vTrust: Remotely Executing Mobile Apps Transparently With Local Untrusted OS

Yutao Tang, Zhengrui Qin, Zhiqiang Lin, Yue Li, Shanhe Yi, Fengyuan Xu, and Qun Li *Fellow, IEEE*

Abstract—Increasingly, many security and privacy-sensitive applications are running on mobile platforms. However, as mobile operating systems are becoming increasingly sophisticated, they are vulnerable to various attacks. In addressing the need of running high assurance mobile apps in a secure environment even though the operating systems are untrusted, this paper presents vTrust, a new mobile app trusted execution environment, which offloads the general execution and storage of a mobile app to a trusted remote server (e.g., a VM running in a cloud) and secures the I/O between the server and the mobile device with the aid of a trusted hypervisor on the mobile device. Specifically, vTrust establishes an encrypted I/O channel between the local hypervisor and the remote server. In this way, any sensitive data flowing through the mobile OS, which the hypervisor hosts, is encrypted from the perspective of the local mobile OS. To enhance the performance of vTrust, we have also designed multiple optimizations, such as output data compression and selective sensor data transmission. We have implemented vTrust, and our evaluation shows that it has limited impact on both user experience and the application performance.

Index Terms—Trusted Execution Environment, Virtualization, Mobile Computing.

1 Introduction

Mobile devices have become increasingly integral and ubiquitous in recent years, with over a billion active devices worldwide today [1], surpassing desktop computers as the most popular personal computing platform [2]. Inevitably, this trend has made many organizations, even though demanding high security, start to use mobile devices (e.g., smartphones, tablets, iPads) at daily work to access security and privacy-sensitive data due to the increased productivity and job satisfaction [3]. For instance, hospitals have allowed doctors and nurses to access patient healthcare records using mobile devices [4], [5]; government agencies and military have allowed classified documents accessed and processed with smartphones [6], [7], [8]; many businesses and financial agencies permit their employees to process confidential data on their mobile devices for convenience [9].

The Problem. Unfortunately, along with the convenience, mobile devices also bring unprecedented security challenges, especially for security-sensitive applications (apps in short). The new forms of malware targeting mobile devices are on the rise with the increasing popularity of mobile phones. For instance, mobile malware increased more than three times between 2015 and 2016 [10]. Attackers and cybercriminals have realized that mobile devices are easier targets than conventional computing platforms (e.g., desktops) because mobile devices are so resource-constraint that, in many cases, security is sacrificed for performance and convenience. These attacks, ranging from ransomware [11] to advanced persistent threats (APT) [12], can persistently and stealthily steal valuable data from mobile devices. Some of them can even get the root privilege [13]. Even security-sensitive departments, such as the military, may fail to protect the devices adequately. For example, most recently, Israeli military personals were reported being spied by attacks through trojanized apps in mobile devices [14].

Local Execution Solutions. Many researchers propose various approaches to protect sensitive data on mobile devices. One widely-used approach is to store all sensitive data remotely in the corporate server and use VPN [15] to access them, which extends the corporate network across a public network through encryption. As a result, employees can still securely connect to the corporate network when working remotely. Unfortunately, although this approach can protect the data during transmission, it cannot guarantee data security in mobile devices. For instance, a compromised OS can easily breach the sensitive data once it is downloaded into mobile devices. Another approach is the full-disk encryption [16], [17], aiming at securing the mobile storage. Several advanced schemes use a remote trusted server to store the decryption keys [18], [19] to ensure security in case that the device is lost. However, the decryption key and intermediate data, such as the decrypted content, are still available in plaintext in the main memory when apps are executed. Therefore, these solutions do not entirely prevent a compromised OS from accessing sensitive data.

Remote Execution Solutions. To tackle the security challenges, a promising direction is to offload security-sensitive apps to a server [8], such as using Virtual Network

- *Yutao Tang is with the Guilin University of Electronic Technology, Guilin, China.E-mail: Yutao.Tang@guet.edu.cn*
- *Zhengrui Qin is with Northwest Missouri State University, Maryville, MO. E-mail: zqin@nwmissouri.edu*
- *Yue Li is with Fackbook,Menlo Park, CA. E-mail: yli@cs.wm.edu*
- *Zhiqiang Lin is with Ohio State University, Columbus, OH. E-mail: zlin@cse.ohio-state.edu*
- *Shanhe Yi is VMWare, Palo Alto, CA. E-mail: yshanhe@vmware.com*
- *Fengyuan Xu is with Nanjing University, Nanjing, China. E-mail: fengyuan.xu@nju.edu.cn*
- *Qun li is with the College of William and Mary, Williamsburg, VA. E-mail: liqun@cs.wm.edu*

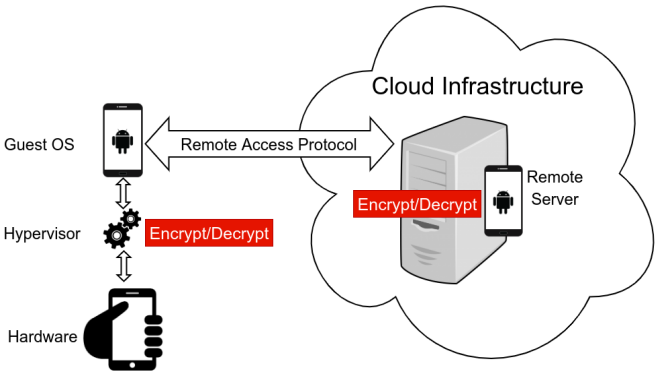


Fig. 1: vTrust solution.

Computing (VNC) [20] or Secure Virtual Mobile Platform (SVMP) [21]. This scheme sandboxes the execution and storage of sensitive apps in a verified remote execution environment. It uses a proxy to transmit I/O data (e.g., sensing, touchscreen input, and screen output) at the mobile OS. This solution does not leave any sensitive data in the local memory or the local disk. Thus, a compromised OS cannot directly access these data. Furthermore, the security of the remote execution environment can also be enhanced through existing security infrastructures. In addition, it facilitates central audit and supervision of sensitive data. Unfortunately, a severe drawback to the existing remote execution solutions is that they do not consider the I/O protection. I/O exposure opens up opportunities for attackers to intercept sensitive information from the input and output data. Without any proper I/O protection, the sensitive apps are far from being immune to local OS compromise. For example, the malicious OS can collect sensitive data through screenshots [22] or sensors [23], [24], [25], [26].

Our Solution. This paper proposes vTrust (V for Virtualization), a new Trusted Execution Environment (TEE) for mobile apps. Like existing remote execution solutions, vTrust enforces data security by outsourcing the computation and storage of a sensitive app into a security-enhanced virtual machine (VM) running on a remote trusted server (e.g., a VM managed by an enterprise). In addition, vTrust leverages the virtualization extension on the mobile device to protect I/O data. As Figure 1 shows, vTrust establishes an encrypted I/O channel between the remote server and the local hypervisor. It encrypts all input data in the hypervisor before sending them to the local mobile OS, and these data will be decrypted in the remote server. The output data works similarly in the opposite direction. In this way, vTrust protects the local mobile I/O from unauthorized access from local mobile OS. Meanwhile, vTrust allows users to install non-sensitive apps on the local mobile and provides a mechanism to seamlessly switch between (non-sensitive) local apps and (sensitive) remote apps.

The benefits brought by vTrust are prominent. In terms of security, vTrust remotely sandboxes sensitive apps and completely prevents a compromised local OS from accessing the memory and storage of these apps. Virtualizing the execution environment on a server also makes it feasible to deploy powerful but resource-hungry security enhancements, such as anti-virus, VM-introspection, and data flow tracking techniques. Meanwhile, it is resilient to mobile device losses since the data is not stored locally at all. Furthermore, vTrust makes management much more straightforward. All VMs on the server are within the same network and fully controlled by the IT department of an organization. Access to data can be granted or revoked at any time, even without the mobile devices' physical presence. The IT department can also easily track the data flow to ensure users are handling them properly. Additionally, it provides a manageable way to upgrade the system and apply system patches.

