screen (e.g., touch events) still go through the Manager in the TEE first and cannot be forged or tampered with by untrusted code.

Special Considerations: Figure 7 shows the timeline of an in-situ attestation. When the "Follow" button appears on the screen, the VButton SDK immediately informs the Manager of the location and size of the button. The Manager then starts to monitor

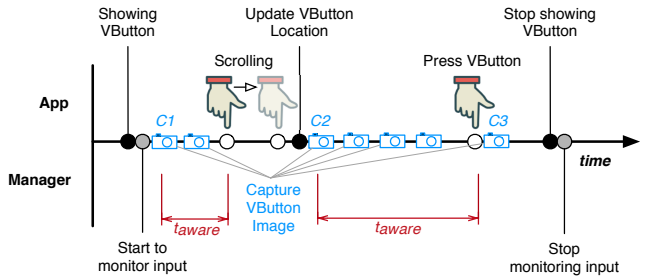

Figure 7: The timeline of an in-situ attestation.

all input events (*C1* in the figure), and capture images of the button at random intervals for verifying visual consistency and detecting TOCTTOU attacks (*C2* in the figure). When a touch event happens within the button area, the Manager captures the button image again (*C3* in the figure), and checks if all captured images are identical, i.e., the user sees the same button for the entire t_{aware} duration. Thus, if the VButton SDK lies about the location of the button, the Manager will compute differences in snapshot images and detect such an attack.

In this mode the VButton UI may move around or even temporarily be rendered off the screen, the t_{aware} value is calculated as the sum of the time periods when the UI is visible since its first appearance. The t_{aware} should be larger than the threshold set by the server.

3.4 VButton API

The VButton SDK provides a set of API for applications to use. Listing 1 and Listing 2 demonstrate how applications may use those APIs to attest the user-driven operation of tapping the Twitter follow button.

Explicit attestation: In Listing 1, Line 3 establishes a secure connection with the Manager in the TEE. Line 4 renders and instantiates the follow button which is a VButton. When the button is pressed, Line 10 requests an operation preview from the remote Verifier. Line 12 feeds the preview image, together with the configuration, to the Manager in the TEE. The Manager then displays the preview over the regular app UI and monitors user input. When the user confirms or dismisses the preview, the Manager returns the screen control back to the Normal World and sends the verification result, including the attestation blob, to the appropriate app callback (Line 14 and 19).

In-situ attestation: In Listing 2, unlike in explicit mode, the follow button is the VButton UI and is composed by the remote server upon the app's request during the initialization phase (Lines 3-4). Line 6 informs the Manager of the button as a monitored region , Lines 7 to 13 update the VButton location as the app UI layout changes. Lines 14 to 31 show the in-situ attestation process triggered by a user click on the button. The app does little more than invoke the in-situ attestation API (Line 18) which takes care of the visual consistency check, user input confirmation, attestation blob generation, etc.

Table 3 lists all the APIs exposed by the SDK that are relevant to either attestation mode.

Listing 1: Sample API use for Explicit Attestation mode.

```
1  import org.VButton.*;
2  ...
3  VButton.init();
4  Button followBtn = (Button)findViewById(R.id.button);
5  followBtn.setOnClickListener(new View.OnClickListener() {
6     // Called when user clicks the FOLLOW button
7     @Override
8     public void onClick(View view) {
9        // Request explicit attestation image from server
10       Bitmap bmp = getPreviewFrame();
11       // Invoke VButton Manager to display and monitor
             the preview UI
12       VButton.UIAttester_attest_explicit(bmp,
             preview_config, VButtonCallback<VerifyResult
             >() {
13         @Override
14         public void onSuccess(VerifyResult r) {
15             // Called when attestation is generated
16             sendFollow("follow", verifyResult);
17         }
18         @Override
19         public void onCancel(VButtonException e) {
20             Log.d(TAG, "User cancels follow operation!");
21         }
22       });
23    }
24 });
```

