TruZ-Droid: Integrating TrustZone with Mobile Operating System

Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du
Syracuse University, Syracuse, New York, USA
{kying,aahlawat,balshari,yjiang13,pthavai,wedu}@syr.edu

ABSTRACT

Mobile devices today provide a hardware-protected mode called Trusted Execution Environment (TEE) to help protect users from a compromised OS and hypervisor. Today TEE can only be leveraged either by vendor apps or by developers who work with the vendor. Since vendors consider third-party app code untrusted inside the TEE, to allow an app to leverage TEE, app developers have to write the app code in a tailored way to work with the vendor's SDK. We proposed a novel design to integrate TEE with mobile OS to allow any app to leverage the TEE. Our design incorporates TEE support at the OS level, allowing apps to leverage the TEE without adding app-specific code into the TEE, and while using existing interface to interact with the mobile OS. We implemented our design, called *TruZ-Droid*, by integrating TrustZone TEE with the Android OS. TruZ-Droid allows apps to leverage the TEE to protect the following: (i) user's secret input and confirmation, and (ii) sending of user's secrets to the authorized server. We built a prototype using the TrustZone-enabled HiKey board to evaluate our design. We demonstrated TruZ-Droid's effectiveness by adding new security features to existing apps to protect user's sensitive information and attest user's confirmation. TruZ-Droid's real-world use case evaluation shows that apps can leverage TrustZone while using existing OS APIs. Our usability study proves that users can correctly interact with TruZ-Droid to protect their security sensitive activities and data.

KEYWORDS

TrustZone, Android

1 INTRODUCTION

Smartphones have become an essential part of our lives, and are used daily for important tasks such as banking, shopping, text messaging, etc. The security of the software running on these devices has become more critical because of widespread smartphone usage. Unfortunately, the recent trend has not been promising. CVE reports show a rise in the number of vulnerabilities in the Android OS, increasing from 125 in 2015 to 523 in 2016 [1]. The design of mobile OSes, such as Android, introduces risks by allowing the untrusted code from app markets to run. By compromising the OS, malware can steal user's secret information such as bank passwords, and spoof actions such as transferring money out of the user's bank accounts on behalf of the user. A technology called TEE (Trusted Execution Environment) has been introduced to help protect users in the event of OS compromise.

The most commonly deployed TEE on mobile devices is ARM TrustZone. Other hardware technologies also support the TEE, including AMD Platform Security Processor, Apple Secure Enclave and Intel Software Guard Extensions (SGX). TrustZone provides a trusted execution environment on the device (called *secure world*), isolated from the environment where normal apps are installed (called *normal world*). TrustZone's security protection can provide confidentiality and integrity guarantees for user's intended activities, leading to its application in many scenarios. For instance, Samsung pay [17] allows users to protect their credit card information in the secure world. Users' typed credit card numbers or pin codes are stored in the secure world rather than in the normal world. Bitcoin Ledger [6] allows the user to confirm a bitcoin transaction inside the secure world; otherwise, the transaction is not accepted by the server. While TrustZone provides a tremendous value for mobile device users, its benefit is only enjoyed by apps supported by device and TEE vendors.

There are two primary ways for apps to leverage TEE; either the app is developed by a device manufacturer like Samsung, or the app developer has worked with the TEE vendor to use their SDK (e.g. Trustonic SDK [14]) and TAM services (e.g. Intercede MyTAM [13]) with OTrP [15]. The first approach is used by apps like Samsung Pay [17], while the second approach is used by apps like Alipay [2]. Since utilizing TEE requires collaborating with the vendor to tailor the app to work with the vendor SDK, such approach makes the use of TEE difficult for the mass of app developers, preventing them from utilizing TrustZone's confidentiality and integrity benefits to protect user's intended activities. Device and TEE vendors do not want to open TEE to any app due to security concerns because vendors do not want any untrusted code to run inside the secure world. This restriction is necessary because allowing untrusted code to run is exactly what has contributed to the OS compromise in the normal world. This is also the reason why TrustZone is locked down in commercial phones before they are shipped, so nobody (other than the vendors) can make changes to the code inside the secure world.

We believe that allowing third-party apps, such as banking, shopping, and medical apps, to use TrustZone can benefit users significantly. Apps commonly take secret input via keyboard and send to servers. Taking mobile banking as an example, we identify two potential risks if a banking app does not use TrustZone. First, when login to the bank's server, users need to type a password, which can be stolen if the OS is compromised. Second, when the user

MobiSys '18, June 10–15, 2018, Munich, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5720-3/18/06…$15.00
https://doi.org/10.1145/3210240.3210338

conducts a money-transfer transaction, the compromised OS can replace the receiver's account number with the one belonging to the attacker, leading to loss of money. If such an app can use TrustZone, the aforementioned risks can be minimized. TrustZone can allow users to type their password in the right app without leaking to the untrusted normal-world OS. Moreover, before an important transaction is committed, TrustZone can ask users for confirmation, so that the transaction can be attested and its integrity can be preserved.

It is important to allow apps to use TEE via existing normal-world OS APIs and without a need to install app-specific TA in the secure world. This is a challenging requirement. Without such support, developers need to make significant changes to their apps to use TrustZone, discouraging them from using it in their apps.

In this paper, we present a design that allows normal-world apps to leverage TrustZone via existing OS APIs. We achieved the goal by incorporating the generic TrustZone support at the OS level so that normal-world apps can use TrustZone without the need to put their own code inside the secure world. We make the following contributions: (1) We proposed a novel design to integrate TrustZone with the mobile OS. Our design consists of two major components: TruZ-UI used for protecting user's confidential input and integrity preserved confirmation. TruZ-HTTP and split SSL used for sending the TEE-protected data to authorized servers. (2) We implemented our system design in the Android OS, running on a prototype that we built using the TrustZone-enabled HiKey board. (3) We evaluated our system using real-world apps, including both open-source and closed-source apps. The evaluation results show that our system allows third-party apps to leverage TrustZone for a variety of use cases with minimal changes to apps. We also evaluate the usability of TruZ-Droid based on users' feedback. Our evaluation results show that users can make the right access control decision to protect their security sensitive activities using TruZ-Droid.

2 PROBLEM AND IDEAS

In this section, we discuss the problems and constraints in providing TrustZone support to third-party apps and our ideas on how to solve these problems.

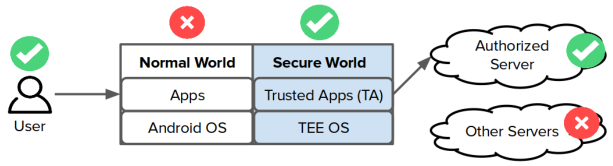

Figure 1: Threat Model

2.1 Threat Model

We assume the adversary model as shown in Figure 1. The user of the device is trusted. The normal world that includes the apps and Android OS is untrusted. They may attempt to steal the user's secret data and spoof an unauthorized action on the user's behalf. The secure world that includes the Trusted Applications (TA) and TEE OS is trusted. It will protect the user's confidentiality and integrity when the normal world is compromised. We assume that the server is trusted after it is authorized by the user. The authorized server wants to protect the user's confidential data and verify the integrity of the user's request; other unauthorized servers are considered untrusted and they may collaborate with the normal world to steal the user's secret data.

2.2 Problem

We state the following problem: *How to enable third-party apps to reuse the existing OS interfaces to leverage generic TrustZone support to protect user's private data and enforce user's intention without putting app-specific code in the TEE?*

We further break down the problem into (a) protecting user's sensitive data and sending it to the server, and (b) protecting and attesting user's intention. Users type sensitive inputs when using Android applications. In order for an app to protect such inputs using TrustZone, users should be able to type a secret without allowing the compromised OS to see the secret. Given a protected secret, the app should be able to send the secret to the authorized server without leaking the secret to the compromised OS. In order for an app to enforce user's intention using TrustZone, users should be able to confirm an action (e.g., money transfer) and the compromised OS should not be able to modify the user's confirmed action. The user's confirmation should be attested (signed) using TrustZone. The attested confirmation should allow the receiving server to verify that the action was confirmed by the user.

The problem of protecting user's sensitive data and user's intention has been solved by TrustZone, but the current solutions [45, 53, 55, 57] do not satisfy the following constraints: (a) normal-world apps can reuse existing OS interfaces to leverage the TrustZone support, (b) no app-specific logic in the secure world, and (c) minimize Trusted Computing Base (TCB) while providing generic TEE support. In order to allow an app to protect user's activities and data with minimal changes, the developer should be able to use existing Android components and APIs, and still be able to leverage TEE support. If an app is required to replace Android components to integrate TEE support, it would result in a significant change to the app. An example of this is Samsung KNOX [55], which provides a vendor SDK that allows app logic to be integrated with TrustZone components in the secure world. It is important to not require the app developer to write code in the secure world, which minimizes the risks to the secure world. Research works [45, 53, 57] allow application-specific logic to run in either the Trusted Application (TA) or the TEE OS [18]. Such an approach restricts the number of apps that can use these research works and increases the risk in the secure world.