As a proof of concept, we have built a prototype of vTrust on an ARM-based development board and a remote server with virtualized Android for x86. To further improve vTrust, we have also applied multiple optimizations, such as output data compression and selective sensor data transmission. Through comprehensive analysis and experiments, we have evaluated the security, efficiency, and overhead of vTrust. Our experimental results show that vTrust can defend against various attacks with little overhead on the protected apps.

In short, we make the following contributions.

- We design a novel virtualization-based TEE— vTrust, which offers easier management and stronger protection by executing sensitive mobile apps in a trusted remote server with secure I/O protected by the local hypervisor.
- We have implemented a prototype of vTrust on an Arndale development board, and a server was running Android-X86. We have also developed several optimizations to enhance the performance of vTrust.
- We have evaluated the performance of vTrust and observed that vTrust incurs little overhead. In addition, there are little user experience change thanks to the transparency offered by vTrust.

The rest of the paper is organized as follows. Section 2 gives an overview of our system vTrust. Section 3 details the system design, and Section 4 presents the implementation details. Section 5 provides the evaluation of our system. Section 6 analyzes the security of our system. Section 7 discusses some limitations of our systems and future improvements. Section 8 compares the related work. Section 9 concludes the paper.

2 Overview

This section describes our design goals, system overview, assumption, threat model, and scope.

2.1 Design Goals

While there are many ways of designing a TEE for mobile apps, we seek to achieve the following objectives:

- **Full Protection**. A compromised mobile OS cannot steal any data of sensitive apps, no matter whether through peeking at intermediate data in memory or the I/O or the persistent data in local storage.
- **Transparent to Apps**. The deployment of our vTrust does not require any modification on the mobile apps, such that legacy apps can run in our system.

- **Seamless Switching**. vTRUST must concurrently support both protected apps running on a server and unprotected apps running in the untrusted mobile device. It needs to ensure a seamless switch at the mobile client between protected and non-protected apps.
- **Low Overhead**. The protected apps must function properly with an acceptable performance overhead.

2.2 Assumptions, Threat Model, and Scope

Assumptions. We consider a trusted computing base (TCB) that includes the hypervisor at the mobile side and the VMs at the remote side. We assume that the hypervisor is secure and trusted (ideally, it is verified). Note that it is also a one-time effort to deploy the hypervisor on the mobile device. Our assumption is practical since the hypervisor only needs to provide a virtual environment for Guest OS and encrypt/decrypt I/O data. It does not offer any service interfaces to the outside world and is transparent to the users. We also assume the users are willing to sacrifice some performance and resolution for security.

The remote VMs are assigned to the users and maintained by the IT administrators. Each VM is carefully protected and monitored by advanced techniques, such as anti-virus software, fine-grained system logs, firewalls, and intrusion detections. We also assume the apps installed on the VMs are carefully developed and verified. Thus they are trusted. The IT administrator installs the apps and possible tools on the VMs for the users and disallows customized installation of any un-trusted apps.

Threat Model. We assume the mobile OS is not trusted. Users may download a malicious app even through an official app store. One recent example is app InstaAgent [27], which stole Instagram user credentials and sent them to a third-party server without the user's knowledge. Furthermore, the mobile device itself may have vulnerabilities, through which some attackers can even get the root privilege and then access all the resources available to the mobile OS.

Scope. We provide this solution for organizations that have high requirements for security. vTRUST aims to guarantee data security rather than attack prevention. vTRUST cannot defend against malicious attacks, such as DoS, but it still can ensure data security and notify the users of the presence of the attacks at the same time.

2.3 vTRUST Terminology

A key enabling technology of vTRUST is virtualization. Both the local mobile device and the remote server leverage virtualization for different purposes. The mobile device uses it to protect I/O data from being accessed by the local untrusted mobile OS, and the server uses it for multiplexing (and other security such as isolation and introspection). To avoid potential confusion, in the rest of the paper, we use the "Hypervisor" to refer to the hypervisor on the mobile device, the "VM" to refer to the VM on the server, and the "mobile OS" to refer to the local mobile OS.

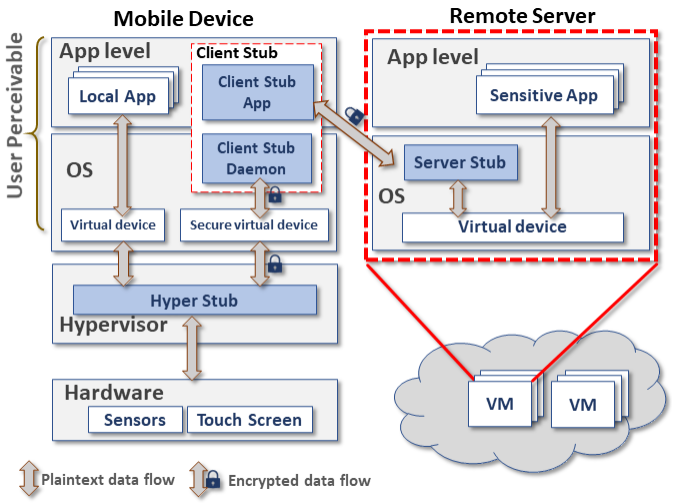


Fig. 2: An Overview of vTRUST.

3 Detailed Design

In this section, we present the design of vTRUST. We will first introduce the architecture of vTRUST, then describe the design of the key components in vTRUST

3.1 vTRUST Overview

Figure 2 shows the overview of vTRUST. At a high level, vTRUST uses a client-server architecture and consists of three key components: *Hyper Stub* and *Client Stub* running in the mobile device, and *Server Stub* running in the remote server (e.g., a VM).

The server in vTRUST can host many VMs for multiple mobile devices, atop each VM runs an Android mobile OS. For a more straightforward presentation, we only show a single VM instance in Figure 2. Meanwhile, in practice, a mobile device can correspond to multiple isolated VMs to provide a higher level of security, and the server can be multiplexed to support many mobile devices.

The mobile device runs a Hypervisor, which hosts a single mobile OS as the Guest OS. An end-user can only interact with the mobile OS, which occupies the whole screen. The Hypervisor is entirely invisible to the end-user and only performs encryption and decryption for security-sensitive apps. It is worth pointing out that, since the Hypervisor hosts only one Guest OS, the Guest OS can achieve a performance being nearly native [28].

When end-users start to access security-sensitive apps, vTRUST enters a particular mode, called Shield Mode. In this mode, vTRUST protects sensitive apps against an untrusted OS by encrypting all input/output data going through the untrusted OS. Specifically, on one hand, any input from the mobile device (e.g., sensors and touch actions) will be encrypted in *Hyper Stub*, then be delivered to the *Client Stub*, and further be transmitted to *Server Stub*. Then, the *Server Stub* decrypts the data and provides it to the app. On the other hand, if the sensitive app has any output, the data will be delivered to the user through the same channel(i.e., encrypted in *Server Stub* and forwarded to the *Client Stub*, and then decrypted at the *Hyper Stub*). Therefore, the untrusted mobile OS can only view encrypted data of

the sensitive apps while the user and the sensitive app can view the unencrypted data.

3.2 Key Components

vTRUST includes three key components: *Server Stub*, *Client Stub*, and *Hyper Stub*. The following paragraph will elaborate on each of them respectively.

Server Stub. The *Server Stub* runs as a system service of the Android Framework on the VM, and it tunnels the communication between the *Client Stub* and the sensitive apps. Specifically, it mainly focuses on two functionalities. First, it communicates with *Client Stub*, including receiving and decrypting user input data from *Client Stub* (e.g., touch actions and sensor readings), and encrypting output data (i.e., audio and screen frames) and sending them to *Client Stub*. Second, it communicates with the sensitive apps, including sending the received input data to or gathering output data from the sensitive app. Since the sensitive apps have no awareness of the *Server Stub*'s existence, they only communicate with the underlying Android framework for I/O like they usually do. Therefore, to make the communication transparent, we create several virtual devices in the VM and feed the data from the *Server Stub* to these virtualized devices. In this way, the above-lying sensitive apps can transparently consume data from the mobile device as if the data were generated locally.