Listing 2: Sample API use for in-situ attestation mode.

```
1  import org.VButton.*;
2  ...
3  VButton.init();
4  Button followBtn = (Button)findViewById(R.id.button);
5  // Register the in-situ preview region
6  VButton.UIAttester_register_insitu_view(followBtn,
       preview_config);
7  followBtn.addOnLayoutChangeListener(new
       OnLayoutChangeListener(){
8     @Override
9     public void onLayoutChange(View v,int left,int top,int
          right,int bottom,...) {
10       // Update the in-situ preview region
11       VButton.UIAttester_update_insitu_view(v,
             preview_config);
12    }
13 });
14 followBtn.setOnClickListener(new View.OnClickListener() {
15    @Override
16    public void onClick(View view) {
17       // Invoke VButton in-situ preview to verify
18       VButton.UIAttester_attest_insitu_preview(view,
             VButtonCallback<VerifyResult>() {
19         @Override
20         public void onSuccess(VerifyResult r) {
21             // Called when attestation is generated
22             sendFollow("follow", verifyResult);
23         }
24         @Override
25         public void onError(VButtonException e) {
26             // Handle error
27             Log.d(TAG, "follow_button: verify failed!");
28         }
29       });
30    }
31 });
```

Table 3: APIs provided by VButton Manager to untrusted OS.

Command and Parameters	Description
`UIAttester_init()`	Establish a secure connection with the Manager.
`UIAttester_final()`	Release an established connection with the Manager.
`UIAttester_register_insitu_view(view_info, track_config)`	Register a new in-situ button with the Manager. `view_info` includes view ID, button location etc. `track_config` is the configuration parameters defined by the server, including nonce and minimal t_{aware}.
`UIAttester_update_insitu_view(view_info)`	Update the Manager with the new location of an in-situ button.
`UIAttester_unregister_insitu_view(view_info)`	Ask the Manager to stop monitoring an in-situ button.
`UIAttester_attest_insitu_preview(view_info)`	Ask the Manager to attest a registered in-situ button.
`UIAttester_attest_explicit_preview(view_info, track_config)`	Ask the Manager to display the explicit preview, collect user response, and generate attestation.

4 IMPLEMENTATION

We have implemented VButton on two TrustZone-enabled devices: a Samsung Exynos 4412 development board and a commercial off-the-shelf smartphone, Xiaomi RedMi2A. Both are equipped with ARM Cortex-A9 processors. On the Samsung board, the OS in the Normal World is Android Lollipop (5.0) with Linux kernel version 3.15. On the Xiaomi smartphone, we use Android KitKat (4.4) with Linux kernel version 3.10. We chose two different OS versions and devices to demonstrate that VButton system design is OS- and device-independent and can be applied to both legacy and new OS versions. The OS running in the Secure World is a commercial TEE compatible with GlobalPlatform TEE API specification offered by TrustKernel [9]. The VButton Manager runs as a trusted application in the TEE. Generating an attestation blob is computing-intensive. In our initial implementation using libtomcrypt [6], the performance is quite poor. We optimize it by implementing hardware floating point support in the TEE, and replacing the critical part of crypto with the Google *boringssl* [4] library.

4.1 Key Management and Attestation Service

For every TEE-equipped device, a per-device key pair and a unique device ID is generated and securely stored on the device during manufacturing. The public key and the device ID are also kept in a trusted server which can be used for various kinds of attestation. The secure on-device storage is either the *efuse* or the RPMB (Replay-Protected Memory Block) partition of the *eMMC*. The efuse storage is relatively expensive and very small in size. Our current implementation uses the RPMB as the secure storage which is widely supported by mobile device vendors.

On the server side, a signed attestation blob from a client can be verified using the corresponding device ID and public keys. To allow third-party service providers to authenticate devices and verify attestation blobs, we implement an attestation server. It exposes two restful attestation APIs to enrolled service providers: *getNonce* and *attestBlob*. The *getNonce* API is used to create a nonce (a random number unique to each attestation). Used with a timestamp, the nonce prevents replay attacks. The *attestBlob* is used to verify the signature and integrity of an attestation blob received from a mobile device. The APIs follow a customized version of the Open Trust Protocol (OTrP) [20]. Third-party service providers can use this API to leverage VButton to attest the authenticity of user-driven operations, without having to know or manage device IDs or public keys.

User privacy could be a concern since the current implementation requires attestation server to maintain device IDs and public keys which are trackable data. Our solution is to use a *disposable device alias* and generate an *anonymous key pair* in place of permanent device IDs and TEE public keys. Such aliases are securely registered with the VButton Manager on devices upon secure app installation. The secure app will send the aliases and the generated public keys to the server through a secure channel. The aliases are made app-specific and disposed upon an app uninstallation or user request.