Given the identified constraints, we further elaborate on the problems of protecting user's interaction and sending of TEE-protected data to the server.

Protecting user interaction. To protect user's secret data, we need to protect the interaction between the user and the device so that the secret data will be never given to the normal-world OS. One of the primary ways that users use to interact with the device for the secret data is via touch input. Two common types of touch-based interactions are typing text (e.g., password) and confirming an action (e.g., confirm money transfer). A compromised normal-world

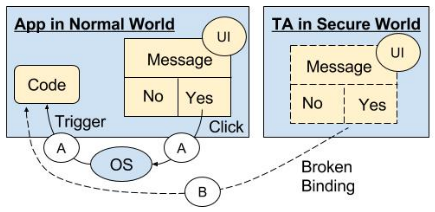

Figure 2: Binding Between Code and UI

OS poses a risk to such interactions. For example, the OS may steal the user's text input. The OS may confirm an action on the user's behalf or change the action after the user confirms it. TrustZone can be leveraged to protect such interactions because of the hardware-level isolation it offers. Given the risks to such interactions from a compromised OS, we state the following problem: *How to allow the normal-world apps to reuse existing EditText and Dialog APIs to protect UI interaction for text input and action confirmation using TrustZone?*

Reusing existing APIs can be achieved by moving the sensitive UI interaction into the secure world, while still maintaining the UI's functionality related to its corresponding code in the normal-world app. Taking the example of Android dialog box for action confirmation, using a dialog box in an app involves two parts: a UI component and a code component. As shown in Figure 2 (*path A*), the OS provides a *binding* between the UI and code to be triggered. Moving the sensitive UI interaction into the secure world breaks the existing binding support provided by the OS, as shown in *path B.* To maintain the same API interface, we should allow the developer to leverage TEE support while using the existing dialog box component and should preserve the UI functionality of the dialog box. The UI's binding to its corresponding code in the app needs to be maintained. When the dialog button is clicked in the secure world, the code for the dialog button in the normal-world app should still be triggered.

Sending protected data to a server. The secret data typed inside the secure world need to be used by applications. If the data are to be processed on the client side (i.e., by apps), it will be difficult not to reveal the data to the normal world. However, in most applications, these secret data, such as passwords and credit card numbers, are only processed on the server side, so they need not be revealed to the normal world, as long as they can be sent from the secure world to the server securely. Given that the normal-world OS is untrusted, we need a way to *allow apps to reuse existing HTTP interfaces to send TEE-protected data to the authorized server without leaking the content to the normal world.*

The research problem can be further broken down into three main challenges: First, our solution needs to be transparent to the existing network protocols and should require minimal modification to the client and server. Second, for performance reasons, our solution should only be used for the communication involving TEE-protected data; other data should simply be transmitted by the normal channel between the app and its server. This requires changes at multiple layers, including HTTP and SSL. Developers know whether TEE-protected data is involved or not, but since they can only tell their intention to the highest layer (the HTTP layer), we need to find ways to convey the developer's intention through multiple layers inside the OS. Moreover, based on the server specific logic, the attacker can trick the server to return a protected secret back to the normal world or post the secret to a website's public field like a Twitter post or a Facebook post. We need to find a way to extend the TEE protection to servers without being intercepted by the normal world. Third, we need to keep the secure-world TCB small. The code inside the secure world is considered as part of the TCB, leading us to keep the entire TCP/IP stack in the normal world for a smaller TCB. We need to send the secret data through the untrusted normal world without leaking the actual content.

2.3 Our Main Ideas

Our main idea to solve the problem is to provide TrustZone support at the OS level, so apps can reuse existing Android components to integrate with TrustZone support with minimal changes. To reduce the risk to the secure world caused by the app-specific logic, we provide generic TAs. We further divide the TrustZone support to protect user's interaction and to protect secret sending to the server.

How to protect user's interaction. Our approach to achieve the required protection is to move the sensitive UI interaction into the secure world and to maintain the binding between the UI interaction and normal-world app code across OSes. This *cross-OS binding* allows the apps to leverage the UI in TEE by using existing APIs. In normal cases, an app developer requests a UI and provides the associated code to be triggered from the UI. Using our approach, the developer will instead request a secure version of the UI and provide the code to be bound to this UI. To the developer, the way to request a secure UI is the same as other UIs, but to the system, when the secure UI needs to be displayed, the corresponding UI is displayed in the secure world. When the UI in the secure world finishes collecting inputs from users, the bound code in the normal-world app is triggered. We refer to this binding support as *TruZ-UI.* To have no app-specific code in the secure world, we provide generic TAs for keyboard and confirmation UIs.

In order to protect the user's interaction in the secure world, the hardware input (touch digitizer) and display (screen content) need to be protected. To protect the user's interaction when the device switches to the secure world, these peripherals should only be accessible from the secure world. Users also need an indicator to identify whether they are interacting with the normal world or secure world. The indicator should be exclusively controlled by the secure world. We leverage the TrustZone Protection Controller (TZPC) to allow the secure world to have exclusive control of I/O and the indicator. When the device is in the secure world, the indicator (we use LED light) is turned on and we show the secure UI on the screen, and accept input from the screen without leaking data to the normal world.

How to send protected data to the server. We work on the network protocol layers, including HTTP and SSL, while leaving the interaction APIs between the application and these underlying layers the same. We split the logic in the network stack in a way that apps can reuse the same HTTP interfaces to send TrustZone-protected data without leaking to the normal world. Because the data in the secure world cannot be revealed to the normal world, existing solutions [29, 45, 55] require moving some of the data-sending logic into the secure world if the data being sent to the server contains TEE-protected information. They perform the splitting at the application layer, this forces the app developers to rewrite their app logic to run on the TEE OS. To be transparent to existing network protocols, we do not modify the protocols but move only the security sensitive logic in these two layers into the secure world. The secure world will encrypt the secret using a one-time key and SSL session key before giving it back to the normal world for sending. The server will decrypt the secret using the one-time key only when the server expects to use the secret in the HTTP request.

Roadmap. Our solution has two main components. We present our detailed design on how to protect the user's interaction and device I/O in Section 3 and our detailed design on how to send protected data to a server in Section 4.

3 TRUZ-UI DESIGN

TruZ-UI allows an app to protect text input and action confirmation using TrustZone by providing cross-OS binding support between the secure-world UI interaction and the normal-world app code. It allows the developer to use existing Android components to leverage secure UI support without adding app-specific logic into the secure world. We next discuss our access control model and how the TruZ-UI design provides protection for user's interactions.

3.1 User Involved Access Control

The OS depends on the user's action to decide how to provide confidentiality and integrity protection for user intended activities. For instance, when a user types a password, he/she depends on the OS (based on the app picked) to provide confidentiality, i.e., the password should go to the right app and its corresponding server. When a user confirms an action in an app, he/she expects the OS to maintain the integrity of the action, i.e., the action that the user confirmed is sent to the server, without being modified. The OS provides confidentiality and integrity guarantees by enforcing access control based on a policy. Part of this policy is decided by the OS, but the other half comes from the user and is derived from the user action. When the user types a password, the OS depends on the user's app selection to decide which app gets the password. When a user confirms an action for a server, the OS can only guarantee that the context of the action will not be modified after the user's approval; the main job of the user is to proofread and ensure that the context of the action indeed matches the user's intention. In our threat model, the normal-world OS fails to provide such security guarantees for users when it is compromised. The only solution for users to protect their security sensitive activities is to convey their intentions to TrustZone to leverage its confidentiality and integrity guarantees.

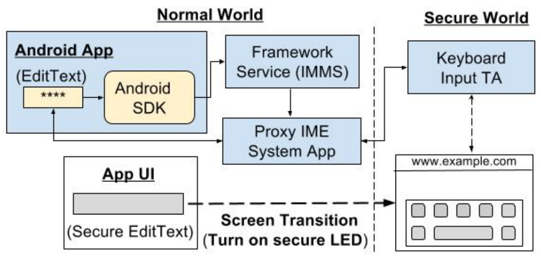

Figure 3: Seamless Keyboard Binding Across OS

3.2 Securing Text Input

In this section, we describe how the user's interaction for text input is protected by seamlessly integrating with the secure-world keyboard UI. Android apps get user's text inputs using a UI element called EditText. When users interact with an EditText, the OS invokes a keyboard. The OS sets up a *binding* between the app and keyboard. The binding allows the keyboard to send user's typed characters to the app's EditText.