Client Stub. As the figure shows, *Client Stub* includes two parts. The upper part, we call it *Client Stub App*, is a standard mobile app running at the App level. *Client Stub App* provides users a graphic portal to access sensitive apps running on the server. Through *Client Stub App*, users can connect to the remote server via traditional methods, such as VPN. After establishing the network connection between the *Client Stub* and the *Server Stub*, the client screen switches to show the decrypted frames transmitted from the VM, and users can interact with *Client Stub App* to control sensitive apps, just like a VNC or remote desktop app. The lower part, we call it *Client Stub Daemon*, are two typical Android Daemon processes which manage input and output data, respectively. *Client Stub Daemon* runs in the system level and plays a messenger role between *Client Stub App* and *Hyper Stub* since they cannot directly transfer messages to each other. The channel setup between *Client Stub App* and *Client Stub Daemon* is relatively easy. They can pass data to each other via Android API such as "Intent". However, the communications between *Client Stub Daemon* and *Hyper Stub* are very challenging due to no API between them. To solve this problem, vTRUST creates several virtual devices in Mobile OS and use them as channels to bridge *Client Stub Daemon* and *Hyper Stub*.

Hyper Stub. *Hyper Stub* is a set of processes running in the hyper level. As a part of TEE, *Hyper Stub* must be designed to process critical tasks only to minimize TEE size. In vTRUST, *Hyper Stub* only focuses on encrypt/decrypt and communication. Specifically, *Hyper Stub* encrypts input data and forwards it to *Client Stub Daemon* by writing them into corresponding virtual devices, and decrypt the output data received from *Client Stub Daemon* before sending them to hardware such that users can view the frames on the screen.

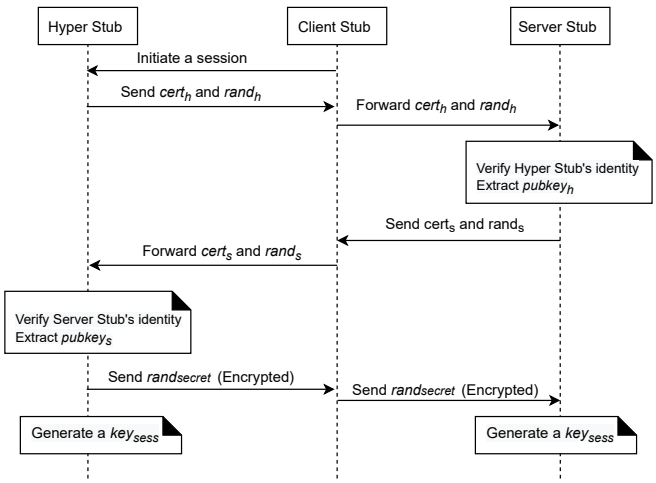


Fig. 3: The detail process of handshake.

We choose to use a *Client Stub* running on the mobile OS instead of the *Hyper Stub* as the I/O relay is mainly due to the following reasons. First, we aim to provide users with a minimum experience change. Having a *Client Stub* enables users to interact with sensitive apps running on the remote server in the same way as they are local. Second, we can rely on the Guest OS to provide a rich runtime environment for *Client Stub*. Third, the *Hyper Stub* can focus on the security part; it only needs to handle encryption/decryption and provide a virtual environment for the guest OS.

4 IMPLEMENTATION

In this section, we share the implementation details of vTRUST. Most of our implementation lies in how we transparently handle the I/O for sensitive apps. Therefore, we first describe how we handle the secure communication in §4.1, then the output in §4.2, and the input in §4.3.

4.1 Secure Communication

Secure communication between *Hyper Stub* and *Server Stub* is the foundation of vTRUST. As mentioned in the previous section, vTRUST encrypts all sensitive data that goes through mobile OS and does the decryption either in *Hyper Stub* or in *Server Stub*. The *Client Stub* only forwards data rather than processes it. Though this solution can effectively protect our sensitive data against a compromised mobile OS, it brings the following challenges to our system.

(I) **How to build a new session?** In vTRUST, the user can decide when to connect to or disconnect from sensitive apps by operating the portal provided by *Client Stub App*. When *Client Stub* receives a signal that the user wants to initiate a secure connection, it will assist *Hyper Stub* and *Server Stub* to handshake before they transmit data between each other. vTRUST uses an asymmetric encryption method for the handshaking process. Figure 3 shows the detailed steps. In a typical handshake, *Client Stub* first notifies *Hyper Stub*, which returns $cert_h$, a certificate of *Hyper Stub*, as well as $rand_h$, a random number. *Client Stub* then sends $cert_h$ and $rand_h$ to *Server Stub*, which verifies the identity

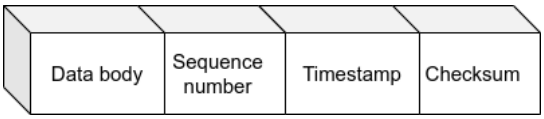


Fig. 4: The format of a typical packet in vTRUST.

of *Hyper Stub* via $cert_h$. Once the verification succeeds, *Server Stub* extracts $pubkey_h$, the public key of *Hyper Stub*, from $cert_h$, and send its own certificate, $cert_s$, and a random number, $rand_s$, to *Hyper Stub*. Similarly, *Hyper Stub* verifies *Server Stub*'s identity and extracts its public key $pubkey_s$. Finally, *Hyper Stub* sends a secret, $rand_{secret}$, encrypted with $pubkey_s$, to *Server Stub*. With $rand_h$, $rand_s$ and $rand_{secret}$, *Hyper Stub* and *Server Stub* can both derive the same key, key_{sess}, and use it to exchange sensitive data. Note that both certificates are provided by the administrator and are not accessible by the attackers.

(II) **How to secure data transferring?** Once both *Server Stub* and *Hyper Stub* obtain key_{sess} in the handshaking stage, they can use it to encrypt data to be transferred with symmetric encryption algorithms for better performance. However, the encrypted data must go through an untrusted OS, which might tamper or discard the encrypted data, or even replace it with previous encrypted data. To solve this problem, we deliberately design the format of the transferring packet. As Figure 4 shows, each packet includes four segments. The first segment is the data body. The second and the third segments are a sequence number and a timestamp, respectively, which are utilized to defend against replay attacks. The last segment is a checksum, which is used for integrity check. **Algorithm 1** presents how an encrypted packet is processed. Furthermore, vTRUST has a session timeout rule. For instance, if the timeout is set to 5 minutes, *Hyper Stub* and *Server Stub* need to restart a new session once the session has been idling for 5 minutes.

(III) **How to separate sensitive from non-sensitive data?** vTRUST allows users to run both local apps and sensitive apps simultaneously to maximize user experience. This means *Hyper Stub* needs to know where the received data is from. To achieve this goal, we create a set of separated virtual devices, called secure devices, in Mobile OS for each sensitive input/output source. Secure devices are only used for exchanging sensitive data and can only be accessed by *Client Stub Daemon* at the system level. As the result, *Hyper Stub* can easily capture sensitive data and non-sensitive data by accessing secure devices and ordinary devices, respectively.