4.2 Secure Display

In the explicit attestation mode, the display or screen needs to be secured when a preview is active to prevent untrusted software in the Normal World from influencing what users see. The VButton Manager secures the display by configuring the TrustZone Protection Controller and setting the Display Controller as a secure peripheral, i.e., assigning the screen and the corresponding framebuffer to be exclusively managed by the TEE.

We implement a small driver in the TEE for the secure display controller. This driver helps the Manager draw VButton previews on the screen in a trusted fashion. For each preview display, the driver first freezes the content currently displayed on the screen (e.g., the app UIs). It then draws the preview on top of the frozen content and notifies the Manager which then starts capturing touch events that fall inside the areas of the preview. All the events outside the area of interest will be passed to the normal world. After the user responds to the preview, the display driver recovers the previously frozen screen, and the Manager hands over control of the display back to the Normal World. Note that even though the normal world UI stops rendering during preview, the entire normal OS and UI components are still active as usual.

In the in-situ mode, by design, the screen and the framebuffer are managed by the Normal World OS which allows apps to update their UIs as needed while the in-situ button is shown on the screen. In this case, the secure display driver in the TEE only needs to take snapshots of in-situ VButton UIs at random intervals which are used by the Manager to check the visual consistency of the

UI. The driver is not used for drawing in this mode. Taking snapshots in TEE is done by directly reading the physical memory of the framebuffer indicated by display peripheral registers.

4.3 Secure Input

In the explicit attestation mode, the touch screen peripheral is assigned to the TEE for the duration of a VButton preview. This is done in a similar way to the display peripheral configuration. Additionally, the Inter-Integrated Circuit (I^2C) peripheral connected to the touch input is also protected and assigned to the TEE. This is because physical touch events are received through the I^2C peripheral. Directly managing the peripheral allows the Manager to collect real user inputs while not being tricked by fake ones generated by Normal World malware. Note that there are multiple I^2C peripherals in a mobile phone, each could be connected to multiple peripherals (slaves) and each I^2C could be set as secure or non-secure device independently.

In the in-situ mode, additional input handling logic is activated in the TEE which allows the Normal World OS to receive user input events while the Manager is monitoring both the touch screen and I^2C peripheral. Without the logic, input events are consumed by the TEE and never reach the Normal World when the peripherals are assigned to the TEE. We implement this logic in the secure touch screen driver in the TEE. It forwards intercepted input events to the Normal World OS by writing into the buffer of the Linux input subsystem.

5 SECURITY EVALUATION

In this section, we evaluate the security of VButton system. We perform several security attacks using a set of malware to show the effectiveness of VButton. To have a comprehensive evaluation, we root our phone with a tool called *KingRoot* [2], making sure that our malware could gain full control of Android OS.

The experiments were performed on an off-the-shelf smartphone, Xiaomi RedMi2A, to get results in real cases. We use a computer with Intel i7 qual-core CPU at 3.2GHz, 8GB memory and 2TB hard disk as the server. The screen resolution of the test device is 1280X720.

5.1 Input Injection Attack

Input injection is a common technique used by user impersonation malware to send requests on behalf of users without their awareness. To simulate input injection attacks, we use the *monkeyrunner* tool [7] and generate pseudo-random streams of user input events (e.g., touches and gestures) which result in system-level UI events.

Attack in explicit attestation mode: In the explicit attestation mode, no matter what kind of input events are injected, the VButton Manager does not respond to injected events. This is because the Manager takes input events directly from hardware instead of the Android OS.

Attack in in-situ preview mode: In in-situ attestation mode, the victim application itself does react to the injected touch events. However, the application gets an "attestation error" when it calls *UIAttester_attest_insitu_preview* to attest user operation because from the perspective of the Manager, the monitored display region (i.e., the VButton UI) was never touched.

Table 4: Preview load and display time.

Display Type	Latency (ms)
Display by app in a separated activity	208.0
Display by another app	2062.0
Display by VButton Manager in TEE	375.6

5.2 Display Overlay Attack

We wrote an example rootkit tool called *UIMon* that runs in the background and monitors all application activities. Once a victim application is launched, it immediately shows a fake screen to the user by writing directly to the framebuffer (`/dev/graphics/fb0`), tyring to trick the user to touch the area of a VButton UI that is currently covered by a fake button.