To protect user's interaction with the keyboard, we move the keyboard UI into the secure world and provide a binding between the keyboard UI and app's EditText across OSes. Android allows developers to specify a keyboard type when using EditText. To allow the developer to use the existing EditText component to leverage the keyboard UI in the secure world, our design adds a special type called *secure*. The effect of requesting a *secure* keyboard type is shown in Figure 3. The app's secure keyboard request is relayed via the modified Android framework service (`InputMethodManagerService` or `IMMS`) to a new proxy IME system app (Keyboard apps are called IME). The OS sets up a binding between the proxy IME and requesting app. This proxy IME app communicates with a generic Keyboard Input TA, resulting in a secure keyboard UI being displayed on the screen with the secure LED turned on. While the secure keyboard is displayed, the normal world does not have access to the screen display or input. In addition to the keyboard keys, the secure UI also displays a hostname (specified with the secure EditText configuration) that represents the destination server for the typed secret. The importance of the hostname is discussed in Section 5.3.

The Keyboard Input TA communicates with the Keyboard UI to get the user's input. Once the input capture has finished, the secret is saved in the secure-world memory, which the normal world cannot access, and a reference (corresponding to the saved input) is returned back to the proxy IME app. The proxy IME app uses its binding with the app's EditText to return the reference, made accessible via EditText's standard API `getText()` (normally used to get the text typed by the user). A visual feedback is shown in the normal-world EditText by displaying a set of stars. The reference returned from the secure world can support different formats for different scenarios such as passwords, credit card numbers, etc. The design added 1114 LOC in Android (including 634 LOC for a native bridge component to invoke the secure world) and 710 LOC in the TA. The design is easy to migrate (took 50 minutes from android 7.0.0_r1 to 7.0.0_r34).

When an app wants to send TEE-protected data corresponding to reference(s) to the server, it will use the existing HTTP/SSL

API to leverage TrustZone to get an encrypted packet containing the user's secret(s). The reference acts as glue among application, HTTP, and SSL layers. Using TruZ-HTTP and Split SSL (explained in Section 4), secure world will be leveraged to construct the packet containing the user's secrets. The packet will be sent to the server via the normal-world TCP/IP stack.

3.3 Securing and Attesting User's Confirmation

In this section, we describe how we protect and attest user's interaction to confirm an action via dialog box and activity, by seamlessly integrating with a confirmation UI in the secure world. For app developers, asking users for confirmation involves showing a confirmation message to the user and providing code to be executed based on whether users approve or deny the message. The OS provides a binding between the confirmation UI and the code provided by the app. Such user's interactions face risk in case the normal-world OS is compromised, as the OS can confirm a request on behalf of the user or change the message confirmed before it is sent to the server.

To allow the developer to leverage TEE support for user's confirmation while using existing components, we provide cross-OS binding along the existing paths for dialog and activity components. The developer requests a dialog using the `show()` API by providing the message to be confirmed. The app gets back the result via Android input event handling framework which triggers the `onClick()` callback for the dialog button. Similarly, the developer requests an activity using `startActivityForResult()` API by providing the message to be confirmed in an Intent and get back the result via the Intent IPC framework. This triggers the `onActivityResult()` callback. Figure 4 shows the *TruZ-UI* design to allow secure confirmation UI integration for apps. The design allows the developer to request a secure confirmation UI via dialog using the existing API by adding a secure configuration. In case of activity, the developer can configure the `Intent` as secure while using the existing Activity API. The system handles a secure confirmation UI request by invoking a generic confirmation TA via a `TEEBridge service`. In case of dialog, the request is relayed via the modified AlertDialog class, while in case of activity, the request is relayed by a proxy activity, which we provide as part of a system app. Invocation of the confirmation TA results in the switching of the screen to show the secure confirmation UI. The normal-world OS cannot access the display or input at this stage.

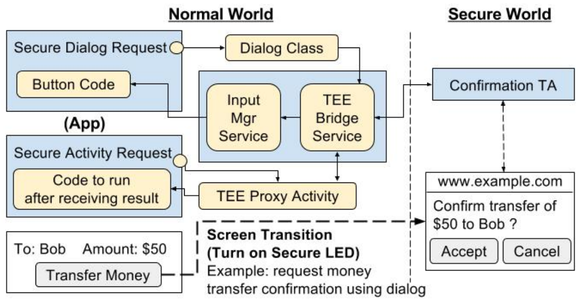

Figure 4: Seamless Confirmation UI Integration

The secure confirmation UI allows the user to approve a message and get it signed by the secure world. As part of the secure configuration, the developer also specifies a hostname, which reflects the server for which the message is being attested, and is displayed in the confirmation UI along with the message. The hostname provides the user a context of the requested confirmation. The hostname serves as a reference to lookup the attestation key in the secure world. The key is setup using our modified HTTP/SSL layer (Section 4). Upon user's confirmation, the message is attested (HMAC signed). In order to improve the user's readability of the message, we allow the developer to add additional formatting in the message to highlight sensitive fields (e.g., a destination account and amount in case of money transfer).

On user's approval, the attestation is returned to the normal-world app. To ensure the confirmation attestation can be returned to existing components, we return the result via existing callbacks for dialog (`onClick()`) and activity (`onActivityResult()`). To return the attestation to the dialog button code, the binding uses the existing event handling framework via the `InputManagerService's` API `injectInputEvent()`, to send a modified `MotionEvent` carrying an attestation. The event triggers the app button's `onClick()` callback where the attestation can be retrieved. In case of Activity, the attestation is returned to the caller app via the `ProxyActivity`, which returns the result to the caller by wrapping the attestation in an `Intent`, which triggers the `onActivityResult()` callback where the attestation can be retrieved. Since the attestation obtained by the app does not contain any user's secret, it can be sent to the server using normal-world HTTP/SSL flow. The server can use the attestation to verify the integrity of the request before taking action. Our attestation scheme currently only applies to user understandable message and cannot work for app-specific semantic like GUID, which users cannot understand. The design added 820 LOC in Android and 680 LOC in the TA. The design used the native bridge mentioned in Section 3.2 and can be easily migrated (took 1.5 hours from android 7.0.0_r1 to 7.0.0_r34).

3.4 Hardware Implementation

All the commercial Android phones with the TrustZone feature have TrustZone locked down by the manufacturers, so we decided to build a TrustZone-enabled prototype platform that can run Android in the normal world and run OPTEE [52] in the secure world. We chose to build our prototype using the HiKey board as our base platform [4] and run the Android 7.0 on it. We used a TFT LCD panel as the screen. The screen uses the HDMI interface for display and the USB interface for touch control.

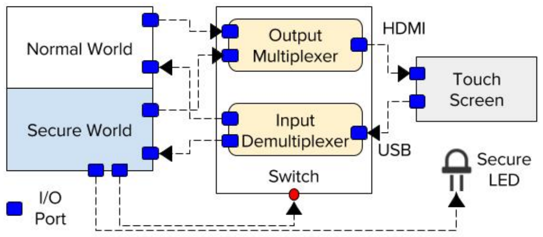

Figure 5: Hardware Implementation

Our hardware implementation provides isolation for the user's input and display. Even though both worlds share the same screen, when the secure world controls it, the normal world cannot access the I/O of the screen. We achieve the isolation at the circuit level. As shown in Figure 5, the I/O of the screen is connected to the multiplexer/demultiplexer. The multiplexer takes the HDMI signal from both the worlds and outputs one of the signals to the screen. The demultiplexer takes the touch input from the screen and gives it to one of the worlds. We use a switch to control the multiplexer/demultiplexer. Each world has separate I/O ports that connect to multiplexer/demultiplexer. The control of the switch is accessible to secure-world I/O ports only. To indicate to users which world they are interacting with, the secure world will turn on a LED when the device is in the secure world. We configure the TrustZone Protection Controller (TZPC) to allow the secure world to have exclusive control of the switch, LED indicator, and secure-world I/O ports.

4 TRUSTZONE-ENABLED INTERACTION WITH SERVER

Normal-world apps can leverage TruZ-UI to allow users to enter sensitive information. The secret data are not revealed to the normal world, but they need to be sent to servers. Given that the normal-world OS is untrusted, we need a way for apps to send the TEE-protected data to the authorized server without disclosing to the normal world.

4.1 TruZ-HTTP

Most apps interact with their servers through HTTPS [7], especially when the data being sent contains secret information, such as passwords and credit card numbers. HTTPS is basically HTTP running on top of the SSL protocol. When an app uses HTTPS, six primary steps are involved: (1) the app provides the URL of the web server to HTTP; (2) the app provides HTTP headers (if needed); (3) the app provides data to HTTP; (4) based on the URL, HTTP invokes SSL to establish a secure channel with the server; (5) HTTP constructs an HTTP request based on the data provided by the app; (6) HTTP gives the completed HTTP request to the SSL layer for sending.