(IV) **How to switch between local apps and sensitive apps?** In vTRUST, the users can switch between a sensitive app and a local app the same way as they do between local apps, e.g., by pressing the "Overview/Recent" button, though they lead to different operations in the low level. When the user switches from a sensitive app to a local app, vTRUST needs to exit Shield Mode immediately such that the local app can receive input from the hardware and display its frames on the screen. To support this functionality, We override the

ALGORITHM 1: Packet Decrypting

```
encrypted ← getReceivedPacket()
isValid ← FALSE
status, decrypted ← decryptPacketMethod(encrypted)
if status = TRUE then
    body, seq_num, timestamp, checksum ←
      parseDataMethod(encrypted)
    if seq_num > current_seq then
        if getCurrentTimestamp() − timestamp ≤
          THRESHOLD then
            body_checksum ←
              calculateChecksumMethod(body)
            if body_checksum = checksum then
                isValid ← TRUE
            end
        end
    end
end
if isValid = FALSE then
    reportAndLogThisIncident()
    return ERROR
end
else
    current_seq ← seq_num
    return body
end
```

$onPause()$ method of *Client Stub App*, which is invoked during the switching process. In this method, *Client Stub App* temporarily stops receiving the output data from *Server Stub*, and notifies *Hyper Stub* to exit Shield Mode. Similarly, we override $onResume()$ method of *Client Stub App*, allowing vTRUST to re-enter Shield Mode and resume data transferring between *Server Stub* and *Hyper Stub* when the user switches back to a sensitive app from a local app. Note that vTRUST does not require *Server Stub* and *Hyper Stub* to restart a new connection. They can resume the previous session unless it is expired.

4.2 Processing the Output

The typical output of mobile apps is screen frames [29] and audios. Currently, we only implement the screen frame output for its prevalence since almost all apps desire screen display. Although vTRUST does not support audio output at this stage, we can easily add this feature into our system since the audio output can follow a similar procedure as screen frames. The process of how screen frames are processed in vTRUST is illustrated in Figure 5. In a typical Android system, all screen updates will eventually be sent to the frame buffer for screen rendering. Therefore, when a session starts, the *Server Stub* periodically fetches plaintext frames from the frame buffer located in the OS kernel, compresses them and converts them into a packet in the format described in Figure 4. Next, *Server Stub* encrypts these packets and sends them to the *Client Stub* on the mobile device through the network. On the mobile device side, the *Client Stub* forwards received packets to *Hyper Stub*. When detecting a new incoming packet, *Hyper Stub* decrypts it with key_{sess}, verify the integrity of decoded data, decompresses the data body to a frame, and sends the restored frame to the screen. During the implementation

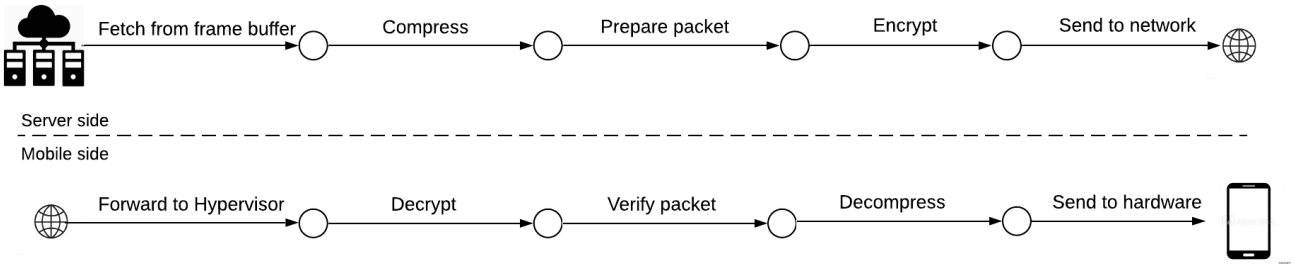


Fig. 5: Screen Frames Processing in vTRUST.

of the frame data transmission, several critical issues need to be addressed:

- **Encryption Algorithm.** A fast encryption algorithm is essential for vTRUST since it aims to provide a satisfactory user experience. In our implementation, we use the AES-128 block cipher [30] with GCM mode. The main reason is that AES is a mature and widely adopted encryption technique. Furthermore, AES-GCM is written in parallel and has lower encryption overhead than other AES variants (like AES-CBC). According to our evaluation in §5.1, AES-GCM is the most efficient encryption algorithms in our evaluation.
- **Data Compression.** Low latency is critical to vTRUST. To provide a decent frame rate even under non-ideal network conditions, we need to avoid sending huge frame data from the server to the mobile device. Thus, it is important to reduce the transmission data size in real-time. Fortunately, after analyzing the frames, we have found that many apps do not change their GUI significantly. As the result, we design a simple compression algorithm, which calculates the difference between two neighboring frames using exclusive-or operation, and then we use LZ4 [31] algorithm to compress the outcome to minimize the data load.
- **Frame Data Integrity.** The integrity check is crucial to vTRUST. An encrypted frame will never be processed by the *Hyper Stub* unless it passes the integrity check. For instance, if an encrypted frame is modified in mobile OS by an attacker, it will fail to pass the integrity check, and *Hyper Stub* will report this incident to *Server Stub*, which eventually reports to the administrator for further inspection. In vTRUST, we use SHA-256 [32] for integrity check.

4.3 Processing the Input

Mobile devices typically have two types of input: touchscreen input and sensor input. In the following, we describe how vTRUST handles them correspondingly.

Touchscreen. The touchscreen data includes the user's raw touch actions on the touch screen, such as PRESS/RELEASE, x, y. It flows in the opposite direction of the frame data, except that the touch screen data does not need compression/decompression due to its small size. Specifically, in Shield Mode, the *Hyper Stub* will encrypt the touchscreen data from the hardware and then feeds the encrypted data to the secure virtual devices supporting the local mobile OS. The ciphertext will be forwarded to the *Client Stub*, and delivered to the *Server Stub*, which further decrypts and adjusts the data field in the event and injects the event directly to the upper-lying Android system through a system API (i.e., `InputManager.injectInputEvent`). In most cases, touch actions from mobile are consumed by sensitive apps. One exception is the reserved "Overview/Recent" button. When the user presses this button, which is apparently not for sensitive apps themselves, *Server Stub* will capture this signal and notify *Client Stub Daemon* to trigger "Overview/Recent" button's event in mobile OS. As the result, the user will switch to local apps and exit Shield Mode.

Similar to the screen frames, the touchscreen input must be encrypted in *Hyper Stub*. In our implementation, we leverage the Prefix Cipher [33], a well-known Format-Preserving Encryption (FPE) scheme based on block ciphers, to encrypt the touchscreen data with the data size being preserved. A Prefix Cipher algorithm is proven to be as strong as the block cipher [33]. We use a standard AES encryption to construct the Prefix Cipher algorithm. Specifically, the ciphertext of our encryption algorithm is generated by applying AES encryption to the plaintext over a key and then taking the order of the AES ciphertext as the ciphertext of our algorithm. Unlike other encryption algorithms, a mapping table is maintained to decrypt the ciphertext since this process is not invertible. We choose Prefix Cipher over other FPE algorithms because of its high efficiency. It requires only one table lookup to decrypt the message, which is necessary when the data volume is high.

Sensors. In the mobile device, sensor data offers input from multiple dimensions and provides richer functionalities to mobile apps. Though the sensors bring a wealth of advantages, it has been shown that sensor data can be leveraged to launch various attacks (e.g., [23], [24], [25], [26], [34]). Therefore, vTRUST has to secure the sensor input.

However, unlike touchscreen data exclusively consumed by one app, sensor data is shared among all apps requesting it. In Shield Mode, local apps cannot receive any sensitive sensor data since vTRUST encrypts and redirects these sensitive input to secure devices for data protection; that is, local apps can only receive non-sensitive sensor data. If we blindly forward all sensor data to *Server Stub* in Shield Mode, it will not only increase the computation/network overhead of vTRUST, but also affect local apps using sensor data that are not requested by the sensitive app at the same

time, leading to a reduced user experience. To mitigate this effect, we choose only to encrypt and transmit the sensor data requested by the sensitive apps. When sensitive apps no longer need certain sensors, vTRUST will release them immediately such that local apps may use them.