Attack in explicit attestation mode: In explicit attestation mode, the fake screen is not shown to the user because the Manager exclusively controls the topmost layer of the screen where the VButton preview is displayed. Only after the preview exits is the fake screen shown. But by that time, the user has already read and responded to the preview.

Attacks in in-situ preview mode: In in-situ preview, the fake screen is shown to the user, which may trick the user into clicking the wrong button. However, the attestation fails because the server fails to verify the image hash of the clicked area, i.e., the look of the clicked button does not match that of the original button generated by the Verifier.

5.3 Implementation Complexity and TCB Size

VButton is designed to be generic and easy to deploy on existing mobile software stacks. The VButton Manager in the TEE represents the TCB size we added which consists of a trusted application in user mode (about 500 lines of C code), and two secure device drivers in privileged mode (about 800 lines of C code). The complexity of the secure drivers are far less than Normal World drivers because VButton only reads input data from I^2C peripheral and writes simple image data directly to the framebuffer while most of the initialization and configuration work are done in the Normal World drivers. For systems that have deployed trusted UI, VButton can add less code (e.g., displaying logic) to the TCB than those without trusted UI. In our implementation, we modified about 50 lines of TEE driver code in the Normal World. The VButton SDK contains about 600 lines of Java code and 300 lines of C code.

6 PERFORMANCE EVALUATION

In this section, we evaluate the performance of VButton system. The setup is the same as in the security evaluation. The VButton Manager is active only when handling sensitive user-driven operations and remains dormant otherwise. For performance evaluation, we focus on the latency overhead caused by the preview generation as well as the attestation. In practice, since the majority of the latency is from the network, the overhead caused by VButton is negligible.

Table 5: Hash calculation and preview generation cost.

Preview size	Hashing on devices (ms)	Preview generation on server (ms)
1*1	6	0.20
5*5	6	0.27
10*10	9	0.31
50*50	10	0.48
100*100	13	0.98
200*200	25	2.70
300*300	45	5.80
720*640	200	28.00

Table 6: Input handling latency of different mode.

Input Type	Duration (ms)
Original input	48.6
Preview input	27.8
In-situ input	52.2

6.1 API Performance Microbenchmarks

VButton preview loading time: The loading and display of previews can introduce latency starting from the moment a preview request is sent to the Verifier by an application to the moment that the preview is shown on the screen by the Manager. As a comparison, in addition to the latency of displaying the preview using the TEE, we also measured the latencies of displaying the same preview using Android in two settings: displaying in a different activity of the same application, and displaying in another application through IPC. The evaluation result is shown in Table 4, which indicates that the latency is relatively small.

Attestation generation time: We calculate the time to generate an attestation blob using the SHA256 hashing algorithm 2048-bit RSA key. We average the time across 100 attestations and the result is 56ms which is acceptable.

Preview generation and hash calculation time: In our design and implementation, the preview frame is generated on the server side. We use a Java library called *textimagegenerator* [8] which creates images from text and image based content to generate preview frames. Display hash calculation is done both server-side and on device using the perceptual hash algorithm [27] which is scaling-resistant and produces a hash value of 64 bits. We measured the time to generate hashes and previews using different VButton UI sizes, and summarize these measurements in Table 5. The results show that the two operations add small latency especially compared with the network latency.

6.2 Input Handling Latency

We measure the input handling latency as the time difference between an input interrupt occuring and the input data being received by the intended software. Table 6 shows the results. *Original input* is the input latency of unmodified Android. *Preview input* is the latency between a generated secure input event and the input being consumed by the Manager. *In-situ input* is the latency from a generated input event to the input being received by the Android application. The evaluation results in Table 6 show that the input handling overhead incurred by putting the input device driver in the TEE is quite small. For secure preview input, the input handling routine in the TEE is simpler and quicker than Android's and thus incurs a shorter latency.