As shown in Figure 6, Steps 1 to 3 involve the app, so changes in these steps should not be at the interface level to maintain the same interface. Since not every HTTP request contains TEE-protected data, for those that do, the secure world should be involved in the sending process; other non-secret-bearing requests should be sent out entirely from the normal world, to avoid the overhead introduced by TrustZone. HTTP does not know whether the payload involves TEE-protected data or not; only apps know that. The question is how to enable apps to inform HTTP about this without changing the way in which they interact with HTTP. We use HTTP headers to solve this problem. We create a new HTTP header field that allows developers to tell HTTP whether the payload contains a reference to the TEE-protected data or not, and if so, where the reference is in the payload. In case of exchanging attestation key with the server (Section 3.3), the app creates an HTTP header with an empty value. The app developers can decide the keep-alive time (for a login session or always alive) for the attestation key. The

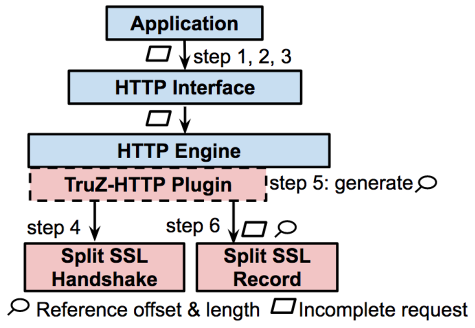

Figure 6: TruZ-HTTP Plugin Design

secure world will fill in the attestation key value for the app in our modified SSL layer (Section 4.2). This solution does not change how apps interact with HTTP and it only adds an extra task in Step 2. We introduce a plugin called *TruZ-HTTP* for the HTTP engine to parse the new HTTP header.

To convey the TrustZone information to the SSL layer, TruZ-HTTP adds additional TrustZone logic to Steps 4,5,6. If an HTTP request involves TEE-protected data, the SSL channel used to send the request must be established by the secure world, so that the encryption keys used by the SSL channel are not revealed to the normal world. Therefore, in Step 4, HTTP will invoke our modified SSL library, so that the secure world is involved in establishing the SSL connection. We provide a detailed discussion on this part in Section 4.2, where we discuss our split SSL design.

After the SSL connection is established, HTTP can give a completely constructed HTTP request to SSL. However, at this point, the secret data inside the request are still represented by their references, not by the actual content. SSL needs to replace the references with the actual content, requiring SSL to know where the references are in the request. This information is known to the HTTP engine after it parses the additional HTTP header provided by the app (the header will be removed after the parsing). Therefore, the HTTP engine has to convey the reference information to SSL. However, SSL is not supposed to understand the logic in the layers above it (e.g., HTTP format). TruZ-HTTP parses the additional header in Step 5 and generates offset and length for each reference. TruZ-HTTP converts the HTTP format specific information into generic string offset and length so that SSL can replace all the references in the request with their actual content without parsing the HTTP request. The additional reference offset and length information is handed over along with the HTTP request to the SSL layer in Step 6. The TruZ-HTTP design assumes that there is no app specific integrity check (e.g., hash) in the HTTP payload. Based on our randomly collected app sample in Section 6.2, most apps meet our assumption.

4.2 Splitting SSL

When apps send sensitive data to their servers, they either go through HTTPS, which is built on top of SSL, or they directly go through SSL. If the sensitive data is in the secure world, part of SSL has to be carried out in the secure world as well.

The main challenge to run SSL in the secure world is to maintain the SSL state information between the two worlds because SSL is a stateful protocol and the TCP/IP stack is still in the normal world. Simply running the entire SSL implementation in the secure world requires conducting the entire SSL Read in the secure world. However, our current design only focuses on protecting a secret being sent to the server, while the data returning from the server will be given to the normal world in its entirety. The entire SSL Read should be conducted in the normal world to avoid overhead. Simply running two copies of SSL in both worlds and synchronizing the SSL states between them is also not a viable solution. The data types of many SSL states are dynamically cast during runtime. Without knowing the actual data types, the states are not serializable between worlds. Our first design moved part of the SSL steps into the secure world and updated the states that have static data type between the worlds. This design was extremely complex because of the complexity of the SSL states and required significant modification of the SSL implementation. We discarded this approach for a cleaner design. A clean splitting SSL design should be stateless across the two worlds and should only run security sensitive logic that deals with the secret in the secure world.

We decided to split SSL between the normal world and secure world to keep all the SSL states in the normal world. SSL can be further divided into SSL session layer and crypto layer. All SSL states are maintained in the SSL session layer. The crypto layer conducts all the crypto operations for SSL and no SSL states are updated in this layer. As shown in Figure 7, we decided to split under the interface of the crypto layer and moved the crypto layer into the secure world. Using our splitting approach, the normal world transfers only essential SSL states to the secure world. Upon exiting from the SSL TA, the crypto return value will be sent back to the normal world. It should be noted that a subset of the crypto return value contains sensitive information that should only be known to the secure world (i.e., the keys used to encrypt the data to be sent to the server). For these crypto return values, the references of keys are returned back to the normal world. The SSL TA stores the actual keys in the secure-world memory.

SSL contains two sub-protocols: the handshake protocol and record protocol [27]. Our design involves splitting these two protocols. For the handshake protocol, if an app needs to use SSL to send data to its server (when part of the data is stored in the secure world), the app, either directly or via HTTP, must go through our split SSL to conduct the handshake with the server. To keep all the handshake SSL states in the normal world, we keep most of the handshake logic in the normal world, except the crypto logic of certificate verification and key exchange. As shown in Figure 7, we further divide the handshake protocol into five main stages (marked as H1 - H5). For each stage, the actual crypto operation is done in the secure world; either the crypto result or the reference of the key is returned back to the normal world. The actual SSL keys are saved in the secure-world memory. Each SSL session is bound with the hostname extracted from the server certificate; the importance of doing so will be discussed later in our security analysis (Section 5.3).

For record protocol, the SSL TA replaces the reference with the real secret. The SSL TA will XOR the secret using a one-time session key that is generated using TEE random device and encodes the XOR result using base64 encoding. The one-time key is inserted in

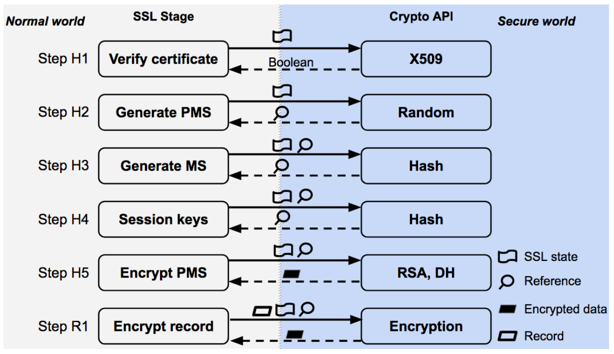

Figure 7: Splitting SSL Design

the HTTP header. The server can extract the one-time key from HTTP header, decode the secret value and XOR the secret data with the one-time key only when the server needs to treat the data as secret. We will discuss the importance of one-time key in our security analysis (Section 5.3). In case an attestation key is needed to attest user's confirmation (Section 3.3), the SSL TA generates an attestation key with random value and inserts into the HTTP header. The SSL TA saves the key and binds it with the hostname as an index. The SSL TA encrypts the record using the write session key. The plaintext key for SSL Read is given to the normal world so that SSL Read can be conducted entirely in the normal world.

TruZ-HTTP adds 595 LOC to the HTTP engine. The size is quite small compared to the total code size of the HTTP engine, which has 74,290 LOC. It is easy to incorporate split SSL design in both worlds. For the normal world, our prototype adds 1012 LOC in 10 functions in Android's SSL library (`boringssl`); for the secure world, our SSL TA consists of 1093 LOC. The crypto part of the `boringssl` library is also ported to OPTEE, but only minimal changes are made. It is also easy to migrate our design for a system update. We migrated our design from android 7.0.0_r1 to 7.0.0_r34. It took 5 hours for TruZ-HTTP plugin migration and 6.5 hours for split SSL migration.

5 SECURITY ANALYSIS

In this section, we present the security analysis of TruZ-Droid. Our design can enforce users' intentions in the presence of either a malicious app or a malicious OS. We pick the stronger attack model and consider a malicious OS as the attacker. Our analysis assumes that the TrustZone hardware platform is trusted and the secure boot process has initialized the integrity-verified OPTEE OS. Hardware attacks, crypto attacks, side channel attacks, and DOS attacks are considered out of scope.

5.1 TruZ-UI Keyboard Security Analysis

As discussed in Section 3.2, normal-world apps can leverage the TruZ-UI Keyboard to capture user's secrets in the secure world. The adversary's goals include monitoring the secret typed, accessing the content displayed, and reading the secret saved in the secure world.