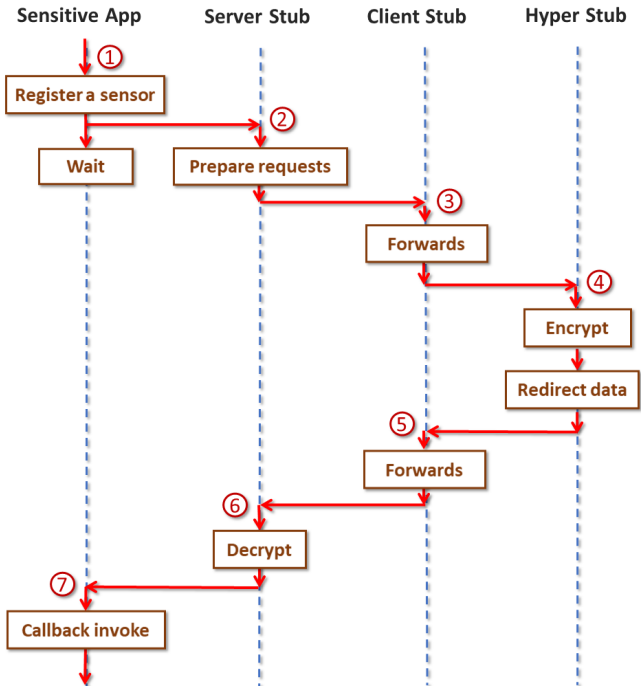


Fig. 6: Sensors Input Processing in vTRUST.

The detailed steps of how vTRUST handles sensor input is illustrated in Figure 6. **Step** ① The sensitive app requests a type of sensor data, such as the accelerometer data, and register a listener to the sensor manager in the Android system of the VM. **Step** ② Our modified sensor manager notifies the *Server Stub* of the request from the sensitive app. **Step** ③-**Step** ④ The *Server Stub* sends the requested sensor type to *Hyper Stub*. Upon receiving the request, *Hyper Stub* will encrypt the sensor data and redirect it from the ordinary virtual device to the corresponding secure device. **Step** ⑤-**Step** ⑦ The data is relayed back to the sensitive app through *Client Stub*, *Server Stub*, and the sensor input system. Finally, the sensitive app consumes these sensor data.

Unlike the touchscreen with an existing system API to input the data, injecting the sensor data is more challenging. Fortunately, we accomplish this process by leveraging the Hardware Abstraction Layer (HAL) [35] in the Android system. In particular, HAL lies between the Android framework and the Linux kernel, which encapsulates the raw data from the device drivers into events for consumption from the above Android framework services, where these events are further delivered to apps. In light of this, we implement a special sensor HAL module in the Android system on the VM to handle all kinds of sensor input without any modification to the upper layer apps and services. Unlike normal HAL modules, which receive data from drivers in the Linux kernel, our sensor HAL modules communicate with the *Server Stub* to accept input data through an internal socket channel.

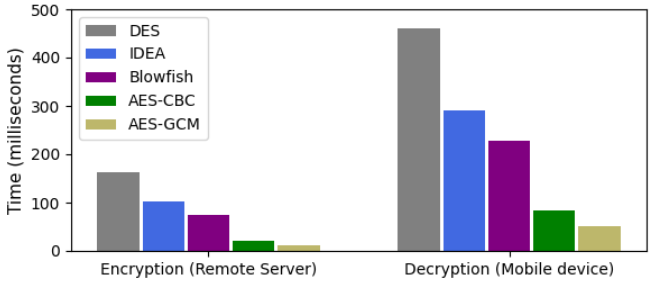


Fig. 7: Comparison of different encryption algorithms.

5 EVALUATION

In this section, we present the evaluation results. Specifically, we evaluated the performance of vTRUST using an Arndale development board [36] as the mobile device since it adopts Exynos 5250 SoC, which supports hardware virtualization. Also, the Exynos 5250 SoC has a Samsung Exynos 5 dual core processor running at 2.0GHz with 2GB of RAM. Meanwhile, the development board runs a lightweight Linux as the host OS and establishes its virtualization environment using KVM and QEMU [37]. KVM and QEMU provide a hardware abstraction layer, upon which an Android 4.1.1 is installed as the guest OS. The remote server used in our experiments is a desktop server with 4.2GHz Intel i7-6700K CPU, 32GB RAM, and 4TB disk. In addition, the development board is equipped with an external accelerometer [38] and 7 inch touchscreen [39]. As for the server, we run Android-x86 VMs on the remote server by using the VMware workstation hypervisor.

In our experiment, we first performed the microscopic measurement of vTRUST that includes the overhead from the encryption and decryption (§5.1), the compression and decompression (§5.2), and then at the macroscopic in terms of throughput (§5.3) and the responsiveness (§5.4).

5.1 Encryption and Decryption

The encryption algorithm is critical for the performance of vTRUST. To ensure optimal user experience, we need to carefully choose the specific encryption algorithm. For input encryption, as we use a mapping table on the input value, the overhead can be negligible due to the simplicity of the algorithm and moderate data size. On the other hand, the output encryption and decryption procedures can introduce significant latency as the data being encrypted are screen frames, which are normally quite large.

In our experiment, the guest OS on the mobile device has a 640x480 resolution with RGB-565-color encoding, giving a frame size of $614,400$ bytes. We evaluate five encryption algorithms, which are DES, IDEA, Blowfish, AES-128 with CBC mode, and AES-128 with GCM mode. For each of the algorithm, we transmit 100 frames and record the average delay introduced by encryption/decryption procedures. The results are plotted in Figure 7.

Our experiments show that the five algorithms have significant difference regarding computation overhead, which is stemmed from different algorithm complexity. Compared to other 4 conventional algorithms, AES-128 with GCM

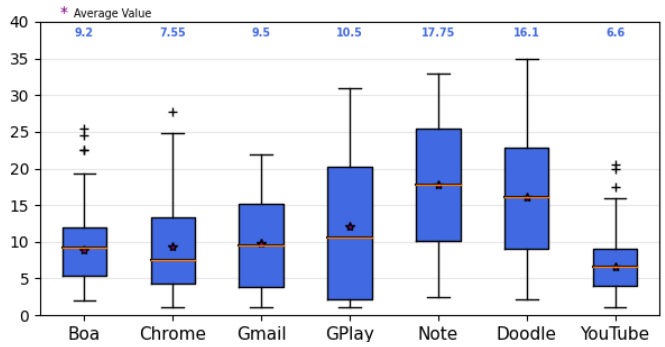


Fig. 8: Compression ratios.

mode has lower computation overhead. Another observation is that the encryption is faster than the decryption for each of the five algorithms, due to the following two reasons: (1) the server where the encryption is conducted is more powerful than the mobile device where the decryption is done; and (2) the server equipped with Intel processor can take advantage of AES-NI [40], which introduced a set of new AES (Advanced Encryption Standard) instructions for hardware acceleration.

5.2 Compression and Decompression

In this experiment, we measured the compression ratios of our compression algorithm for different apps.

To cover more use cases, we deliberately select 7 typical mobile apps from different genres in our evaluation (as seen in Figure 8). In our experiment, we leverage the `MobiPlay` tool [41] to generate the workload. This tool records a user's interaction on the tested app as a sequence of high level events and can then replay these recorded events at a later time. To record the initial workload, we manually operate each app for 60 seconds. For example, when testing `Chrome`, we scroll a web page up and down to emulate user activities when browsing. Note that the same workload is used for the throughput and responsiveness evaluations in §5.3 and §5.4.

Figure 8 illustrates the box-plots of the compression ratios for the 7 apps. As shown in the figure, apps can get decent compression ratio with our method. Some apps, like `Gplay`, `Notes`, and `Kids Doodle`, have compression ratio greater than 10.

5.3 Throughput

Here we evaluate the frame throughput of vTRUST. The frame throughput is defined as the number of frames that are transmitted from remote server and shown on the mobile device screen per second. Frame throughput is an important factor that directly impacts the user experience. For instance, a low frame throughput may lead to jagged motions on the mobile device screen, which is considered as poor user experience. As the frame throughput is mainly affected by encryption overhead, compression ratio, and network bandwidth, we tune our experimental settings with 3 different network bandwidth: (a) 100 Mbps, (b) 25Mbps, and (c) 10Mbps. For each network bandwidth, we measure the frame throughput in *uncompressed + encrypted* and *compressed + encrypted* forms using the same apps and replay tools mentioned in §5.2.