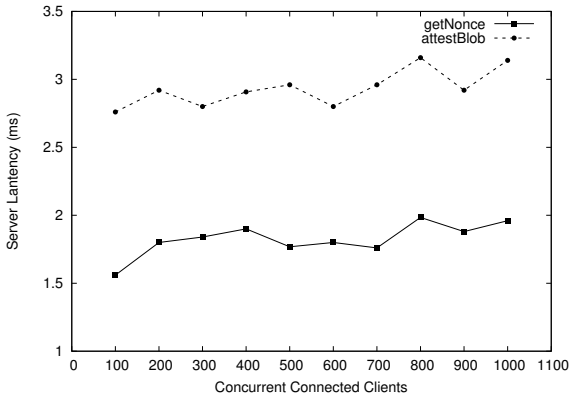

Figure 8: Latency of attestation server.

6.3 Server-side Attestation Performance

We measure the server latency by simulating 100 to 1,000 concurrent user requests using Apache Benchmark. As introduced in Section 4.1, the attestation server provides two restful APIs, *getNonce* and *attestBlob*, which are called in pairs for each attestation request. Since the signing time is longer than verification time, we use a set of pre-signed requests for server stress tests, and disable the replay detection in the attestation server. The preview frame generation is replaced with a predefined image in *getNonce* API in this test. The number of device public keys stored in the server is 100,000. Figure 8 shows the attestation latency observed in a local network, which is lower than usual cellular network latency and scales well as the number of concurrent clients increases.

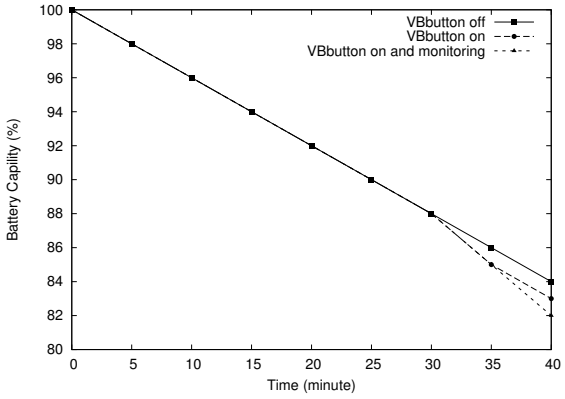

Figure 9: Changing of battery capacity.

6.4 Power Consumption

We evaluate power consumption of the VButton system under three configurations of the Manager: disabled, on-in-background, and on-and-monitoring. We create a simple, 10-minute workload that includes: playing a video, surfing the Internet, playing interactive games, and typing messages. Figure 9 shows how much battery capacity is consumed after 4 repeated workloads (40 minutes), under the three configurations. Even in the worst case when the Manager is on and constantly monitoring UI and input events, its impact on battery is barely noticeable. In reality, VButton is invoked on demand instead of always on. Therefore, we expect that VButton incurs negligible battery overhead in practice.

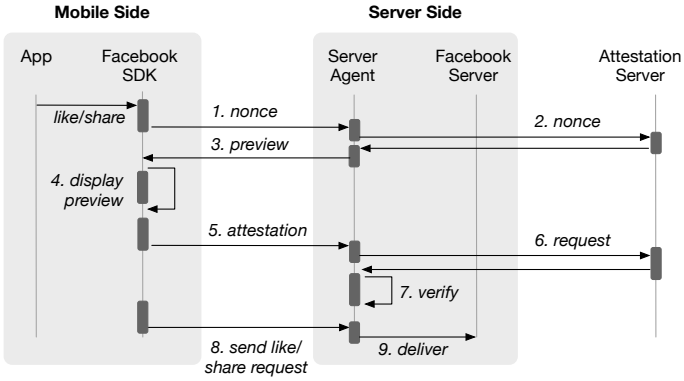

Figure 10: Facebook like & share attestation timeline.

6.5 Case Study: Facebook Like & Share Button

Online social networks such as Facebook provide *like* buttons and *share* buttons for users to easily endorse or share webpages and apps among their friends. While these buttons are convenient to use, they are subject to hijacking attacks where users unknowingly "like" or "share" unintended content. Facebook urges users to "be cautious to avoid infecting (their) computer(s) with malware" [1].

VButton can defeat these attacks even if a users' mobile device is infected with malware. To demonstrate how to adopt VButton in this case, we customize the Facebook Android SDK (only 32 lines of changed code) and use a *server agent* as the Verifier. Figure 10 shows the interactions among the app, the server agent, the Facebook server and the attestation server. When a user clicks a button, the Facebook SDK sends a 184-byte compressed request package to the server agent, and in return, receives a 299-byte nonce package and a 4KB compressed preview image (300 * 257 pixels) which contains an excerpt of the content the user wants to share. Next, the SDK feeds the preview to the VButton Manager for display. After the user responds and the attestation is generated, the SDK sends the attestation blob to the server agent for verification, whose post body is 231-byte long (compressed). The verification process checks the origin of the request through the attestation server and compares the hash of the preview image. The SDK eventually sends the "like" request to the server agent, which validates the request and delivers it to the Facebook server. The network communication with Facebook server generates 12KB of TLS traffic.