As mentioned in the hardware setup in Section 3.4, the secure world shows the secure UI and gets the screen input through the multiplexer/demultiplexer. The switch controls the USB demultiplexer and HDMI multiplexer. The switch is only controlled by the secure-world I/O ports. We configured the TrustZone Protection Controller (TZPC) to allow the secure world to have exclusive control over the switch and secure-world I/O ports.

The security analysis of TruZ-UI Keyboard involves three properties. The first security property is that the secret typed in the secure world cannot be monitored by the normal-world OS. Since the normal world can neither switch the screen USB input nor read the screen input via the secure-world I/O port, the normal world cannot monitor the user's input in the secure world. This prevents keylogging attacks. The second property is that the content displayed from the secure world is not accessible to the normal-world OS. The normal world can neither switch HDMI output of the screen nor observe the screen content over the secure-world I/O port, preventing it from observing content displayed in the secure world. This helps prevent screen capture attacks. The third security property is that the secret typed in the secure world is never disclosed to the normal world. When a normal-world app uses a secure EditText, the secret typed in the secure world is saved in the secure-world memory. Only the reference of the secret is returned to the normal world.

5.2 TruZ-UI Attestation Security Analysis

As discussed in Section 3.3, normal-world apps can request a secure confirmation UI that provides an attestation for user's approved message. The adversary's goals include forging the approval of the message on behalf of the user and forging or replaying the attestation sent to the server.

The security analysis involves three properties. The first security property is that the attestation generated is always tied to the message displayed in the secure world. The attestation is computed based on the message that the user approves in the secure world when the content matches with the user's intention. The second security property is that the normal world cannot forge user's approval of the message that is displayed in the secure world by performing any type of key injection. This is because the normal world cannot access the touch input when the device is in the secure mode (explained in section 5.1). The message is attested in the secure world only when the user approves it. The third security property is that the attestation generated in the secure world cannot be forged by the normal-world OS. The attestation key is generated inside the secure world and only saved in the secure-world memory. The normal world cannot forge an attestation without the keys. Furthermore, we append a nonce when computing the attestation to avoid replayability.

5.3 TruZ-HTTP and Split SSL Security Analysis

In Section 4, we discussed how TruZ-HTTP and split SSL enable apps to send TEE-protected data to the server, without the normal world knowing the actual content. To defeat this protection, adversaries can attempt two types of attacks: (1) stealing the encryption keys from the secure world, and (2) launching man-in-the-middle (MITM) attacks. The normal world with a compromised OS can trick the secure world into sending the secret data to a malicious server owned by the adversary or trick the TA to send the secret to a website's public field. The normal world can modify the plaintext handshake message to use a weak cipher or a vulnerable SSL version. All attempts are defeated by our security properties.

The first security property is that the key used to encrypt TEE-protected data is never visible to the normal-world OS. We use the key lifecycle to analyze the security of our key management. We can divide the key lifecycle into 1) generation 2) exchange 3) storage 4) usage 5) destruction.

In the key generation, the SSL master secret (MS) is generated from the pre-master secret (PMS), client random number and server random number. Although the normal world can know the client random and server random numbers in plaintext, the PMS is generated in the secure world using TEE-random device and is stored in the secure-world memory only. The normal world cannot infer the MS without knowing the PMS. The SSL key material is derived from MS. Without knowing the MS, the attacker cannot know the key material. Half of the key material is used for SSL Read key and another half of the key material is used for SSL Write key. Although we return the Read key back to the normal world, there is no correlation between the Write and Read keys because the key material randomness is based on the PMS randomness. Our TEE-random device ensures the enough entropy before generating the PMS. The PMS is generated right after the server certificate verification. We bind the PMS with the server hostname to prevent the normal world from misbinding the PMS with other servers.

During the key exchange, the secure world encrypts the PMS using the server public key before giving to the normal world for sending. The normal world cannot know the plaintext PMS because the normal world does not know the server's private key that is protected by the server. The normal world cannot send the PMS to the wrong server because the PMS is bound to the server hostname and the SSL TA refuses to encrypt the PMS using the wrong public key extracted from the certificate.

When the secure world stores the key, we store the PMS, MS, key material, SSL Write key in the secure-world memory only. So the normal world cannot access the keys stored in the secure-world memory.

The normal world can provide the SSL session id that the SSL TA returns after certificate verification to use SSL Write key for encryption. The encryption key is never returned to the normal world. The key is bound to the hostname. To prevent the normal world from using the wrong encryption key (known to the malicious server and stored in the secure world) to encrypt the TEE-protected data and sending to the malicious server, the SSL TA ensures the hostname of the TEE-protected data matches with the key's hostname before doing the replacement and encryption. Each key is also bound with a counter. The counter is used to limit the time that the key can be used to encrypt the data.

We destruct the SSL Write key in the secure world after the secure world encrypts the packet that contains the secret. To avoid the normal world using the plaintext to brute force the key, we will also set the key value to zero when the key's counter reaches its limit. So we limit the footprint of the key in the secure-world memory.

The second security property of our design is that the TEE-protected data is only sent to the authorized server. An adversary in the normal world with a compromised OS can steal the reference for a TEE-protected data, such as the password for Facebook, and ask the secure world to send the password to the adversary's server, which has a valid certificate. This attack is defeated by three critical decisions in our design. First, when getting a secret data item from the user inside the secure world, the user is presented with a message that clearly states the hostname of the server, to which the secret belongs. Once the user approves it and types in the secret data, the data item is bound to the hostname. TrustZone will ensure that the data will only be sent to the server with that hostname. Second, during the SSL handshake protocol, after verifying the server certificate, the SSL TA extracts the common name from the certificate and binds the name to the current SSL session. This binding is saved in the secure-world memory, so the normal world cannot change this binding. Third, when the SSL TA sends out a protected data item to the server, it checks whether the hostname bound to the data item matches with the common name bound to the SSL session. If not, the secret data will not be sent out. The password for Facebook will never be sent to another server because of our protection.

Our design prevents the user's secret from leaking within the authorized server. For example, the attacker can trick the SSL TA to send out the Twitter password as a Twitter post. So the user's secret becomes a public visible message on the Twitter website. The attacker can also append the secret reference to a redirect URL. In a redirect URL, the secret will be sent to the authorized server first. The authorized server will echo the redirect URL that contains the TEE-protected data back to the normal world. Because the split SSL design returns the SSL Read key back to the normal world, the compromised normal world can get the secret data that gets echoed back from the authorized server. To prevent such data leak from the authorized server, the SSL TA encrypts the user's secret twice. The secret value is first XOR with a one-time key generated from the TEE random device. Because SSL TA encrypts the one-time key using SSL Write key that is protected by the secure world (1st security property), so the normal world cannot know the one-time key. The server will XOR the secret with the one-time key only when the server treats the data as secret. The website will post the one-time key encrypted secret value to any public field when the server only treats the secret as data and the server will not decrypt the value. The server can decide not to echo the HTTP header back to prevent the normal world from getting the one-time key. It prevents the one-time key from leaking out of the server.

To prevent the normal world from using downgrade attacks [12, 16, 19, 23] for MITM, the SSL TA disables weak cipher suites and vulnerable SSL versions. The SSL handshake will fail if the normal world rollbacks the cipher suites or the SSL version because the SSL encryption is done in the secure world. The SSL TA rejects the use of a weak cipher suite or a vulnerable SSL version to establish the SSL connection.

5.4 User Decision Security Analysis

In our design, users make the final access control decision when interacting with TruZ-Droid. The only choice for attackers to break TruZ-Droid is to trick users into making the wrong decisions. For instance, attackers can provide a fake hostname or spoof a fake secure-UI window in the normal world when the user types a password. Attackers can provide a fake confirmation message and ask users to confirm the wrong message. We consider all these possible attacks and conduct a usability evaluation in Section 6.3.

6 EVALUATION

In this section, we evaluate our design from four aspects, namely, effectiveness, ease of adoption, usability, and performance. We tested a variety of use cases using real-world applications and measured the ease of adoption for the developers. We tested whether the users can make right decisions when interacting with TruZ-Droid. We also measured overhead imposed by our design and suggested future improvements.

6.1 Effectiveness: Applications

To demonstrate the effectiveness, we added new security features to open-source applications by making changes on the client side and server side (if needed). We modified seven open-source applications, including Elgg [10] and Drupal [9]. To measure the effectiveness in the case of closed-source apps, we modified the OS only for evaluation purpose.