Figure 9 shows that the throughput of uncompressed frame is significantly slower than compressed frame under condition (b) and (c) . With the help of our compression technique, vTRUST can maintain a relatively high throughput even under a low bandwidth condition. Meanwhile, we can find that apps with lower compression rate usually have poor FPS. Although the FPS achieved by vTRUST is not comparable to local execution, we consider it is acceptable for the security-sensitive applications with the security gain.

5.4 Responsiveness

Besides the throughput, the response time is also critical in vTRUST since long response time significantly hinders user experience, causing many usability issues. In light of this, we conducted an experiment to measure the input latency and output latency of vTRUST.

The input latency is defined as the duration from the time when data, such as sensor data, is received in hypervisor to the moment when it is received by sensitive app. To evaluate the input latency, we further generate 500 touchscreen events (by pressing touchscreen randomly) in addition to our initial workload on the client device under the 3 scenarios. Note that the sensor data is very similar to touchscreen data in size, and evaluating the touchscreen is representative for all input data. Figure 10(a) illustrates the results. Since each touchscreen event is very small in size, the network latency introduced is small, which gives us a total input latency of less than 30ms. Such a latency is acceptable for most users.

Similarly, we measure the output latency, which is the duration from the time when the frame is written into the frame buffer to the moment when it is shown on the mobile screen, and the results are shown in Figure 10(b). The output is from the `Chrome` workload, which has the largest output size after compression, and thus the largest latency. As expected, the latency in network transmission is higher than the input latency, varying from 90ms to 160ms in average under different network conditions. This latency is noticeable. However, for most sensitive apps, it is still usable and fairly responsive.

6 Security Analysis

Having presented the design and implementation of vTRUST, next, we discuss how and why our system can defend against various attacks and keep the sensitive apps secured under untrusted mobile operating systems.

As stated in the threat model in §2.2, an attacker can access or modify all resources that the local mobile OS is entitled to. In addition, she can eavesdrop or manipulate the network connections between the mobile device and the server. But, the attacker cannot read the memory or storage of sensitive apps, as the apps are running on another machine, which is out of the attacker's control. In addition, the attacker is also isolated from the critical resource of the hypervisor thanks to the virtualization technology on ARM. Consequently, vTRUST's architecture intrinsically

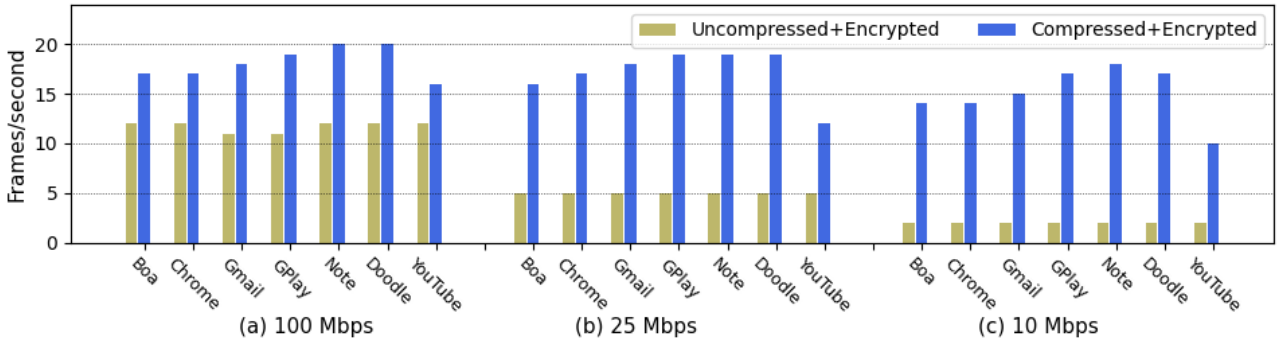


Fig. 9: FPS of vTRUST output.

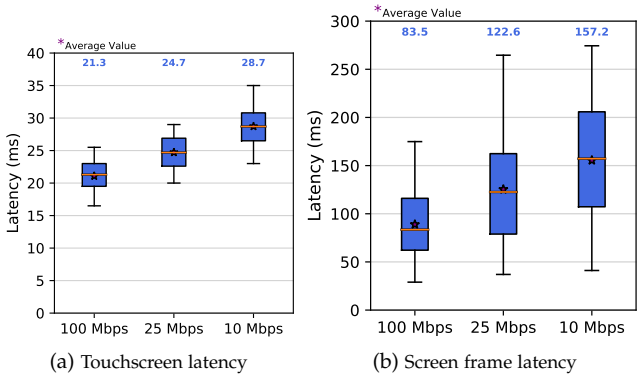


Fig. 10: vTRUST responsiveness.

secures both the storage and the memory of sensitive data. Therefore, the only attack interface of vTRUST is the I/O data, which needs to go through the mobile OS controlled by the attacker. In order to secure this part, vTRUST encrypts all I/O data that the attacker can access so that the user can securely access the sensitive app from their mobile device.

This paragraph lists several typical attacks targeting mobile OS and explains how vTRUST protects the sensitive apps.

1) **Tampering.** vTRUST relies on the mobile OS to forward input and output data from remote VM to Hypervisor. Hence, an attacker could tamer with the received data before sending it to Hypervisor. For this type of attack, vTRUST can easily detect it through integrity check since the attacker does not have our key_{sess}. Any incoming data that failed passing integrity checking will be discarded, and vTRUST will log this incident for further analysis.
2) **Replay.** The client OS could replay old frames to the Hypervisor. In this case, although the received data can pass integrity check, vTRUST still can detect this type of attack by checking its sequence number and timestamp. If its sequence number is less than the current one or the timestamp is too old, vTRUST will regard it as an incident.
3) **Fast Switching.** The client OS may quickly switch between the privileged app and a local app. In this scenario, the attacker cannot get any frame or touchscreen data. However, it is possible that the attacker can infer the sensor data that has recently been used by the sensitive app. When client OS switch to a local app, since vTRUST only encrypt the sensor data when a sensitive app is using it and switching will force vTRUST to release the sensor. To mitigate this problem, we can either rely on the administrator to set up rules specifying which sensors are available to the local app or implement a delay release strategy in which the Hypervisor will only release this sensor after a certain period of time, e.g., 5 minutes after vTRUST exits Shield Mode.
4) **Interrupting.** The client OS might throw away the packets intended to the Hypervisor or VM. When this type of attack occurs, vTRUST does not leak any information, and the user will quickly notice this problem. Currently, We rely on the user to report this incident.

7 Limitations and Future Work

Limitations. vTRUST is still not perfect and it has a number of limitations. First, the I/O encryption cannot fully protect the sensor data. Persistent data such as GPS, temperature, or light sensor data, can be easily inferred from local apps since they do not get drastically changed. Instead, vTRUST focuses on protecting transient sensor data (e.g., the gyroscope or the accelerometer), since these data has been used in many side-channel attacks [34], [42].

Second, vTRUST transmits screen frames through the network. The performance overhead, in terms of FPS, is non-negligible, especially in the case that the screen content changes rapidly. Therefore, high FPS demanding apps, such as video-playing apps, may suffer from noticeable user experience degradation. Though not ideal when running on the server, these apps are normally considered non-sensitive and should be locally installed to ensure high quality of service. On the other hand, sensitive apps, such as banking or email applications, are usually more static in display, and vTRUST is able to provide a more satisfactory user experience.

Third, in our current prototype implementation, the vTRUST server runs on Android for X86, which cannot run ARM-based apps. However, this limitation would no longer exist in an ARM-based server. On the other hand, we also note that more and more apps start to support X86 platform.