Note that the app that uses the customized Facebook SDK does not need to be modified. Malware may tamper with the SDK, but doing so does not help malware bypass the attestation.

7 DISCUSSION & LIMITATION

Deployability: TEE has become prevalent in mobile devices. Mobile payment providers have utilized TEE to secure online and offline payment in hundreds of millions of mobile devices, including Alipay [11], Wechat Pay [41] and Apple Pay [14]. Since Android Nougat, Google uses TEE as a platform security foundation in Android [3] and has mandated all Android vendors to support TEE in their systems in order to pass the Android Compatibility Test Suite (CTS). In the meantime, more and more TEE vendors support OTrP for dynamic trusted application deployment. The design and implementation of VButton follows this trend by requiring slight modification to existing OS input drivers, and installing VButton manager as an independently trusted application in TEE; thus the deployment effort is small.

Downgrade attack: Usually, a server will not enforce attestations for all the operations, since a user may not use VButton, or may operate on another device (e.g., PC) without any attestation mechanism. Thus, an attacker could install an old version of the apps or claim that the mobile phone does not support TrustZone, to force the server to downgrade to original mode and thus bypass the attestation mechanism. We suggest application developers use device binding to avoid downgrade attacks: once VButton has been used in a device, any downgrade should be rejected or treated as suspicious. To make it more practical, this mechanism is configurable for each operation by users according to their different requirements. Further, the server can also adopt security policies to detect anomalies, e.g., if a lot of non-attested operations are received in a relatively short period, they may be issued from a smartphone farm.

Fooling the user: Although we try to minimize dependence on the end user, VButton still relies on the user to check the preview. It is still possible that an attack may bypass the attestation. One case is that the preview has multiple pages and the user does not check all the pages; another is that the preview content does not match user's intention but the user does not notice it. In the in-situ mode, an attacker may fool the user by placing some unrelated context near the button. In this case, we require the developers to ensure that the in-situ button contains sufficient information. Otherwise, it is more suitable to use the explicit attestation mode. Similarly, the in-situ mode of VButton is not designed to defend against phishing attacks intended to gain private data. For example, a malware may fake a login page, and steal a user's password. VButton cannot handle such cases, and could be enhanced by other complementary mechanisms like turning on a secure LED indicator when displaying a secure password-input UI.

Relay attack: Relay attack is issued by secretly replacing a VButton-enabled app with a faked one with the same UI. Then it uses a relay device with VButton enabled to talk to the server by employing a real operator to touch the screen for each relay

request. For example, the user uses a faked VButton to follow Alice, the faked app will send request to follow Eve, and use a relay device to send attestation for following Eve. The key of this attack is that the operation on a user's device is decoupled with the attestation generated on the relay device. This kind of attack could be defeated by device binding: a user needs to enroll the VButton enabled device explicitly the first time she uses it, so that the attestation generated by a relay device will not work.

No authentication: Authentication is not considered in our design, which means if a phone is left unlocked, an attacker may do any operation on behavior of the owner, even if VButton is deployed. Such attacks can be prevented by using existing mechanisms like fingerprint, iris checkings, or face detection.

Vulnerabilities of TEE: TEE can have bugs. If an attacker exploits some bug in TEE, and further steals the private key from the TEE, she can then generate any attestation. Currently, VButton trusts the TEE and does not consider such an attack.