Sensitive file upload. In this case study, we demonstrated how our work can enable normal-world apps to transparently upload a TEE-protected file (e.g., a tax file, a medical record that is only needed by the server, not the client) to the authorized server without adding any app-specific code in the secure world. In contrast, DroidVault [45] requires the app-specific code in the secure world. We use an open-source app called `Seafile` to act as the tax e-file server. The `Seafile` client allows a user to enter a secret (e.g., tax account) via EditText and save it in a file. The app can then upload the tax file to its server using HTTP/SSL. We modify the `Seafile` app to allow the user to enter the secret file content using a secure EditText. The user types the file content using the TruZ-UI keyboard, and the file content is saved in the secure world. The normal world gets a reference, which is saved in a file. When the user asks for the file to be uploaded to the server, the app issues an HTTP request using the normal-world file content (containing the reference). This triggers our modified HTTP engine, which traps into the secure world, where the content of the actual protected file replaces the file content in the HTTP request. The file upload request is then sent to the server. Our TruZ-HTTP and split SSL allow the file to be uploaded successfully to the `Seafile` server.

In this case study, we assume that the file content will not be sent back from the server to the normal world. Our design does not solve the sharing of the file once it reaches the server. Due to the HTTP header from TrustZone, the server will be able to identify that the file is uploaded from TrustZone. The server could deny download of this file until the request comes from TrustZone.

TrustZone-enabled Android authenticator. To demonstrate that our design can support the Account Manager framework (used to manage Android passwords), we wrote an authenticator app for Elgg. When a third-party app needs to login to our `Elgg` server, it will ask the Account Manager, which invokes the authenticator app's login activity. This activity uses a secure EditText to trigger our TruZ-UI keyboard in the secure world. Once the user types the password, a reference is given back to the `Elgg` authenticator. The `Elgg` authenticator then sends the reference to the server through our modified HTTP and SSL. The password reference is saved by the Account Manager, which is not even aware that what it stores is not the actual password. TruZ-Droid allows Account Manager to manage the authentication requests for third-party apps without storing the actual passwords in the normal world. Our design requires no change to the Account Manager framework.

Attested post. We installed `Drupal` on an Ubuntu server and modified the handling of the post content type to verify attestation. We used the `Drupal` Editor app [8] as a client. We modified the app to have an attested post functionality, which allows the user to confirm the post in the secure world before it is sent to the server. We utilized the proxy Activity (refer Section 3.3) for this test to integrate with the confirmation TA. The app sends the secure world attestation along with the post message to the server. The `Drupal` server verifies the attestation before it publishes the post.

Protecting secrets. Apps written today need to protect different types of user's secrets. TruZ-Droid allows developers to protect any text-based secret that can be typed in apps. We evaluated this by using seven different open-source apps and made minimal changes to the apps corresponding to the secrets that needed protection. This involved modifying the layout file containing the EditText corresponding to those secrets and configuring them as secure. The types of secrets protected in apps during the tests included login credentials and payment information.

6.2 Ease of Adoption

We evaluate the ease of adoption of our design by measuring how much effort developers need to make to add TrustZone support to their apps. We conducted the evaluation using both open and closed-source apps. For open-source, we downloaded both the client and server code from public Github repositories [11]. For closed-source, we downloaded apps from Google Play. To ensure their diversity, we downloaded apps from different categories, including shopping, traveling, productivity, finance, medical, business, food, etc.

We totally modified 7 open-source apps, by either adding new features to them (e.g., attestation) or leveraging TrustZone to protect their existing features (e.g., login). We recorded the time spent on the modification and the number of lines of code (LOC) modified for each app. Table 1 shows the result. 1 LOC for TruZ-HTTP, 2 LOC for secure EditText, 4 LOC for secure confirmation.

As shown in Table 1, for apps to leverage TruZ-Droid to protect their login credentials, only 3 lines of code are modified on the client side and the time spent on making the changes is within an hour. For server-side changes, we need 4 lines of code to extract the secret data from the HTTP request. In case of attestation, the attestation logic may vary depending on what to attest. The overall change on the server side is less than 20 lines of code.

To evaluate whether TruZ-Droid works for apps from the market, we enabled closed-source apps to leverage our TruZ-Droid features. To protect users' secret in the secure world, we modify the apps to protect user's sensitive data, including passwords, credit card numbers, and files containing a secret. We repackaged the closed-source apps by configuring some selected EditText in their layout files, so when sensitive data needs to be provided by users, our TruZ-UI keyboard is invoked and the data are typed inside the secure world. To protect users' confirmation in the secure world, we hardcoded the confirmation UI name (activity or activity containing dialog) and the corresponding message in a configuration file. The system uses the file to get a message (corresponding to a confirmation UI request) attested by the user in the secure world. To verify on the server side, we set up a proxy server to verify the attestation. The secure world shares the SSL keys with the proxy server, so it can intercept all the SSL traffic.

Table 1: Evaluation Results for Open-Source Apps

Test Case	Client	Server	Time Spent
Drupal Attested Post	4 LOC	20 LOC	1 hour
Elgg Attested Payment	4 LOC	12 LOC	30 mins
Elgg Authenticator	3 LOC	4 LOC	30 mins
Drupal Login	3 LOC	4 LOC	30 mins
GNUSocial Login	3 LOC	4 LOC	40 mins
Kandroid Login	3 LOC	4 LOC	30 mins
Redmine Login	3 LOC	4 LOC	30 mins
Owncloud Login	3 LOC	4 LOC	40 mins
Seafile Upload	3 LOC	4 LOC	50 mins

In our design, apps need to tell the underlying HTTP and SSL layers that the data to be sent to the server contains the TEE-protected secret, attestation message or attestation keys. Since it is difficult to modify the code of these apps, we hardcoded the information in a configuration file, and let our modified HTTP engine obtain the needed information from this file rather than from the app. All configuration files and the proxy server are only for demonstration purpose. If we can modify the app, such files are not needed.

We collected 31 apps, including Chase, Github, Southwest Airline, Piazza, Priceline, Box, Poshmark, Listonic, Dropbox, MediaFire, Applebee's, Discover, Secure Cloud Storage, etc. We use 15 apps for TEE-protected login, 5 for TEE-protected payment, 2 for TEE-protected file upload, and 9 for attestation. Our results are shown in Table 2. All the experiments were successful, except two cases in the login category. The reason for the failures is not representative; they calculate HMAC of the HTTP request inside the payload. If we have the source code for these failed cases, we can easily make them work with TruZ-Droid.

Table 2: Evaluation Result for Closed-Source Apps

Test Case	Login	Payment	Upload	Attestation
Success/Total	13/15	5/5	2/2	9/9

6.3 Usability

To study whether users can make the right decision when using our system, we conducted an online survey to study the usability of TruZ-Droid. We wanted to test three concepts that TruZ-Droid introduced: (1) using LED to identify different worlds, (2) identifying correct hostname before typing the secret in the secure world, (3) confirming the intended message in the secure world before approval.

Methodology. We recruited survey respondents from Amazon Mechanical Turk (MTurk) where workers had at least 90% task acceptance rate. We conducted a survey in November 2017 and received a total of 161 valid responses. Due to the page limit, the complete survey can be found online [20–22].

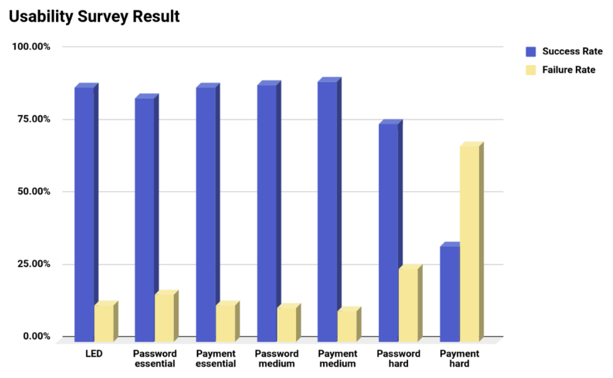

Figure 8: Usability Survey Result

Result. The difficulty of survey tasks can be divided into three categories (1) essential tasks that test whether users can use TruZ-Droid correctly, (2) medium-level tasks that test whether users can identify simple attacks, and (3) hard tasks that test whether users can pinpoint phishing attacks. Respondents made the right decision for all the essential tasks. For instance, 87.58% of respondents correctly identified which mode the device is in by identifying the LED light; 83.95% of respondents chose not to type the bank password when the LED light is off; 87.50% of respondents can confirm the right transaction when the LED light is on. When we increased the task difficulty to medium level, respondents made the right decision in the secure world. For example, 88.46% of respondents chose not to type the bank password for an obvious wrong hostname when the LED light is on, and 92.86% of respondents refused to confirm the transaction with a wrong amount when the LED light is on. When we increased the task difficulty to hard level, the respondents still performed well for typing password but performed in a less promising way for payment confirmation. For instance, 75% of respondents refused to type the bank password for a phishing hostname that is similar to the original hostname, and 31.82% of respondents refused to confirm the transaction with a wrong email address that is similar to the receiver's email address.