Future Work. There are a number of avenues to improve VTRUST. In addition to address the above limitations, we can also work on improving its performance and security.

- **Compression Overhead**. Currently, VTRUST still incurs less satisfactory compression ratio for apps that have intensive output change, such as `Youtube`. However, adopting stronger compression algorithms may introduce longer delay. To reduce the time needed for compression while maintain high compression ratio, we can leverage hardware-assisted lossless compression techniques, such as H.265 [43]. These techniques are very efficient, we believe VTRUST could have a much shorter latency and higher FPS with them.
- **Advanced protection**. We have mentioned that VTRUST can be built on very strong security infrastructure on the server side. However, there is still a limited number of security infrastructures that protect Android system due to the fact that most Android devices are too resource-restrained to apply advanced security measures in the device itself. Interestingly, VTRUST opens up new opportunities for adopting Android-specific security products, e.g., Android framework level logging and tracing systems, and more powerful data flow tracking tools like TaintDroid [44], in our remote VMs.

8 Related Work

Computation offloading approaches. Many projects [18], [45], [46], [47], [48], [49], [50] seek to protect the sensitive data with the assistance of a remote cloud. CleanOS [18] monitors the usage of sensitive data and encrypts data that are temporarily not used. To avoid the leak of encryption key in case of device losses, CleanOS stores the encryption keys in a trusted cloud and downloads them only when necessary. TinMan [45] goes further along this direction. It keeps track of the processes that access the sensitive data, and migrates these processes to a highly-secured environment in the cloud for remote execution. When these processes finish accessing sensitive data, they will be migrated back to the mobile device. In this way, the sensitive data is protected from the local untrusted OS. However, these solutions all focus on protecting non-user-interactive data, and cannot be applied to protect I/O data.

Systems	C1	C2	C3	C4	C5	C6	C7	C8
CleanOS [18]	✗	✗	✓	✗	✓	✓	✓	✗
TinMan [45]	✓	✗	✓	✗	✓	✓	✓	✗
TrustZone [51]	✓	✓	✓	✗	✓	✗	✓	✗
Overshadow [52]	✓	✓	✓	✗	✗	✓	✗	✓
OSP [53]	✓	✓	✓	✗	✓	✗	✗	✗
SGX [54]	✓	✓	✓	✓	✗	✗	✓	✓
VNC [20]	✓	✗	✓	✓	✗	✓	✓	✓
VTRUST	✓	✓	✓	✓	✓	✓	✓	✓

C1: Memory protection
C2: I/O protection
C3: Storage protection
C4: Easy management
C5: Supporting mobile device
C6: Supporting legacy applications
C7: Securing data after device loss
C8: Resource-rich

TABLE 1: Comparison with the related work.

TrustZone-based solutions. Trustzone was first introduced in 2003 for ARM processors. As we have mentioned, this technology aims to provide a deterministic protection mechanism to protect apps from the untrusted OS running in the normal world. Many efforts [51], [55] take advantage of this feature and save the SCC in the secure world. Processes running in the normal world can only access the SCC by invoking a set of well-defined APIs. This design assumes that the secure world is fully trusted. Unfortunately, in practice, Trustzone is still vulnerable to attacks [56], especially when more SCC is put into the secure world [57].

Hypervisor-based solutions. Some works [58] attempt to enforce security policies and provide TEEs with the aid of a hypervisor. Systems like Overshadow [52] and CHAOS [59] aim to protect the whole process even when the OS is malicious. However, these techniques are designed for the PCs rather than the mobile systems, and they are also found being vulnerable to newly identified attacks [60]. While OSP [53] combines a hypervisor and TrustZone to provide an on-demand protection and secure I/O, it requires modification of existing apps. Meanwhile, unlike VTRUST that offers a centralized security management, OSP cannot achieve this.

Other hardware-based solutions. Intel recently introduced SGX [54] for app developers to protect their own sensitive code and data using a hardware protected secure enclave, in which the data remains protected even when the BIOS, virtual machine monitor, operating system, and device drivers are compromised. While SGX holds the greatest promises for TEE, it has been mainly applied in cloud computing (e.g., SGXBOUNDS [61] for shielded execution and VC3 [62] for secure analytics) and we have not witnessed how it can be used to protect mobile apps.

Remote Execution Solutions. Many existing applications, such as VNC [20], SVMP [21] and Rio [63], allow I/O or hardware sharing between different devices. For example, Rio enables two mobile devices to share their hardware resources, such as the camera, or the speaker. Though VTRUST and Rio provide similar functions, our system is security-oriented and has a different implementation. One key design of VTRUST is to securely transmit I/O data.

Summary. A summary of the comparison between VTRUST and the existing closely related efforts can be found in Table 1. We notice that VTRUST holds all of the capabilities compared, and the most closely related system is the VNC [20], especially from the user experience perspective. However, with VNC, untrusted operating systems are still able to view the I/O of the sensitive apps, which may open up attack opportunities for attackers.

9 Conclusion

We have presented VTRUST, a novel software-based trusted execution environment for mobile apps based on a server and a hypervisor. The key insight is to leverage virtualization in both mobile devices and servers to construct a secure execution environment across two trusted parties: the hypervisor on a mobile device and a remote server. VTRUST ensures no exposure of data in memory, storage, and I/O by delegating the mobile app computation and storage to the remote server and securing the I/O channel via encryption. As such, VTRUST protects the execution of sensitive apps from an untrusted operating

system. We implement a prototype of vTRUST and conduct extensive evaluations. Our experimental results show that vTRUST introduces little impact on user experience and the performance of security-sensitive apps.

ACKNOWLEDGEMENT

This project was supported in part by US National Science Foundation grant CNS-1816399 . This work was also supported in part by the Commonwealth Cyber Initiative, an investment in the advancement of cyber R&D, innovation and workforce development. For more information about CCI, visit cyberinitiative.org.