8 RELATED WORK

User-driven security: There is a rich body of research on leveraging user actions or intents to inform security detection or enforcement systems. BINDER [18] was among the first to correlate user interactions and security events for the purposes of detecting intrusions. NAB (Not-A-Bot) [23] is a system that defends against botnet attacks on the server side by differentiating human-generated traffic versus bot-generated traffic. Jang et al. [25] proposed to block malicious outgoing traffic of (compromised) network applications by permitting outgoing traffic only when it is attributed to text-based user input. More recently, researchers studied user-driven access control as a means for operating systems to achieve dynamic and minimum granting of permissions [37, 38]. Aware [35] is a newly proposed system that controls apps' access to sensors on Android. Based on a trusted OS, it binds apps' sensor-access requests to user input events and lets users authorize such requests. In comparison, our work follows the same promising idea at the high level—monitoring user interactions for detecting attacks. However, we apply this general idea to a new problem domain (i.e., remotely attesting client-side operations) and solve the unique technical challenges such as untrusted apps and untrusted OS, diverse hardware specifications, and server-side attestation.

Remote attestation for mobile devices: Some previous works have built special-purpose attestation tools for mobile or embedded devices [13, 30, 31, 33]. Liu et al. [31] proposed software abstractions for attesting data collected by mobile sensors. VeriUI [30] focuses on protecting user login by verifying the integrity of the login process to a remote server. C-FLAT [10] enables control-flow attestation on embedded devices. A common design choice among these works, which we also follow, is using ARM TrustZone as the client-side TCB for securely collecting or storing attestation measurements. SchrodinText [13] is proposed to protect the output process of confidential text, including messages, verification codes, etc. It decouples the rendering and displaying of text, leverages the Android OS to get layout of text without accessing the data of the text but only the number of characters, and uses TEE to reorder some pre-rendered glyphs to display the text. VButton focuses on protecting the interaction between user and app, not the confidentiality of displayed data. AdAttester [29] is a system for attesting the integrity of advertisement display on mobile devices (i.e., checking if an ad banner has indeed being displayed as is on a mobile device). Similar to AdAttester, VButton also uses image-based matching as part of the attestation process. However, our work goes beyond the "look" of a UI object, and has the ability to understand and reconstruct the "meaning" of a UI object which is more general and essential to verifying users' intent and perception of a to-be-initiated action.

Securing ARM platform with hardware features: Researchers try to enhance system security with different hardware features on ARM platform like TrustZone and virtualization. Pocket hypervisor [16] proposes to use hypervisor for securing OS and various services on mobile platform. YouProve [21] uses trusted hardware to build the up-layer software stack for protecting the data of mobile sensors. fTPM [36] proposes a TrustZone-based software implementation of TPM-2.0 without a real TPM chip. Ditio [34] tries to improve the security of IoT devices by recording sensor activity logs with both TrustZone and virtualization support. vTZ [24] targets the ARM server platform and supports multiple secure worlds for different virtual machines. These works are orthogonal to our work.

9 CONCLUSION

In this paper, we present a new system, named VButton, to enable a mobile service provider to reliably verify the authenticity of user-driven operations originated from untrusted client devices. Thanks to this new verification capability, service providers can now only accept user requests that are initialized and intended by real users, consequently stopping user-impersonating or user-deceiving malware. By leveraging ARM TrustZone, a widely available secure hardware feature on mobile devices, our system is not affected by powerful malware or even a compromised/rooted OS. Our design of the VButton UI and the two attestation modes makes our system easy-to-use for developers, lightweight for app users, and generic enough to support a wide range of operations.

We implemented our system in a suite of software tools for all parties involved, including the developer SDK, the on-device monitor called the Manager, and the Verifier on the server side. We used a development board and a commercial smartphone as our hardware reference platforms. We evaluated the system in terms of both security and performance. We found that the system is robust against powerful attacks, effective at detecting forged or unintended user operations, and lightweight in terms of its runtime overhead.

ACKNOWLEDGMENTS

We thank our shepherd Jeremy Andrus and the anonymous reviewers for their insightful comments. This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000104, the National Natural Science Foundation of China under Grant Nos. 61572314 and 61525204.