Conclusion. When users interact with TruZ-UI, they know how to make the right decision to protect their intended activities in the secure world. Users can correctly identify different worlds using the LED. Users can also make the right decision to only type the password for the right server in the secure world. In case of the payment transaction in the secure world, users can understand how to confirm a message in the secure world and identify obviously wrong confirmation message in the secure world. However, enabling users to make informed and appropriate choices is a hard research problem. We acknowledge that there is an entire research community of usable security researchers working on this challenging problem [28, 36, 42]. In our usability study, although only 32% of respondents correctly refused a confirmation with a spoofed email address, we believe that we can further improve the usability of TruZ-Droid by leveraging these usable research studies [28, 36, 42].

6.4 Performance Evaluation

In this section, we present the performance evaluation result for each major component in the TruZ-Droid design.

We designed experiments to measure the round-trip time for code to secure UI invocation and back. The overhead (average over 20 trials) of our implementation adds over the normal case by not counting the drawing time or the user's input time. The TruZ-UI keyboard integration adds 123 ms overhead. The confirmation UI integration adds 53 ms overhead. Overall, the delay caused by the overhead for the TruZ-UI is barely noticeable when users interact with TruZ-UI.

We also designed an experiment to evaluate the overhead of TruZ-HTTP plugin and split SSL design. We measure the overhead (average of 20 trials) caused by our design. Our evaluation was conducted based on Google, Amazon, and Facebook web servers. The average overhead of TruZ-HTTP plugin is 6.3 ms. The split SSL adds 304.3ms to each HTTPS request. Although this seems to be high, we need to keep in mind that the overhead is only incurred if an HTTPS request contains TEE-protected data (e.g. passwords, credit card number). If not, then there is no overhead. Therefore, this cost is more of a one-time cost.

7 RELATED WORK

In this section, we further discuss the existing works related to TruZ-Droid in the areas of TEE research and Android privacy enhancement research.

TEE-protected UI. Several existing works [44, 45, 53, 64, 66, 70] protect user's interactions by leveraging TEE. All of them move the UI interaction into the secure world, and overcome the broken binding between the UI and corresponding code by moving the code into the secure world as well (binding is maintained within the secure world). These works require the developer to provide the TA code to be executed, resulting in an app-specific TA. VeriUI [47] protects the login web page by porting the WebKit engine and GUI library into TrustZone. VeriUI is designed to protect the entire web page. However, TruZ-UI targets the granularity of UI view elements that build the entire activity. We identify unique design challenges related to view protection in Section 3.2 and 3.3. The existing works require the developer to write TA code and change the app for the TA code invocation. This changes how developers write normal-world apps, preventing them from leveraging TEE support by using existing Android components with minimal change to their apps.

TEE-protected network communication. We compare our design with the related work [29, 45, 47, 53, 55]. The first type of work [47, 53] moves all the layers of communication, including the TCP/IP stack, into the secure world. Although this approach creates a completely isolated communication channel for sending the TEE-protected data to servers, it significantly increases the TCB size. To lower the size, the second type of work [29, 45, 55] keeps the TCP/IP stack in the normal world but ensures that the payload is encrypted in the secure world, and the normal world cannot know the actual content.

Our solution falls into the second type but has several advantages compared to the existing works, including DroidVault [45], TruWalletM [29], and Samsung KNOX [55]. TruWalletM [29] proposes to use two logical SSL subchannels to protect user's login credentials.

They simulated the design completely in the normal world. We have developed a practical solution on the real hardware instead of using simulations, and have identified significant challenges in our split SSL design (Section 4.2) that would not be discoverable only via simulations. DroidVault [45] requires app-specific logic in the secure world to manage the sensitive data from the server and requires the server to adopt the mutual authentication protocol to communicate with the secure world. Our design requires no app-specific logic in the secure world and is transparent to existing network protocols. Furthermore, neither DroidVault nor TruWalletM considers the ease of adoption to apps; they both require significant modifications to the app logic. KNOX [55] requires apps to use new KNOX APIs, which link the normal-world application logic with TrustZone components directly in the application layer. KNOX integration requires introducing new TrustZone components in the application layer and cannot reuse any existing Android components to leverage TrustZone. Unlike KNOX, our solution allows developers to use the standard Android APIs because our modifications are done inside the existing Android components. Since the data in the secure world cannot be revealed to the normal world, all solutions require moving some of the data-sending logic into the secure world if the data being sent to the server contains TEE-protected information. However, whether we can maintain the same APIs for the developers is decided by where the logic is split. All the existing works perform the splitting at the application layer [29, 45, 55]. We work on the underlying layers, including HTTP and SSL, while leaving the interaction APIs between the application and these underlying layers the same.

Trusted execution environment. TruZ-Droid requires no app logic in the secure world and has generic support for common application requirements. These features distinguish us from the most closely related works. Rubinov et al. [53] automatically partitions the app sensitive logic into the TEE. DroidVault [45] establishes a secure channel for uploading and downloading sensitive data to/from the server by leveraging TrustZone. LightSPD [70] emulates a secure portable device in TrustZone to protect the users' privacy. TrustUI [44] enables secure user's interaction by leveraging TrustZone while maintaining a small TCB. TrustICE [64] and PrivateZone [40] load verified normal-world code into the TEE-protected memory and execute them in containers. Liu et al. [49] preserves the integrity of sensor readings by applying TrustZone peripheral protection. Keystore [3] and Fingerprint [5] are part of Android's built-in TrustZone support but their threat model is different from ours. Keystore [3] only protects the keys in the secure world but the compromised normal world can still ask the secure world to decrypt the content using the key. Fingerprint [5] only protects the users' biometrics data but the compromised normal world can still spoof the fingerprint approval without the users' consent. All these works [40, 44, 45, 49, 53, 64, 70] require application logic in the secure world. TrustOTP [63] integrates hardware-based one-time password solution with TrustZone. AdAttester [43] uses TrustZone to provide attested click and display for android advertisements. TruZ-Droid also integrates TrustZone with Android, but our design has generic supports compared to these works [43, 63]. SchrondinText [56] protects the text output of the applications while we focus on text input.

TEE researchers also cover a variety of research directions. TrustZone has an attack surface when using shared memory [50] during the communication between the two worlds, or via side channel like cache [37, 73]. TrustZone also suffers from physical memory forensics [31, 54]. CaSE [72] and CacheKit [71] enhance TrustZone's memory privacy against physical attack. One category of TrustZone research focuses on monitoring the integrity of normal-world memory [25, 41]. VTZ [38] virtualizes TrustZone in the VM. SGX cannot protect user's typed secret like TrustZone, because SGX does not have a separate OS to control the I/O peripherals. SGX is mainly applied in cloud-based applications. SGX studies [24, 26, 58, 60] propose to protect the user's code and data in the enclave even when they are running in a hostile environment. OpenSGX [39] provides Intel SGX emulation platform to develop enclave programs in the emulator. SGX platform suffers from side-channel vulnerabilities [67, 69]. Various solutions [51, 59] have been proposed to solve the side-channel vulnerabilities in SGX.

Android privacy enhancement. Researchers have proposed various solutions [30, 32–35, 46, 48, 61, 62, 65, 74, 75] to prevent privacy leakage in Android. MalloDroid [33] detects potential vulnerabilities against MITM attacks for Android apps. Zhang et al. [74] prevents sensitive runtime information gathering by monitoring suspicious background processes. TISSA [75] proposes fine-grained user privacy access control during runtime. Screenpass [48], TIVO [34], Secure Input Overlay [61] and Guardroid [65] protect the user's password by modifying the Android framework. Unlike these works that fail when the Android OS is compromised, TruZ-Droid can preserve the user privacy even when the OS is compromised.

Split SSL related works. TinMan [68] offloads the confidential data in mobile apps into a trusted node. TinMan synchronizes the SSL states between the client and trusted node. TruWallet [68] and TruWalletM [29] protect the confidential data in the wallet. Both works create SSL proxy and use two SSL connections to isolate the untrusted connection from the trusted connection.

8 SUMMARY

In this paper, we proposed a design to integrate TrustZone with Android that allows apps to leverage TrustZone to protect user's interactions and protect sending the user's secret to the authorized server. We implemented TruZ-UI and split HTTPS/SSL components by modifying Android and OPTEE OS. We tested TruZ-Droid on the HiKey board. Through real-world evaluation, we have shown the effectiveness and ease of adoption of our design.

9 ACKNOWLEDGMENTS

We would like to thank our shepherd, Yubin Xia, and our anonymous reviewers for their insightful comments. This project was supported in part by the NSF grant 1718086. We would like to thank Andrew Henderson and Yaxing Yao for their technical suggestion. We also thank Zhenyu Wang and Amey Ashok Patil for their support to the TruZ-Droid project.