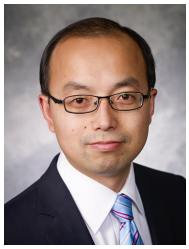
REFERENCES

[1] H. Leonard, "There will soon be one smartphone for every five people in the world," http://www.businessinsider.com/15-billion-smartphones-in-the-world-22013-2, February 2013.
[2] "Mobile marketing statistics compilation," http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/, Jul 2015.
[3] "Ibm mobile solutions drive digital innovation," http://www.ibm.com/mobile, September 2017.
[4] A. Nerminathan, A. Harrison, M. Phelps, K. M. Scott, and S. Alexander, "Doctors' use of mobile devices in the clinical setting: a mixed methods study," *Internal medicine journal*, vol. 47, no. 3, pp. 291–298, 2017.
[5] "Mobile security essential healthcare provider priority," https://healthitsecurity.com/news/mobile-security-essential-healthcare-provider-priority, 2017.
[6] "U.s. military phones: Android is system of choice," http://www.zdnet.com/article/us-military-phones-android-is-system-of-choice/, February 2012.
[7] "U.s. government, military to get secure android phones," http://www.cnn.com/2012/02/03/tech/mobile/government-android-phones/index.html, February 2012.
[8] "Digital government - bring your own device," https://obamawhitehouse.archives.gov/digitalgov/bring-your-own-device, August 2012.
[9] "Financial services rely on byod – how do they stay secure?" https://www.forbes.com/sites/louiscolumbus/2019/10/31/financial-services-rely-on-byod--how-do-they-stay-secure/?sh=49c9d8447ace, 2019.
[10] "Mobile advertising trojans exploiting super-user rights became the top mobile malware threat in 2016," https://www.kaspersky.com/about/press-releases/2017_mobile-advertising-trojans, January 2017.
[11] "One kind of android smartphone ransomware is behind a massive rise in malicious software," http://www.zdnet.com/article/, June 2017.
[12] "Advanced persistent threats now hitting mobile devices," https://www.networkworld.com/article/2173639/wireless/advanced-persistent-threats-now-hitting-mobile-devices.html, December 2013.
[13] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones: Attacks and implications," 2009.
[14] "Hackers are using android malware to spy on israeli military personnel," https://thehackernews.com/2017/02/android-malware-israeli-military.html, February 2017.
[15] "Why frame rate matters," https://gizmodo.com/why-frame-rate-matters-1675153198, Jan 2015.
[16] S. M. Diesburg and A.-I. A. Wang, "A survey of confidential data storage and deletion methods," *ACM Computing Surveys (CSUR)*, vol. 43, no. 1, p. 2, 2010.
[17] "Encrypting file system in windows xp and windows server 2003," https://technet.microsoft.com/en-us/enus/library/bb457065.aspx, 2003.
[18] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: Limiting mobile data exposure with idle eviction." in *OSDI*, vol. 12, 2012, pp. 77–91.
[19] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy, "Keypad: an auditing file system for theft-prone devices," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 1–16.
[20] "Virtual network computing," https://en.wikipedia.org/wiki/Virtual_Network_Computing, 2016.
[21] "Virtual smart phones in the cloud," https://svmp.github.io/, November 2014.
[22] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets," in *21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA*, 2014.
[23] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion." *HotSec*, vol. 11, pp. 9–9, 2011.
[24] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 113–124.
[25] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "Accessory: password inference using accelerometers on smartphones," in *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*. ACM, 2012, p. 9.
[26] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: your finger taps have fingerprints," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACm, 2012, pp. 323–336.
[27] "Instaagent app pulled after 'harvesting passwords'," https://www.bbc.com/news/34787402, Nov 2015.
[28] V. O. Systems, "Kvm on arm performance," http://www.virtualopensystems.com/en/products/kvm-performance/, November 2014.
[29] T. Li, C. An, X. Xiao, A. T. Campbell, and X. Zhou, "Real-time screen-camera communication behind any scene," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015, pp. 197–211.
[30] N.-F. Standard, "Announcing the advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, vol. 197, pp. 1–51, 2001.
[31] "Lz4," https://lz4.github.io/lz4/, Jul 2016.
[32] H. Gilbert and H. Handschuh, "Security analysis of sha-256 and sisters," in *International workshop on selected areas in cryptography*. Springer, 2003, pp. 175–193.
[33] J. Black and P. Rogaway, "Ciphers with arbitrary finite domains," in *Cryptographers' Track at the RSA Conference*. Springer, 2002, pp. 114–130.
[34] E. Novak, Y. Tang, Z. Hao, Q. Li, and Y. Zhang, "Physical media covert channels on smart mobile devices," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 367–378.
[35] "Android hardware abstraction layer documentation," https://source.android.com/devices/halref/index.html, Feburary 2015.
[36] Samsung, "Arndale board," http://www.arndaleboard.org/wiki/index.php/Main_Page, November 2014.
[37] V. O. Systems, "Kvm virtualization on arndale development board," http://www.virtualopensystems.com/en/solutions/guides/kvm-virtualization-on-arndale/, November 2014.
[38] "Serial accelerometer dongle," https://www.sparkfun.com/products/retired/10537, November 2014.
[39] "Raspberry pi touch display," https://www.raspberrypi.org/products/raspberry-pi-touch-display/, November 2016.
[40] "Introduction to intel® aes-ni and intel® secure key instructions," https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-aes-ni-and-intel-secure-key-instructions.html, Jul 2010.
[41] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 571–582.
[42] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals." in *USENIX Security*, 2014, pp. 1053–1067.
[43] "High efficiency video coding," https://en.wikipedia.org/wiki/High_Efficiency_Video_Coding, Jul 2016.
[44] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smart-

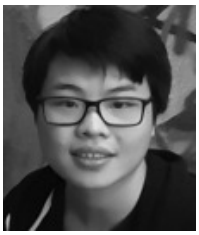
phones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.
[45] Y. Xia, Y. Liu, C. Tan, M. Ma, H. Guan, B. Zang, and H. Chen, "Tinman: Eliminating confidential mobile data exposure with security oriented offloading," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 27:1–27:16. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741977
[46] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *MobiSys '10*, 2010.
[47] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.
[48] R. Kemp, N. Palmer, T. Kielmann, and H. E. Bal, "Cuckoo: A computation offloading framework for smartphones." in *MobiCASE*. Springer, 2010, pp. 59–79.
[49] Y. Tang, Y. Li, Q. Li, K. Sun, H. Wang, and Z. Qin, "User input enrichment via sensing devices," *Computer Networks*, vol. 196, p. 108262, 2021.
[50] A. Bandi and J. A. Hurtado, "Big data streaming architecture for edge computing using kafka and rockset," in *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*. IEEE, 2021, pp. 323–329.
[51] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *ASPLOS '14*, 2014.
[52] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *ASPLOS '08*, 2008.
[53] Y. Cho, J.-B. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices." in *USENIX Annual Technical Conference*, 2016, pp. 565–578.
[54] "Intel® software guard extensions," https://software.intel.com/en-us/sgx, Jan 2017.
[55] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "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*. ACM, 2014, pp. 90–102.
[56] Y. Chen, Y. Zhang, Z. Wang, and T. Wei, "Downgrade attack on trustzone," *arXiv preprint arXiv:1707.05082*, 2017.
[57] S. C. Misra and V. C. Bhavsar, "Relationships between selected software measures and latent bug-density: Guidelines for improving quality," in *International Conference on Computational Science and Its Applications*. Springer, 2003, pp. 724–732.
[58] E. Bugnion, J. Nieh, and D. Tsafrir, "Hardware and software support for virtualization," *Synthesis Lectures on Computer Architecture*, vol. 12, no. 1, pp. 1–206, 2017.
[59] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, and W. Mao, "Tamper-resistant execution in an untrusted operating system using a virtual machine monitor," 2007.
[60] Y. Cheng, X. Ding, and R. Deng, "Appshield: Protecting applications against untrusted operating system," *Singaport Management University Technical Report, SMU-SIS-13*, vol. 101, 2013.
[61] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "Sgxbounds: Memory safety for shielded execution," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 205–221.
[62] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 38–54.
[63] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A system solution for sharing i/o between mobile systems," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 259–272. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594370

Yutao Tang Dr. Tang is a researcher with the School of Artificial Intelligence at Guilin University of Electronic Technology. He got his Ph.D. in Computer Science from the College of William and Mary. His research focuses on edge computing, deep learning, and network security.

Zhengrui Qin Dr. Qin is an Associate Professor with the School of Computer Science and Information Systems at Northwest Missouri State University. He earned his Ph.D. in Computer Science from the College of William and Mary. His research interests lie in cybersecurity and mobile computing.



Zhiqiang Lin Dr. Lin is a Professor of Computer Science and Engineering at The Ohio State University. His research focuses on cybersecurity, particularly on software security and trusted computing, and most recently on mobile, IoT, cloud, and blockchain security. Dr. Lin is a recipient of NSF CAREER Award and AFOSR YIP Award.



Yue Li Dr. Li is a Senior Research Scientist at Meta. He got his Ph.D. in Computer Science from the College of William and Mary. His research focuses on network security and mobile computing.

Shanhe Yi Dr. Yi is an Research Scientist at Meta. He got his Ph.D in Computer Science from the College of William and Mary. His research focuses on wireless networking and edge computing.

Fengyuan Xu Dr. Xu is a Professor with the Department of Computer Science and Technology at Nanjing University. He earned his Ph.D. in Computer Science from the College of William and Mary and received the Distinguished Dissertation Award in Natural and Computational Sciences.His research focuses on the Real-Time Cyber-Physical Metaverse, Autonomous Trust and Privacy Mechanisms, and Intelligent Edge Computing Systems.

Qun Li Dr. Li is a Professor of Department of Computer Science at William and Mary. He got his Ph.D from Dartmouth College. His current research interests include wireless networks, IoT, edge computing, pervasive computing, and security and privacy. Dr. Li is a recipient of the NSF CAREER Award and IEEE fellow.