REFERENCES

[1] "Keeping facebook activity authentic," https://www.facebook.com/notes/facebook-security/keeping-facebook-activity-authentic/10152309368645766/, 2014.
[2] "Kingroot," https://kingroot.net, 2016.
[3] "Authentication | android open source project," https://source.android.com/security/authentication/index.html, 2017.
[4] "boringssl," https://boringssl.googlesource.com/boringssl/, 2017.
[5] "Chinese click farm where 10k phones boost app ratings," http://www.dailymail.co.uk/news/article-4499730/click-farm-10-000-phones-boost-product-ratings.html, 2017.
[6] "Libtomcrypt," https://github.com/libtom/libtomcrypt, 2017.
[7] "monkeyrunner," https://developer.android.com/studio/test/monkeyrunner/, 2017.
[8] "textimagegenerator library," https://github.com/jcraane/textimagegenerator, 2017.
[9] "Trustkernel tee," https://www.trustkernel.com, 2018.
[10] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 743–754.
[11] Alipay, "Ali pay," https://www.alipay.com/, 2017.
[12] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, 2004.
[13] A. Amiri Sani, "Schrodintext: Strong protection of sensitive textual content of mobile applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 197–210.
[14] Apple, "Apple pay," www.apple.com/apple-pay/, 2017.
[15] ARM, "Connected devices need e-commerce standard security say cyber security experts," https://www.arm.com/about/newsroom/connected-devices-need-e-commerce-standard-security-say-cyber-security-experts.php, 2016.
[16] L. P. Cox and P. M. Chen, "Pocket hypervisors: Opportunities and challenges," in *Mobile Computing Systems and Applications, 2007. HotMobile 2007. Eighth IEEE Workshop on*. IEEE, 2007, pp. 46–50.
[17] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in android applications," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 123–134.
[18] W. Cui, R. H. Katz, and W.-t. Tan, "Binder: An extrusion-based break-in detector for personal computers," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 363–366.
[19] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 3–14.
[20] O. I. I. E. T. Force, "The open trust protocol (otrp)," https://tools.ietf.org/html/draft-pei-opentrustprotocol-01, 2017.
[21] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox, "Youprove: authenticity and fidelity in mobile sensing," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2011, pp. 176–189.
[22] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet, "Multi-digit number recognition from street view imagery using deep convolutional neural networks," *arXiv preprint arXiv:1312.6082*, 2013.
[23] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy, "Not-a-bot (nab): Improving service availability in the face of botnet attacks," 2009.
[24] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing arm trustzone," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 541–556.
[25] Y. Jang, S. P. Chung, B. D. Payne, and W. Lee, "Gyrus: A framework for user-intent monitoring of text-based networked applications." in *NDSS*, 2014.
[26] M. Jiang, P. Cui, and C. Faloutsos, "Suspicious behavior detection: Current trends and future directions," *IEEE Intelligent Systems*, vol. 31, no. 1, pp. 31–39, 2016.
[27] N. Krawetz, "Perceptual hash algorithm: the average hash algorithm," http://www.hackerfactor.com/blog/?/archives/432-Looks-Like-It.html, 2011.
[28] K. Lee, J. Caverlee, and S. Webb, "Uncovering social spammers: social honeypots+ machine learning," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2010, pp. 435–442.
[29] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 75–88.
[30] D. Liu and L. P. Cox, "Veriui: Attested login for mobile devices," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. ACM, 2014, p. 7.
[31] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 365–378.
[32] W. Liu, Y. Zhang, Z. Li, and H. Duan, "What you see isn't always what you get: A measurement study of usage fraud on android apps," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2016, pp. 23–32.
[33] C. Marforio, R. J. Masti, C. Soriente, K. Kostiainen, and S. Capkun, "Hardened setup of personalized security indicators to counter phishing attacks in mobile banking," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2016, pp. 83–92.
[34] S. Mirzamohammadi, J. A. Chen, A. A. Sani, S. Mehrotra, and G. Tsudik, "Ditio: Trustworthy auditing of sensor activities in mobile & iot devices," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2017, p. 28.
[35] G. Petracca, A.-A. Reineh, Y. Sun, J. Grossklags, and T. Jaeger, "Aware: Preventing abuse of privacy-sensitive sensors via operation bindings," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 379–396. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/petracca
[36] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A software-only implementation of a tpm chip," 2016.
[37] T. Ringer, D. Grossman, and F. Roesner, "Audacious: User-driven access control with unmodified operating systems," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 204–216.
[38] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Security and privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 224–238.
[39] S. Sivakorn, J. Polakis, and A. D. Keromytis, "I'm not a human: Breaking the google recaptcha," *Black Hat,(i)*, pp. 1–12, 2016.
[40] T. Support, https://twitter.com/support/status/421400317524070402, 2016.
[41] Tecent, "Wechat pay," https://pay.weixin.qq.com/index.php/public/wechatpay, 2017.