REFERENCES

[1] 2016. Google Android: Vulnerability Statistics. http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224. (2016).
[2] 2017. Alipay online and mobile payments. https://intl.alipay.com/. (2017).

[3] 2017. Android Developers: Android Keystore System. https://developer.android.com/training/articles/keystore.html. (2017).

[4] 2017. Android Developers: Selecting Devices. https://source.android.com/source/devices.html. (2017).

[5] 2017. Android Fingerprint HAL. https://source.android.com/security/authentication/fingerprint-hal. (2017).

[6] 2017. BitCoin Ledger. https://www.ledgerwallet.com/beta/trustlet. (2017).

[7] 2017. Common used Application Layer Protocols. http://www.informit.com/articles/article.aspx?p=169578. (2017).

[8] 2017. Drupal Editor - app by dissem on Github. https://github.com/Dissem/Drupal-Editor. (2017).

[9] 2017. Drupal: Open Source CMS. https://www.drupal.org/. (2017).

[10] 2017. Elgg: a powerful open source social networking engine. https://elgg.org/. (2017).

[11] 2017. F-Droid repository. https://f-droid.org/en/packages/. (2017).

[12] 2017. FREAK CVE-2015-0204 Detail. https://nvd.nist.gov/vuln/detail/CVE-2015-0204. (2017).

[13] 2017. IntercedeâĂŹs MyTAM. https://www.intercede.com. (2017).

[14] 2017. Learn how to become a Trusted Application Developer with Trustonic. https://developer.trustonic.com/. (2017).

[15] 2017. Open Trust Protocol. https://elinux.org/images/2/20/A_More_Open_Trust_Protocol.pdf. (2017).

[16] 2017. Poodle Bites TLS. https://blog.qualys.com/ssllabs/2014/12/08/poodle-bites-tls. (2017).

[17] 2017. Samsung Pay. http://www.samsung.com/us/samsung-pay/. (2017).

[18] 2017. SoC and CPU System-Wide Approach to Security. https://www.arm.com/products/security-on-arm/trustzone. (2017).

[19] 2017. This POODLE Bites: Exploiting the SSL 3.0 Fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf. (2017).

[20] 2017. TruZ-Droid Usability Education Video for Payment. https://youtu.be/qVWwmpickgQ. (2017).

[21] 2017. TruZ-Droid Usability Education Video for Typing Password. https://youtu.be/BZrpKZdQB0g. (2017).

[22] 2017. TruZ-Droid Usability Survey pdf. https://drive.google.com/open?id=1w0_X_99X69eXw4vEeKD_e0QNbPOTNeO6. (2017).

[23] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelink, and Paul Zimmermann. 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *22nd ACM Conference on Computer and Communications Security.*

[24] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 16).* Savannah, GA, USA.

[25] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS' 14).* Scottsdale, Arizona, USA.

[26] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 14).* Broomfield, CO, USA.

[27] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Beguelin, and Jean Karim Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. In *IEEE Symposium on Security and Privacy.*

[28] Christina Braz and Jean-Marc Robert. 2006. Security and Usability: The Case of the User Authentication Methods. In *Proceedings of the 18th Conference on L'Interaction Homme-Machine.*

[29] Sven Bugiel, Alexandra Dmitrienko, Kari Kostiainen, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. TruWalletM: Secure web authentication on mobile platforms. In *International Conference on Trusted Systems.* Beijing, China.

[30] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I. Hong, and Yuvraj Agarwal. 2017. Does This App Really Need My Location?: Context-Aware Privacy Management for Smartphones. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* (2017).

[31] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, , and Alec Wolman. 2015. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15).*

[32] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. 2014. SpanDex: Secure Password Tracking for Android. In *23rd USENIX Security Symposium (USENIX Security 14).*

[33] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security.*

[34] Earlence Fernandes, Qi Alfred Chen, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. 2014. *TIVOs: Trusted Visual I/O Paths for Android.* Technical Report. University of Michigan.

[35] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. 2017. *Android UI Deception Revisited: Attacks and Defenses.*

[36] John D. Gould and Clayton Lewis. 1985. Designing for Usability: Key Principles and What Designers Think. *Commun. ACM* (1985).

[37] Roberto Guanciale, Mads Dam, Hamed Nemati, and Christoph Baumann. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *Proceedings of the 37th IEEE Symposium on Security and Privacy.* San Jose, CA, USA.

[38] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium (USENIX Security 17).*

[39] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. 2016. OpenSGX: An Open Platform for SGX Research. In *Proceedings of the Network and Distributed System Security Symposium (NDSS' 16).* CA, USA.

[40] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. 2016. PrivateZone: Providing a Private Execution Environment using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing* (2016).

[41] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and TEE. In *NDSS.*

[42] Ronald Kainda, Ivan Fléchais, and A.W. Roscoe. 2010. Security and Usability: Analysis and Evaluation. *2010 International Conference on Availability, Reliability and Security* (2010).

[43] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. 2015. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services.* NY, USA.

[44] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. 2014. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems.* Beijing, China.

[45] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. 2014. Droidvault: A trusted data vault for android devices. *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on* (2014), 29–38.

[46] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. 2017. PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* (2017).

[47] Dongtao Liu and Landon P. Cox. 2014. VeriUI: Attested Login for Mobile Devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications.* CA, USA.

[48] Dongtao Liu, Eduardo Cuervo, Valentin Pistol, Ryan Scudellari, and Landon P. Cox. 2013. ScreenPass: Secure Password Entry on Touchscreen Devices. In *Proceeding of the 11th annual international conference on Mobile systems, Applications, & Services.* Taipei, Taiwan.

[49] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. 2012. Software Abstractions for Trusted Sensors. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services.* New York, NY, USA.

[50] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi1, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium (NDSS' 17).* San Diego, CA, USA.

[51] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Security Symposium.* Austin, Texas, USA.

[52] OP-TEE. 2015. OPTEE OS. (2015). https://github.com/OP-TEE/optee_os

[53] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated Partitioning of Android Applications for TEE. In *Proceedings of the 38th International Conference on Software Engineering.* NY, USA.

[54] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. GUITAR: Piecing together android app GUIs from memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 120–132.

[55] Samsung. 2013. KNOX White Paper. (2013).

[56] Ardalan Amiri Sani. 2017. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*.

[57] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[58] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*. CA, USA.

[59] Jaebaek Seo, Byoungyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the Network and Distributed System Security Symposium*. San Diego, CA, USA.

[60] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS' 17)*. San Diego, CA, USA.

[61] Louis Sobel. 2015. *Secure Input Overlays: Increasing Security for Sensitive Data on Android*. Master's thesis. Massachusetts Institute of Technology, MA, USA.

[62] Yihang Song and Urs Hengartner. 2015. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*.

[63] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*. Denver, Colorado, USA.

[64] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *Proceedings of the International Conference on Dependable Systems & Networks*. Brazil.

[65] Tianhao Tong and David Evans. 2013. Guardroid: A trusted path for password entry. In *Proceedings of Mobile Security Technologies*. San Francisco, CA, USA.

[66] Trustonic. 2012. Trustonic TEE Trusted User Interface. (2012).

[67] Nico Weichbrodt, Anil Kurmus, Péter Pietzuch, and Rüdiger Kapitza. 2016. Async-Shock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Proceedings of the 21st European Symposium on Research in Computer Security*. Heraklion, Greece.

[68] Yubin Xia, Yutao Liu, Cheng Tan, Mingyang Ma, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. TinMan: Eliminating Confidential Mobile Data Exposure with Security Oriented Offloading *(EuroSys)*. New York, NY, USA.

[69] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P' 15)*. San Jose, CA, USA.

[70] Sileshi Demesie Yalew, Gerald Q. Maguire Jr., and Miguel Correia. 2016. Light-SPD : A Platform to Prototype Secure Mobile Applications. In *Proceedings of Workshop on Privacy-Aware Mobile Computing*. Paderborn, Germany.

[71] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y. Thomas Hou. 2016. CacheKit: Evading Memory Introspection Using Cache Incoherence. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P' 16)*. Saarbrucken, Germany.

[72] Ning Zhang, Kun Sun, Wenjing Lou, and Y. Thomas Hou. 2016. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P' 16)*. San Jose, CA, USA.

[73] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. 2016. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive* 2016 (2016), 980.

[74] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiaoyong Zhou, and Xiaofeng Wang. 2015. Leave me alone: App-level protection against runtime information gathering on Android. In *2015 IEEE Symposium on Security and Privacy*. 915–930.

[75] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. 2011. Taming Information-stealing Smartphone Applications. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*. Springer-Verlag, Berlin, Heidelberg, 93–107.